CESE4025 Real Time Systems

Davide Liuni (5052637), Dario Capitani (5521912)

January 18, 2026

1 Part I: First encounter with the synthesizer

Sampling Frequency and Buffer Size

Like any digital instrument, the synth needs to send a digital signal to the DAC such that it can be sent to the amplifier and then to the analog aux output, as shown in Figure 1. Due to the standard audio sampling frequency of 44.1kHz, a sample needs to be computed and sent every 22.7 μs to the DAC. Because this would be unfeasible and too constrained even for professional studios set ups, industry standard DAWs (Digital Audio Workstations) use buffer sizes that range from 16 samples to 512, which the user can set while recording/mixing depending on the workload, compute resources and latency needs of the job at hand.

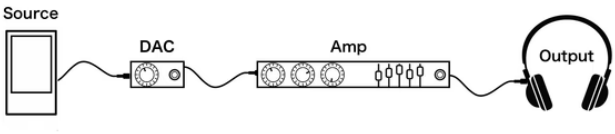

Figure 1: Caption

Block size	Latency	Typical use
16 samples	0.363 ms	Live Performance
64 samples	1.45 ms	Home Studio Recording
128 samples	2.91 ms	Acceptable playback
256 samples	5.81 ms	Mixing under heavy load
2205 samples	50 ms	RTS on a Microcontroller

In the initial implementation, the synthesizer executes Tasks 1–4 (T1-T4) sequentially inside a superloop every 50ms. This buffer size will later be decreased to 30 and then 20ms to reduce the playback latency. Each task executes once per loop iteration and has an effective period equal to the superloop iteration time. This value will be used as the task period for Tasks 1-4 when computing processor utilization when analyzing the initial implementation.

Q1 Execution times and processor utilization

Originally each LED pulse was wrapped around the calls inside the main loop, so T1–T3 included loop overhead and T4, being the last one, measured the "remaining" time of the 50 ms period rather than its own work. By moving each LED toggle inside the function it measures, the pulses were updated to measure only the real task executions, getting T4's execution time to a fixed 2.25 µs instead of any leftover loop time.
Additionally, Task 4 was measured immediately after startup due to the documented increase in execution time over time, and no big discrepancy was measured with later measurements.

Q1 Use Cases

Average Use Case I: 1 Note and 1 Encoder

Average use cases when playing a digital synth would be pressing one or two keys while changing a single encoder. Since, unlike a piano, a synth has all these tunable parameters, such a use case mirrors real life usage.

Task	Idle AECT	1 Note AECT	1 Note WCET	Holding AECT	Holding WCET
T1 (μs)	290	290	290	290	290
T2 (μs)	1.25	1.5	6.00	1.5	9.25
T3 (ms)	1.01	12.51	12.53	12.51	12.56
T4 (μs)	1.75	2.00	2.00	2.00	2.00

Table 1: Measured average and worst-case execution times (ms) for playing 1 note every second or holding the note steady.

For the WCET, the highest values out of all the measurements of these specific use cases were taken. For the AECT, the median over 20 periods was used. This choice was made because execution times deviated only once or twice within a 20-period window; therefore, the median (or the mode in most scenarios) better represents the typical execution time than the mean, which would be disproportionately influenced by rare outliers. A typical timing diagram of the execution times of T1 to T4, starting from top to bottom, is shown in Figure 2, while the execution times in Table 2. Overall, Task 4 was measured to be quite stable and deterministic at around 2μs, as well as task 2, staying in the μs scale except for very rare events where it jumps to 2ms when playing two notes and one encoder. T1 remains at 0.29ms if no encoder is being used and goes to 3.08ms when the user starts rotating any encoder. T3 is the most variable task, and was found to depend on the number of keys pressed. This led to the decision taking different measurements for one or two keys pressed.

Table 2: Execution Times for playing 1 Note and 1 Encoder

Task	T1	T2	T3	T4
AECT (ms)	0.29	0.002	14.3	0.002
WCET (ms)	3.08	0.006	17.2	2.70

When playing two notes, only T3 differed considerably, to AECT = 27.23ms and WCET = 30.2ms, and T2 was found to spike once to 2.86ms.

Figure 2: Average Use Case Execution Times, Logic2 Output

WCET Use Case

The worst case event, where the highest execution times for T1 to T3 took place, was when the maximum number of keys are pressed and one encoder at the same time, which is the maximum number of actions that a person can take with two hands. Because T3 now needs to compute and sum all these notes together, it can take significantly more time, up to 40ms, as shown by Figure 3.

