Real-Time Systems - Assignment 02 Group 9

Zhengtao Huang (5833469, zhengtaohuang)
Code Commit Without Part IV (4ada2246)

Barry Tee Wei Cong (5662834, btee)
Code Commit With Part IV (5cdfb387)

1 Introduction

In this lab assignment, a fully-written code of a synthesiser is imperfect due to inefficiently designed tasks execution in an infinite loop. This report will seek to apply the knowledge learnt from CESE4025 Real-Time Systems and to answer the questions listed in the lab manual. The assignment involves the use of the Zephyr Real-Time Operating System (RTOS) and its built-in scheduler to implement a fully functional synthesiser, which will be documented in each section. The topics that our report focuses include characterising the tasks, separating them into threads, handling overload, and using an interrupt-driven peripherals interface. The overall goal of the functional synthesiser is to produce clean sounds with high responsivity.

2 First Encounter with the Synthesiser

As shown below, the team would refer to the assignment instructions for the four main tasks.
Task 1: Check if the state of any peripheral (switch or rotary encoder) has been changed and update the appropriate synthesiser data structures.
Task 2: Check and process the input keys from the keyboard.
Task 3: Synthesise/Generate the sound to be played.
Task 4: Send the generated sound to the audio amplifier.

2.1 Qn 1 - Task 3 and 4

The team checked the hardware and started with the SW1 switch in the upward position with one note playing. The team then used the Logic Analyser to record the Worst Case Execution Time(WCET) and Average Case Execution Time(ACET) as shown in Table 1.

Qn1	**Task 3 (ms)**		**Task 4 (µs)**	
Values	31.911	31.913	2.500	2.500
	31.900	31.895	2.750	2.500
	31.903	31.909	2.500	2.500
	31.908	31.903	2.500	2.500
	31.895	31.900	2.500	2.500
WCET	31.913 ms		2.750 µs	
ACET	31.904 ms		2.525 µs	

Table 1: Tasks 3 & 4 - ACET and WCET

2.2 Qn 1 - Task 2 - Use Cases

The team defined 2 Use Cases that will lead to ACET and WCET. We believed both Use Cases stated below are two extremes that are general and representative of measuring ACET and WCET as shown in Table 2.
Case 1 - Only Key is Pressed
Case 2 - Multiple Keys are Pressed

Task 2	**Case 1 (µs)**		**Case 2 (µs)**	
Values	6.500	1.750	10.250	10.000
	2.000	1.750	1.750	6.250
	1.750	1.750	1.750	9.750
	2.000	1.750	1.750	9.750
	1.750	2.000	1.750	19.750
WCET	6.500 µs		19.750 µs	
ACET	2.300 µs		5.889 µs	

Table 2: Task 2 - ACET and WCET

2.3 Qn 1 - Task 1 - Use Cases

The team defined 2 Use Cases that will lead to ACET and WCET. We believed both Use Cases stated below are two extremes that are general and representative of measuring ACET and WCET as shown in Table 3.
Case 1 - No Peripherals Are Tuned
Case 2 - Frequency Shifter Knob Turned

Task 1	**Case 1 (µs)**		**Case 2 (µs)**	
Values	817.75	817.75	879.75	817.00
	817.75	817.75	879.75	880.75
	817.75	817.75	879.75	880.75
	817.75	817.75	879.75	879.25
	817.75	817.75	879.75	818.25
WCET	817.75 µs		880.750 µs	
ACET	817.75 µs		867.475 µs	

Table 3: Task 1 - ACET and WCET

2.4 % of Processor Utilisation (PU)

In Table 4, the WCET and the respective PU of the tasks are shown. The period for all tasks in the superloop is set to 50 milliseconds.

Period - 50ms	**Time - WCET**	**PU (%)**
Task 1	880.75 µs	1.762
Task 2	19.750 µs	0.040
Task 3	31.913 ms	63.826
Task 4	2.750 µs	0.006
Total	**32.740 ms**	**65.634**

Table 4: Processor Utilisation - Task 1 to 4 (WCET)

The team assigned 64.00% to Task 3 inside the superloop to accommodate Task 3 not to easily lapse into overload situations, while setting a boundary for Task 3's task to have excessive occupation within one full period.

3 Synthesiser Tasks Overhaul

3.1 Qn 2 - Lower and Upper Bound - Chosen Period

The team first explored the possible range available to choose for Task 2, 3, 4. The lowest period that the team considered plausible was 30 ms. Any value lower than 30ms was not ideal as the audio output of the synthesiser would sound distorted. The longest period was 100ms, as any higher value would cause the audio output to be slow and unresponsive.

The team chose 80ms as the period for Task 2, 3, 4 for two main reasons. The first reason is due to having sufficient duration for all four tasks to be feasible to schedule, given that the processor is sufficiently utilised with a low probability of overloading. The second reason is that 80ms is fast enough for quick music beats to be played. This enables high system responsivity, which is a requirement in music genres like Electronic Dance Music(EDM).

Although the team would like to reduce the period to 50ms, the period would be set at 80ms to ensure that minimal issues crop up during the development stages for the nine questions to be completed in the assignment.

The team then explored the possible range available to choose for Task 1. The lowest period that the team considered plausible was 2ms. Any value lower than 2ms was not ideal as the audio output of the synthesiser would sound distorted. The highest period was 10ms as any higher value would cause the audio output to be slow and unresponsive.

The team chose 8ms as the period for Task 1 for it to not overload and yet be responsive.

3.2 Qn 3 - Set Priority for Each Thread

Before determining the priority for each thread, the team read the thread priorities offered by Zephyr RTOS. As shown in Figure 1, Zephyr RTOS provides two options, cooperative threads and preemptible threads.

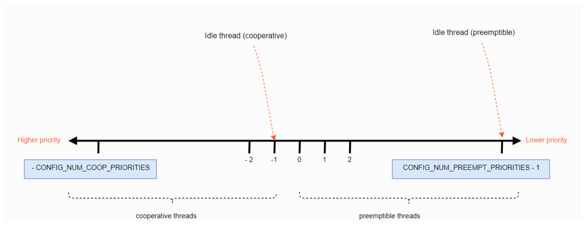
Idle thread (cooperative)
Idle thread (preemptible)
Higher priority
Lower priority
- CONFIG_NUM_COOP_PRIORITIES
- 2
-1
0
1
2
CONFIG_NUM_PREEMPT_PRIORITIES - 1
cooperative threads
preemptible threads

Figure 1: Cooperative Thread vs Preemptible Thread

Since cooperative threads will continue to run even if higher priority threads are ready to run, it is not the team's goal for cooperative threads to ensure a task complete without interruption. In contrast, the team having been taught of preemptive task scheduling chose preemptible threads that can be interrupted by the scheduler to allow another thread of higher priority to run. This process also known as preemption is selected for the music synthesiser to function based on the requirement set in the assignment.

Another rationale for the selection of preemptible threads for priority allocation is also based on the logical process of how the music synthesiser is operating. The case of the rotary switch turning or the key being pressed is an example of user's actions having higher priority in Task 1 and 2 respectively. Assigning higher priorities to the two tasks would increase the responsiveness of the system and thus provide a better user experience. The user inputs captured by Task 1 and 2 determine the sound generated by Task 3 and transmitted by Task 4. This in turn causes Task 3 and 4 to generate the audio signal and to have the signal requested by the user amplified based on the higher priority tasks and to produce output from the synthesiser.

This means if a preemptible thread, i.e. Task 3 and 4 is currently running and a higher priority thread, i.e. Task 1 and 2 becomes ready to run, the scheduler will interrupt the current thread and switch context to the higher priority thread.

Between Task 1 and Task 2, the team has decided to assign the highest priority to Task 1, considering that it has a much shorter period and requires a faster response. As for Task 3 and 4, the lowest priority is assigned to Task 4, due to the data dependency between the two tasks and the fact that allowing the preemption of Task 3 by Task 4 would make the implementation of double buffer more complex.

Zephyr documentation reveals that "the kernel supports a virtually unlimited number of thread priority levels". Therefore, to reduce the complexity of the assignment, four threads have been configured for the four assigned tasks. The allocation of the priorities is shown below.