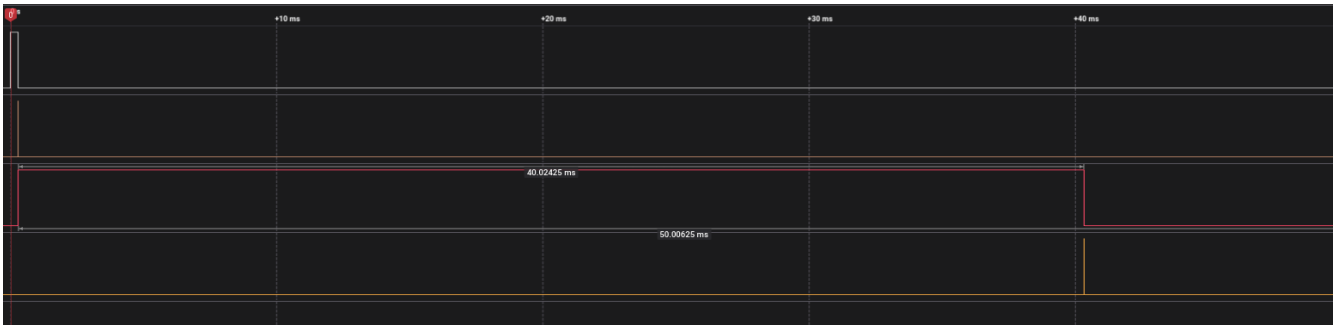

Figure 3: Worst Case WCET

Table 3: Execution Times for the Worst Case

Task	T1	T2	T3	T4
AECT (ms)	0.29	0.002	25	0.002
WCET (ms)	3.08	2.86	40.3	2.70

Processor utilization (PU) in the superloop

Taking 50ms as each task's period, the processor utilization for a task i is computed as:

$$\mathrm{PU}_i = \frac{C_i}{50} \times 100\% = C_i * 2$$

The average execution time was taken from the AECT of the average use case, while the worst case computation times were taken from the WCET of the worst case scenario, both of them written in ms.

Task	C_{avg}	C_{wc}	$\mathrm{PU}_{\mathrm{avg}}$	$\mathrm{PU}_{\mathrm{wc}}$
T1	0.29	3.08	0.58%	6.16%
T2	0.002	2.86	0.004%	5.72%
T3	14.3	40.3	28.6%	80.6%
T4	0.002	2.7	0.004%	5.4%

Table 4: Measured execution times and derived processor utilization in the superloop design.

$$\mathrm{PU}_{\mathrm{tot,avg}} = \sum_{i=1}^{4} \mathrm{PU}_{i,\mathrm{avg}} = 29.19\% \qquad \mathrm{PU}_{\mathrm{tot,wc}} = \sum_{i=1}^{4} \mathrm{PU}_{i,\mathrm{wc}} = 97.88\%$$

In the current implementation, during the event of many input changes, the execution of the tasks would take around 98% of the period's time.

How much PU to assign to Task 3 inside the superloop

Task 3 (sound synthesis) is the least time-critical task because it operates on buffered audio samples. While the DAC is converting and outputting the samples of the current buffer, Task 3 can compute samples for the next buffer with a certain slack. The exact execution instant of Task 3 is, therefore, not critical, as long as the buffer is fully computed before it is consumed by Task 4. In contrast, delays in Task 1 or Task 2 lead to delayed or incorrect input state timing being used during synthesis, causing incorrect note timing or parameter updates, while delays in Task 4 result in buffer underruns and artifacts such as clicks or silence. These tasks therefore have stricter timing constraints.

In a superloop architecture, this implies that Task 3 can be assigned the remaining processor utilization after accounting for the worst-case utilization of Tasks 1, 2, and 4; something that will be taken into account when deciding thread priorities. Using worst-case values, the available processor utilization budget for Task 3 is given by:

$$\mathrm{PU}_{3,\mathrm{budget}} = 100\% - (\mathrm{PU}_{1,\mathrm{wc}} + \mathrm{PU}_{2,\mathrm{wc}} + \mathrm{PU}_{4,\mathrm{wc}}) = 82.72\%$$

Since the processor utilization of Task 3 in the very worst cases was 80.6%, this budget is sufficient.