[1] for Task 1
[2] for Task 2
[3] for Task 3
[4] for Task 4

3.3 Qn 4 - Separate Superloop into Threads

Based on the priority set, the code block was written to implement the preemptible threads for the respective Task 1 to 4. In the following code block, Task 1 to 4 were allocated into the respective threads that were created with their respective names of entry point functions and priority numbering, as shown in bold text.

```
k_tid_t my_tid_0
= k_thread_create(&my_thread_data_0,
                  stack0,
                  K_THREAD_STACK_SIZEOF(stack0),
                  task_update_peripherals,
                  NULL, NULL, NULL,
                  1, 0, K_NO_WAIT);
k_tid_t my_tid_1
= k_thread_create(&my_thread_data_1,
                  stack1,
                  K_THREAD_STACK_SIZEOF(stack1),
                  task_check_keyboard,
                  NULL, NULL, NULL,
                  2, 0, K_NO_WAIT);
```

```
k_tid_t my_tid_2 =
k_thread_create(&my_thread_data_2,
                stack2,
                K_THREAD_STACK_SIZEOF(stack2),
                task_make_audio,
                NULL, NULL, mem_block,
                3, 0, K_NO_WAIT);
k_tid_t my_tid_3 =
k_thread_create(&my_thread_data_3,
                stack3,
                K_THREAD_STACK_SIZEOF(stack3),
                task_write_audio,
                NULL, NULL, mem_block,
                4, 0, K_NO_WAIT);
```

As shown in Table 5, when the peripherals are not turned, the execution time for Task 1 takes longer, as ACET shows 821.100μs as compared to the previous 817.75μs in Question 1. When multiple threads are used, context switching makes the system save the state of one thread before loading another. Thus Task 1 took longer when multiple threads are used.

However, when the frequency shifter knob is turned, the execution time for Task 1 becomes slightly shorter as shown in Table 5, as WCET shows 840.750μs as compared to the previous 880.750μs in Question 1. When threads are created for separate tasks, the team would have the flexibility to fine-tune the period for each task based on the timing constraints and to assign the processor utilisation (PU) accordingly. When implemented in a superloop, every task has the same period of 50ms. After the tasks were separated into threads, the period of Task 1 was set at 8ms, the PU of Task 1 was changed significantly. As shown in Table 6, the PU of Task 1 has risen to around 10.264 percent, while it was 1.762 percent in Question 1. In this way, the task execution can be better organised by the programmer, and it is a significant benefit that outweighed the slight increase in switching time overhead during idle state.

Task 1	**Case 1 (µs)**		**Case 2 (µs)**	
Values	821.25	821.00	819.00	840.00
	821.00	821.00	840.75	820.75
	821.25	821.25	840.75	840.00
	821.00	821.00	840.75	820.75
	821.00	821.25	840.00	820.75
WCET	821.250 µs		840.750 µs	
ACET	821.100 µs		832.350 µs	

Table 5: Task 1 - New ACET and WCET

Qn 4	**WCET Time**	**PU (%)**
Task 1	840.75 µs	10.222
Total	**8 ms**	**100**
Qn 4	**ACET Time**	**PU (%)**
Task 1	821.10 µs	10.264
Total	**8 ms**	**100**

Table 6: Task 1 - Processor Utilisation for WCET

3.4 Qn 5 - Identify Race Conditions

The team reviewed the source code to find race conditions that would potentially occur. For each race condition listed, the team listed the threads that are involved. The team also described the operation to explain how it qualifies as a data race. Consequences have been listed to analyse their severity. Relevant actions were implemented in the code to resolve the intolerable data race condition.

- Race Condition between threads `task_make_audio` and `task_write_audio`

  Type of Operation - Both threads are accessing and updating the shared memory block without any mutual exclusion. `synth.makesynth(uint8_t *) mem_block` in the function `task_make_audio` has audio data being written into the memory block while the `writeBlock(mem_block)` in the function `task_write_audio` sends the data in the memory block to the audio amplifier through Direct Memory Access(DMA).

  Consequences - This race condition might lead to data corruption in the shared memory block, resulting in incorrect sound generation, which the team experienced and described as a trickling sound.

  Action Taken - As instructed in the lab manual, the team implemented a double buffer and mutexes as the synchronisation method. The detailed solution is documented in Question 6 of the lab report. It is ensured that only one of the two threads has access to one block of the double buffer at any given time.

- Data Race on keys array in `task_check_keyboard` thread and `task_make_audio`

  Type of Operation - The keyboard checking thread might update the keys array while the audio synthesis thread is accessing it.

  Consequences - This data race might result in unpredictable behaviour related to key states, hold times, and release times.

```
k_mutex_lock(&mutex_keys, K_FOREVER);
for (int i = 0; i < MAX_KEYS; i++)
    if (key == keys[i].key
        && keys[i].state != IDLE)
        keys[i].state = PRESSED;
        keys[i].hold_time = sys_timepoint_calc
                            (K_MSEC(500));
        keys[i].release_time = sys_timepoint_calc
                               (K_MSEC(500));
        key_pressed = true;


// The second loop is necessary to avoid
// selecting an IDLE key when a
// PRESSED or RELEASED key is located
// further away on the array
    if (!key_pressed)
        for (int i = 0; i < MAX_KEYS; i++)
            if (keys[i].state == IDLE)
                keys[i].key = key;
                keys[i].state = PRESSED;
                keys[i].hold_time =
                    sys_timepoint_calc
                    (K_MSEC(500));
                keys[i].release_time =
                    sys_timepoint_calc
                    (K_MSEC(500));
                keys[i].phase1 = 0;
                keys[i].phase2 = 0;
                break;



    k_mutex_unlock(&mutex_keys);
```

Action Taken - The provided code implements a thread-safe mechanism for handling key state changes within a data structure that represents keys, utilising a mutex to ensure mutual exclusion. When a thread wants to update the state of keys, it first locks the mutex, which prevents other threads from entering the critical section where the shared keys array is being modified. The first loop searches for a key that matches a specified condition (non-IDLE state and same key value) and updates its state to PRESSED, along with setting its hold and release times. If this key is not found, indicating that it was not previously pressed, a second loop searches for an IDLE key to update with the new state and timing information.

Upon completion of the updates, the mutex is unlocked, allowing other threads to access and modify the keys array. This locking and unlocking pattern around the critical section where the keys array is accessed ensures that only one thread can make changes at a time, thus preventing data race.

- Data Race in `Synthesizer::makesynth()` function (synth.cpp) and the `peripherals_update()` function (peripherals.cpp). The two functions are called from the audio synthesising thread and the peripherals checking thread, respectively. The code suggests potential concurrent access to shared resources, specifically the `switches[]` and `encoders[]` arrays, without adequate synchronisation, which could result in race conditions.

  Type of Operation - Reading and updating the shared state of switches[] and encoders[] arrays across different threads, namely `task_update_peripherals` and the `task_make_audio` thread.

  Consequences - If these race conditions are not managed, they may lead to an inconsistent representation of peripheral states, causing unpredictable user interface behaviour for the synthesiser. Moreover, there might be missed updates or incorrect readings of switch states and encoder values.

```
k_mutex_lock(&mutex_peripherals, K_FOREVER);
for (int j = 0; j < MAX_KEYS; j++)

    if (keys[j].state == PRESSED &&
        !sys_timepoint_expired
        (keys[j].hold_time))
        sample += get_sound_sample(keys[j]);
     else if (keys[j].state == PRESSED &&
            sys_timepoint_expired(
                        keys[j].hold_time))
        keys[j].state = IDLE;



// Apply LPF
if (synth._lpf._cutoff_freq > 0.)
    sample = synth._lpf.filter(sample);

k_mutex_unlock(&mutex_peripherals);

if (k_sem_take(
            &sem_peripherals,
            K_FOREVER) == 0)
    set_led(&debug_led0);
    k_mutex_lock(
            &mutex_peripherals,
            K_FOREVER);
    peripherals_update();
    k_mutex_unlock(&mutex_peripherals);
    reset_led(&debug_led0);
```

  Action Taken - As shown in the code blocks above, `void Synthesizer makesynth` function in synth.cpp contains a critical section within `k_mutex_lock(&mutex_peripherals, K_FOREVER);` and unlock portion. The `peripherals_update()` function in the `task_update_peripherals` thread is ensured mutual exclusion by the same mutex. The action taken by the code in synth.cpp with the mutex is to generate an audio sample corresponding to keys that are currently pressed and process it through a low-pass filter if active, all within a thread-safe manner by using a mutex to prevent concurrent access to shared resources.