2 Part II: Synthesizer tasks overhaul

Q2 Period bounds and chosen periods

In the threaded redesign, each task i is assigned a period T_i that must be large enough to avoid self-overload given its worst-case execution time (WCET), and small enough to keep the synthesizer responsive and to satisfy timing constraints, especially the audio streaming deadline. The WCETs used below are taken from the previous measurements (Table 2 and 3).

Task 4 is the consumer of audio frames. Audible artifacts, such as gaps in audio or undefined clicks, happen if there are buffer underruns. Thus, the strictest deadline is that T4 needs to send a block of samples to the DAC exactly when the previous one has finished. Practically, this means it should be run periodically at 50 ms with low jitter. Task 3 must finish producing the next buffer before Task 4 needs it, that is it's upper bound. A conservative bound is to require T_3 to be no smaller than its WCET, otherwise it cannot finish even in isolation. Thus we select a period aligned with the frame, producing exactly one buffer per audio frame. Making it any faster would produce more blocks during one period, while any slower it would not provide the block in time for T4 to consume it.

With many simultaneous keys, we measured $C_{2,\text{wc}} = 2.86\,\text{ms}$.
In this design the dominant latency is already the audio frame size (50 ms), so it is undesirable for input polling to be slower than one frame; otherwise key-to-sound latency can exceed one frame. This keeps input latency well below the audio frame time while avoiding scheduling the task faster than its measured worst case.
It wasn't part of the worst case scenario, but the maximum measured execution time for scanning 2 encoders at the same time was $C_{1,\text{wc}} = 6.1\,\text{ms}$.
The encoders have 96 slits per turn. To avoid missing fast user interactions, polling should be fast enough to observe multiple slit transitions during a quick turn. Human knob rotation can easily reach a few turns per second; thus, millisecond-to-tens-of-milliseconds polling is reasonable for a responsive feel. Again, Task 1 should not be scheduled faster than it can complete under worst-case scanning, keeping UI latency low while remaining above the measured WCET.

Selected periods

Table 5: Chosen periods with justified bounds (based on measured WCETs and audio/frame constraints).

Task	T_{lower},[ms]	T_{higher},[ms]	**Chosen** T,[ms]
T1 (peripherals)	6.1	20	10
T2 (keyboard)	2.8	50	50
T3 (synthesis)	41	50	50
T4 (audio out)	50	50	50

Q3 - Thread Priorities

The design choices for the threads were made on the basis of input, computation and output; therefore, three threads were chosen, one for task 1 and 2, followed by two threads for task 3 and 4. This does create a discrepancy between the periods of tasks 1 and 2 chosen in table 5, however, the idea was to try and reduce the period of a block itself, to make the whole system more reactive and reduce the latency between a key pressed and a note heard. Task 4 is assigned the highest priority, due to the buffer underrun artifacts possibility. The combination of task 1 and 2 have the second priority as any delay is immediately hearable, while, as explained previously, task 3 has a certain slack since finishing the block earlier than it's deadline provides no benefit, making it the right choice for the lowest priority. Task 4 transmits the blocks which were produced from task 3, in particular task 4 calls an `i2s_write()`, which blocks the thread during transmission allowing for other threads to operate. Ideally, during this time, task 3 should then take control and produce data for the next transmission. With the remaining thread serving the ring buffer for keyboard presses and state changes for the rotary.

Main Thread

The main() function in Zephyr executes within its own system-generated thread, and is assigned its own dedicated stack memory. In our design, the role of the main thread is strictly that of a system initializer and supervisor. Its functionality is to perform all hardware and software initialization, such as configuring USB, initializing the audio driver, peripherals, and instantiating the synthesizer core, before spawning the dedicated worker threads (Task

1/2, Task 3, and Task 4). After successfully creating these threads, the main thread transitions into a dormant state by calling k_thread_join() on the worker threads. While this effectively halts the main thread, it is a critical design consideration for resource-constrained embedded systems. Typically, a dormant thread continues to occupy its stack memory (often 1KB or more), which is wasted RAM. An alternative, more memory-efficient design would be to allow the main thread to exit (return), thereby freeing its resources, or to repurpose the main thread itself to execute one of the worker loops (such as Task 1/2), thus reclaiming its stack space for productive work. However, keeping it as a separate supervisor allows for future expansion where it could handle system-wide error recovery or mode switching.