3.5 Qn 6 - Double Buffering

The team implemented double buffering for the audio buffer first by multiplying the `BLOCK_SIZE` by 2 in the macro `K_MEM_SLAB_DEFINE_STATIC` found in line 12 of the audio.cpp file to obtain twice the amount of contiguous memory space. void `*mem_block = allocBlock();` in line 146 of the main.cpp is a statement that calls the function `allocBlock()` and assigns its return value to a pointer variable named `mem_block`. `allocBlock()` is a function that allocates a memory block from a memory slab and then starts a transmission for an I2S device.

```
atomic_t write_mem0 = ATOMIC_INIT(1);
void task_make_audio(void *p1, void *p2,
                    void *mem_block)
    k_timer_start(&sync_timer_task3,
                K_MSEC(BLOCK_GEN_PERIOD_MS),
                K_MSEC(BLOCK_GEN_PERIOD_MS));
    //bool is_mem0 = true;
    bool is_first = true;
    while (1)
        // Make synth sound
        set_led(&debug_led2);
        if (atomic_get(&write_mem0) == 1)
            if(is_first)
                is_first = false;

            else
                k_mutex_unlock(&mutex_mem1);

            k_mutex_lock(&mutex_mem0,
                        K_FOREVER);
            synth.makesynth((uint8_t *)
                            mem_block);
            atomic_set(&write_mem0, 0);
         else
            k_mutex_unlock(&mutex_mem0);
            k_mutex_lock(&mutex_mem1,
                        K_FOREVER);
            synth.makesynth(((uint8_t *)
                            mem_block)
                        + int (BLOCK_SIZE));
            atomic_set(&write_mem0, 1);

        reset_led(&debug_led2);
        k_timer_status_sync(&sync_timer_task3);
```

The code utilises two kernel mutexes, `mutex_mem0` and `mutex_mem1`, to protect each half of the double buffer. The `atomic_t write_mem0` variable is a flag to indicate which buffer (`mem_block` for the first half and `mem_block + BLOCK_SIZE` for the second half) is currently available for writing by the sound synthesiser.

In the `task_make_audio` function, an atomic check on `write_mem0` determines which buffer to lock and write to. If `write_mem0` is set to 1, it locks `mutex_mem0` and writes to the first half of the buffer using the synthesiser. After writing, it clears the atomic variable to 0 indicating that the first buffer has been filled and is ready for reading. If `write_mem0` is 0, it locks `mutex_mem1`, writes to the second half of the buffer, and sets `write_mem0` back to 1.

```
void task_write_audio(void *p1, void *p2,
                    void *mem_block)
    k_timer_start(&sync_timer_task4,
                K_MSEC(BLOCK_GEN_PERIOD_MS),
                K_MSEC(BLOCK_GEN_PERIOD_MS));
    while (1)
        set_led(&debug_led3);
        if(atomic_get(&write_mem0) == 0)
            k_mutex_lock(&mutex_mem0,
                        K_FOREVER);
            writeBlock(mem_block);
            k_mutex_unlock(&mutex_mem0);
         else
            k_mutex_lock(&mutex_mem1,
                        K_FOREVER);
            writeBlock(((uint8_t *)
                        mem_block)
                        + int(BLOCK_SIZE));
            k_mutex_unlock(&mutex_mem1);

        reset_led(&debug_led3);
        k_timer_status_sync(&sync_timer_task4);
```

The `task_write_audio` function is similar. It checks the state of `write_mem0`. If 0, it knows that `task_make_audio` has filled the first half of the buffer and locked `mutex_mem0` to safely read and send this data to the I2S interface via writeBlock. If `write_mem0` is 1, it locks `mutex_mem1` and reads from the second half of the buffer.

Both tasks use a timer (`sync_timer_task3` and `sync_timer_task4`) to regulate their execution period defined by `BLOCK_GEN_PERIOD_MS`. This helps in synchronising audio generation with data transfer and ensures they operate at a consistent rate.

LEDs (`debug_led2` and `debug_led3`) are set/reset at the beginning and end of each block operation for visual debugging to indicate when audio processing or data transfer is occurring. For example, the `reset_led(&debug_led3)` call would turn off the orange LED found on the STM32F4 Discovery Board, where the orange LED label is marked with an orange box as indicated in Figure 2 shown below.

This design ensures that while one buffer is being written to by the synthesiser, the other buffer is being read by the DMA, thus preventing race conditions. The use of mutexes ensures that access to each half of the buffer is exclusive, preventing simultaneous read/write operations which could result in audio artifacts like clipping or beeps. The result is a clear audio output when playing a single note.

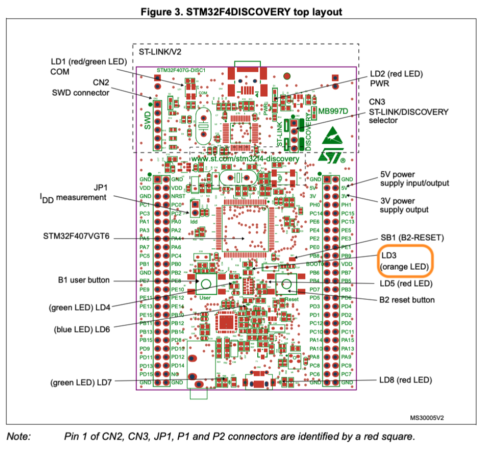

Figure 2: STM32F4 Discovery Board LEDs

4 Overload

4.1 Qn 7 - Handling Methods

```
k_timer_start(&timer_task_overload,
            K_MSEC(BLOCK_GEN_PERIOD_MS - 5),
            K_NO_WAIT);
k_mutex_lock(&mutex_keys, K_FOREVER);
for (int i = 0; i < BLOCK_SIZE; i += 2)
    if (k_timer_status_get(
                &timer_task_overload) > 0)
        k_mutex_unlock(&mutex_keys);
        memset(block + i, 0, BLOCK_SIZE - i);
        set_led(&status_led3);
        return;

    float sample = 0;
...
```

As shown in the code block above, makesynth function in line 485 of synth.cpp, generates audio samples and writes them into a buffer block, called *frame*. The code written `set_led(&status_led3);` function call turns on the red LED which serves as an indicator to the user that the synthesiser is overloaded. Likewise, after successfully filling the buffer without overload, the code `reset_led(&status_led3)` at the end of the function turns off the red LED, indicating that processing has been reset and returned to normal.

As for the code logic to detect cases of overload, the code starts by setting a timer that expires just before the deadline of the current buffer generation period, which is `(BLOCK_GEN_PERIOD_MS - 5)` milliseconds. There is a check for `k_timer_status_get`. If the system time has reached the `k_timer` time, it means that the processor is in danger of overloading and missing the deadline.

When an overload is detected, the code enters an overload handling routine. The program stops synthesising the audio for the current frame, and the buffer is zeroed out, ensuring that any partially filled buffer does not contain garbage values that could produce noise.

5 Interrupts

5.1 Qn 8 - Switches

```
static struct gpio_callback sw0_cb;
static struct gpio_callback sw1_cb;
static struct gpio_callback sw2_cb;
static struct gpio_callback sw3_cb;
static struct gpio_callback sw4_cb;
static struct gpio_callback sw5_cb;
static struct gpio_callback sw6_cb;
static struct gpio_callback sw7_cb;
```

To change the polling update of the state of the 4 switches (8 pins) into interrupt based implementation, the team first created the static struct `gpio_callback variables` for `(sw0_cb to sw7_cb)` in main.cpp as shown in the code block above. They are used to store the callback information for each switch's interrupt.

```
K_SEM_DEFINE(sem_peripherals, 0, 1);
```

As shown in the code block above, `K_SEM_DEFINE(sem_peripherals, 0, 1)` in main.cpp defines a binary semaphore that will be used to offload the work from the interrupt service routine (ISR) to the thread that will process the peripheral updates. The semaphore is initialised to "0", indicating that there's no initial permit for the semaphore. The processing threads wait until a semaphore is given by an interrupt handler.

```
void attach_interrupt_switch(void)
int ret0 = gpio_pin_interrupt_configure_dt
            (&sw_osc_dn, GPIO_INT_EDGE_BOTH);
int ret1 = gpio_pin_interrupt_configure_dt
            (&sw_osc_up, GPIO_INT_EDGE_BOTH);
int ret2 = gpio_pin_interrupt_configure_dt
            (&sw1_dn, GPIO_INT_EDGE_BOTH);
int ret3 = gpio_pin_interrupt_configure_dt
            (&sw1_up, GPIO_INT_EDGE_BOTH);
int ret4 = gpio_pin_interrupt_configure_dt
            (&sw2_dn, GPIO_INT_EDGE_BOTH);
int ret5 = gpio_pin_interrupt_configure_dt
            (&sw2_up, GPIO_INT_EDGE_BOTH);
int ret6 = gpio_pin_interrupt_configure_dt
            (&sw3_dn, GPIO_INT_EDGE_BOTH);
int ret7 = gpio_pin_interrupt_configure_dt
            (&sw3_up, GPIO_INT_EDGE_BOTH);
```