Q4 - Separate the superloop into threads

You can find the threaded implementation of the tasks inside the main.c file. Three functions were added `task_1_2_thread_entry`, `task_3_thread_entry` and `task_4_thread_entry`. With regards to improving, the timing for task 4, we decided to go with changing the parameter of ticks per second. This was done by changing prj.conf, `CONFIG_SYS_CLOCK_TICKS_PER_SEC` and setting it to 25,000 compared to the inital 10,000. Doing so improved the resolution of each tick happening from every 0.1ms to every 40μs

Average Use Case

When comparing the execution times from the initial superloop implementation to the threads, the same use cases and methology was followed as in Section 1, with table 6 being a copy for reference. The execution times for T1,T2 and T4 remained comparable, with a slight increase in T1's execution when no encoders were being moved. The only major difference was for task 3, which for the average use case decreased. It's important to note that the block period was decreased to 30ms for these measurements.

Table 6: Execution Times for playing 1 Note and 1 Encoder

Task	T1	T2	T3	T4
AECT (ms)	0.29	0.002	14.3	0.002
WCET (ms)	3.08	0.006	17.2	2.70

Table 7: Execution Times for playing 1 Note and 1 Encoder with Threads

Task	T1	T2	T3	T4
AECT (ms)	0.30	0.002	4	0.002
WCET (ms)	3.08	0.006	9.7	2.70

Worst Case

Below is the table for the new Worst case execution and for the new threaded design, the only difference to Section 1 is that both the AECT and WCET of T3 are 26.7ms. In this implementation, Task 4's thread is blocked when using i2s_write() during data transmission. This allows the RTOS to immediately yield the CPU to Task 3 to begin producing the next audio block, whereas a superloop might waste cycles polling or waiting for the next iteration. Another reason for the improvement was also the removal of print statements.

Table 8: Execution Times for the Worst Case

Task	T1	T2	T3	T4
AECT (ms)	0.29	0.002	26.7	0.002
WCET (ms)	3.08	2.86	26.7	2.70

Q5 - Shared Resources

Keys List - Data Race between Task 2 and Task 3

Status: Tolerable, as it only occurs during Overload.
Threads Involved: Task 1/2 (2nd Priority, Producer) and Task 3 (3rd Priority, Consumer).
Mechanism & Trigger: A race condition exists on the global keys array, but it is effectively masked during normal operation by the scheduling and priority design. Both tasks share the same period. Since Task 1/2 has higher priority, it is guaranteed to execute and complete its update of the keys array at the start of every period before Task 3 is scheduled. However, if the system becomes overloaded ($T_{3,exec} > T\_frame$), Task 3 will still be processing audio when the next period begins. At that moment, Task 1/2 will wake up and preempt Task 3 in the middle of its synthesis loop to process new USB inputs. This can cause Task 3 to resume execution with inconsistent key states. We chose not to lock this resource because the race condition is mutually exclusive with correct system behavior; it simply cannot happen unless the deadline has already been missed. Since the system is already in a failure state (audio stutter/glitch due to missing the deadline), a potential minor artifact from a key-state race is inconsequential compared to the primary failure. Adding a mutex would only increase overhead in the critical path without solving the root cause (CPU overload).

Encoder Update - Data Race between Task 1 and Task 3

Status: Tolerable, as it only occurs during Overload.
Threads Involved: Thread Task 1/2 and Thread Task 3.
Mechanism & Trigger: This data race has the exact same mechanism as the previous, following the same reasoning, it was chosen not to fix it.

Audio Buffer Memory - Data Race between Task 3, Task 4, and DMA

Status: Fixed, via Double Buffering and Message Passing.
Threads Involved: Thread Task 3, Thread Task 4
Mechanism Trigger: A race condition exists if the Task 3 writes the generated audio to a memory buffer while the I2S DMA controller is still reading from it to send data to the DAC. This is a classic 'Reader-Writer' problem. If Task 3 modifies the buffer currently being transmitted, the output audio will contain a mix of old and new data, resulting in severe distortion or artifacts. Additionally, if Task 4 preempts Task 3 to send an incomplete buffer, the output will contain undefined data (silence or random samples).
Fix Design Performance Consequence:
This was fixed by implementing Double Buffering, also explained in the next section, combined with Message Queues. Zephyr k_msgq are used to pass the pointer to the audio buffer from Task 3 to Task 4. Task 3 generates the full audio frame in a local buffer. Only after completion does it push the pointer to the queue. Task 4 blocks on this queue; it can only access the buffer once Task 3 has explicitly relinquished control. This prevents Task 4 from sending incomplete data, regardless of thread preemption. k_mem_slab are used to manage multiple audio buffers. When Task 3 begins a new cycle, it allocates a fresh, distinct memory block. Meanwhile, the previous block (passed to Task 4) is held by the I2S driver for DMA transmission. Because Task 3 writes to Buffer A while DMA reads from Buffer B, no simultaneous access occurs. The buffer is freed back to the slab only after the DMA transfer completes.