As shown in the code block above, the `attach_interrupt_switch` function configures GPIO pins to generate interrupts on both rising and falling edges `(GPIO_INT_EDGE_BOTH)` for each switch. This ensures that any change in the switch state (on-to-off or off-to-on) will trigger an interrupt. The `gpio_pin_interrupt_configure_dt` function calls return codes (ret0 to ret7), which should be checked to ensure the configuration was successful.

```
void switch_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_peripherals);
    return;

void switch0_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_sw0);
    return;

void switch1_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_sw1);
    return;
```

```
void switch2_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_sw2);
    return;

void switch3_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_sw3);
    return;
```

As shown in the code block above, the interrupt service routines (ISRs) for the switches, such as `switch0_pressed`, `switch1_pressed`, `switch2_pressed`, and `switch3_pressed`, are designed to handle the GPIO interrupt signals. When a switch is pressed, the corresponding ISR is called and gives a semaphore using `k_sem_give`. These semaphores (`sem_sw0`, `sem_sw1`, `sem_sw2`, and `sem_sw3`) are likely defined similarly to `sem_peripherals` and are used to signal a processing thread that a switch has been pressed. Return statements terminate ISR functions after the semaphore has been given.

Each ISR is associated with a specific switch, allowing for individual handling and debouncing of each switch's state. The ISRs are minimalistic to ensure they complete quickly, as it is best practice to spend as little time as possible in an ISR context due to their high priority in the system. The design is that after an ISR gives its semaphore, a separate thread is responsible for handling the state change of the switch. This separate thread would wait on the semaphores given by the ISRs (`sem_sw0` to `sem_sw3`) and then execute code to manage the state change of the switches. This design effectively decouples the interrupt handling from the switch state management, allowing for responsive system behaviour while maintaining organised code structure.

```
void ThreePosSwitch::update()
    // Get the new state
    bool temp_up = gpio_pin_get_dt(_up);
    bool temp_dn = gpio_pin_get_dt(_down);
    int num_check = 0;
    // software debouncing
    while(num_check <= 9)
        if(temp_up == gpio_pin_get_dt(_up)
        && temp_dn == gpio_pin_get_dt(_down))
            num_check = num_check + 1;
            k_usleep(20);
         else
            temp_up = gpio_pin_get_dt(_up);
            temp_dn = gpio_pin_get_dt(_down);
            num_check = 0;

    bool up = temp_up;
    bool dn = temp_dn;
...
```

As shown in the code block above, the `ThreePosSwitch::update()` function uses a software debouncing technique to ensure the stability of the switch state. It starts by capturing the initial state of the 'up' and 'down' positions of the switch, and then enters a loop to perform multiple checks while tracking the consistency of the switch state.

If the state remains unchanged across consecutive checks, the function increments a counter and introduces a short delay using `k_usleep(20)` to filter out any transient noise or contact bouncing. If the state changes during this process, the function resets the counter and updates the temporary states before continuing the checks. Once the loop exits, the function stores the final debounced states, ensuring that only stable and consistent switch state changes are acted upon.

5.2 Qn 9 - Rotary Encoders

```
buttons
    compatible = "gpio-keys";
    switch0: button_0
        gpios = <&gpioe 8 GPIO_ACTIVE_HIGH>;
    ;
    switch1: button_1
        gpios = <&gpioe 9 GPIO_ACTIVE_HIGH>;
    ;
    switch2: button_2
        gpios = <&gpioe 10 GPIO_ACTIVE_HIGH>;
    ;
    switch3: button_3
        gpios = <&gpioe 11 GPIO_ACTIVE_HIGH>;
    ;
    switch4: button_4
        gpios = <&gpioe 12 GPIO_ACTIVE_HIGH>;
    ;
    switch5: button_5
        gpios = <&gpioe 13 GPIO_ACTIVE_HIGH>;
    ;
    switch6: button_6
        gpios = <&gpioe 14 GPIO_ACTIVE_HIGH>;
    ;
    switch7: button_7
        gpios = <&gpioe 15 GPIO_ACTIVE_HIGH>;
    ;
    rot_int: button_8
        gpios = <&gpiob 4 GPIO_ACTIVE_HIGH>;
    ;
;
```

As shown in the code block above, the newly declared `button_8` in the app.overlay file defines a button associated with a specific GPIO pin, `<&gpiob 4 GPIO_ACTIVE_HIGH>`. This configuration indicates that the GPIO pin from the "gpiob" controller with pin number 4 is designated to handle interrupts from a rotary encoder or a port expander.

```
static struct gpio_callback rot_cb;
```

To move the code for updating the port expander that controls all the rotary encoders from the peripheral thread into separate interrupt handlers, the team first created the static struct `gpio_callback variables` for `(rot_cb)` in main.cpp as shown in the code block above. They are used to store the callback information for the interrupt generated by the port expander.

```
K_SEM_DEFINE(sem_peripherals, 0, 1);
```

As shown in the code block above, `K_SEM_DEFINE(sem_peripherals, 0, 1)` in main.cpp defines a binary semaphore is the same as the one for the switches.

```
void attach_interrupt_switch(void)
int ret8 = gpio_pin_interrupt_configure_dt
            (&rot_int, GPIO_INT_EDGE_FALLING);
```

As shown in the code block above, the `attach_interrupt_switch` function configures the GPIO pin connected to the port expander to generate interrupts on the falling edge `(GPIO_INT_EDGE_FALLING)`. This configuration indicates that the interrupt should trigger when the signal is shifted from high to low. The `gpio_pin_interrupt_configure_dt` function calls the return code (ret8), which should be checked to ensure that the configuration was successful.

```
void switch_pressed(const struct device *dev,
                    struct gpio_callback *cb,
                    uint32_t pins)

    k_sem_give(&sem_peripherals);
    return;
```

As shown in the code block above, the `switch_pressed` function is the ISR that will be called when the GPIO pin connected to the port expander detects a falling edge. Similar to Question 8, this routine does not process the state change; it simply signals the semaphore with `k_sem_give(&sem_peripherals)`, indicating that a peripheral update is required. When a semaphore is given, the work is offloaded to the thread that is waiting for interrupts to record the new states of the rotary switches.

6 Conclusion

Based on the analysis and implementation detailed in this report, the team aims that the assignment findings has informed the reader with a comprehensive understanding of real-time systems.

The thorough exploration of thread priorities, period selection, task separation, synchronisation mechanisms, and overload handling has provided a key overview into the intricacies of designing a functional music synthesiser with efficient task execution.

The use of Zephyr RTOS thread scheduler has proven to be pivotal in facilitating the prioritisation and scheduling of tasks, ensuring that the synthesiser functions smoothly and responsively. The implementation of double buffering, semaphore usage, and overload handling mechanisms has further enhanced the robustness and reliability of the synthesiser.

Not forgetting, the thorough investigation and identification of potential race conditions with the correct measures taken to address them have contributed significantly to the overall stability and performance of the synthesiser.

In conclusion, this assignment has been an enriching experience that has deepened the team's learning and understanding of real-time systems. The chance to work with embedded system design for a real-world application in music synthesis provides valuable insights and practical skills that are essential for a career in embedded systems.

References

[1] Project, Zephyr. *www.zephyrproject.org/common − multithreading − problems − and − their − fixes/* "Common Multithreading Problems and Their Fixes." Zephyr Project, 6 July 2023 (accessed 2 Jan. 2024)

[2] Project, Zephyr. *https − //docs.zephyrproject.org/latest/kernel /services/threads/index.html#thread − priorities* "Threads — Zephyr Project Documentation." Docs.zephyrproject.org, 5 Apr. 2022 (accessed 2 Jan. 2024)

[3] TU Delft *https : //cese.pages.ewi.tudelft.nl/real − time − systems/docs/Synthesizer_v1_3_schematic.pdf* "Synthesizer Print Schematic," CESE, Aug. 29, 2023. (accessed 2 Jan. 2024).

[4] STMicroelectronics *https : //www.st.com/resource/en/user_manual/ um1472 − discovery − kit − with − stm32f407vg − mcu − stmicroelectronics.pdf* "Discovery Kit with STM32F407VG MCU - User Manual," STMicroelectronics, Oct. 01, 2020. (accessed 2 Jan. 2024).