Double Buffer

Inside `audio.cpp` a memory slab is defined using the `K_MEM_SLAB_DEFINE` allocating two blocks in the slab. The process of populating the slab occurs in task 3, via three steps. Task 3 first needs a block, which it can acquire via the `allocBlock` function and will timeout on the duration of the `BLOCK_GEN_PERIOD_MS`. If this fails, then we set the block to a null pointer and a boolean `i2s_started` to false, in which case the current is skipped and we continue to the next iteration. The second step is via `makesynth`, the block is populated. To then transmit the block to task 4, a message queue is used, with a timeout of forever. Task 4, will wait for the message queue to be filled via the `k_msgq_get` with a timeout of forever, which allows the other tasks to occur. When a block is received, the function `writeBlock` is called, which will start the i2s driver correctly if the `i2s_started` bool is false, transitioning via it's statemachine into the START state. In case writing the block fails, block is freed using the `k_mem_slab_free` function. With regards to setting up the i2s correctly, we first move the driver from ERROR to READY state with `I2S_TRIGGER_PREPARE`, then we queue two blocks with `i2s_write` so the TX queue is primed, and finally I transition READY to RUNNING with `I2S_TRIGGER_START`. If any step fails, the `I2S_TRIGGER_DROP` is

called to flush all the blocks and transition the driver back to READY and mark `i2s_started` as false, otherwise to true.

3 Part III: Overload

As mentioned previously, task 1/2 and task 3 share the same deadline, being the current timepoint when they enter the start of the loop plus the duration of a block. Due to their priorities task 3 executes after, which gives it a worst case computation time of block period minus the computation time of task 1/2. The deadline is passed to the function makesynth, which during the for loop will check every `check_interval` if the remaining time is less than `guard_ticks`. In which case the boolean `overloaded` is set to true which is returned, breaks and set the remaining block to zero's. The boolean value that `makesynth` returns -`overloaded`- is used to indicates overload on the following frame; if allocation fails, it also sets `overload_next` and skips synthesis. Block allocation will fail when overload occurs as the memory slabs are being filled too fast, resulting in the time out when trying to allocate a block from failing. As shown in Fig.4 which shows a domino effect of how overloading increases the total computation time of task 3.

Figure 4: The red line -task 3- where each high state is the duration of a memory block allocation during overload. This is being ran with a block period generation of 20 ms

4 Part IV: Interrupts

The code for all the previous parts is present in the main branch. From this point onwards, for both question 8 and the bonus, the relevant code is in the 'bonus' branch with hash 66417dc8b3c0a2a4ebfd299c5bbbcf548a67077c. To add interrupt for switches, we started of by changing the initialization part for each switch. Inside `init_peripherals` when `initialize` is called on the switches, we pass a function address called `interrupt_handler`. The interrupt handler, will implement debouncing using the `k_work_reschedule` key word, which will wait 5 ms before forwarding to the actual function callback, `debounce_handler`. This function will create a structure, with a pointer to the switch, the new states and will push the struct into a message queue. The thread running task 1/2 will then run `peripherals_update`, the thread drains that queue and calls update(up, down), which computes the new 3-position state for that switch.

Bonus

To implement an interrupt for the encoder, we followed the same approach as for the switch. To begin with, the motherboard is connected to the interrupt of the I2C chip via the pin `PB4`, ergo we had to add this information to the `app.overlay`. This line is added `gpios = <&gpiob 4 GPIO_ACTIVE_LOW>` with the label `expander_irq` and the alias `expander-int = expander_irq;`. Doing so will declare that pin as a GPIO pin, which allows us to then define an interrupter as per the previous question. The pin is configured to be active low as per the documentation and the function `expander_isr` inside `peripherals.cpp` is set to be it's callback function. As the interrupt only

occurs when a change of state happens to an encoder, the flag `enc_irq` is set by the callback function only. The thread for task 1/2 will then check if the pin is true, set it back to false and then update the encoder states. A short debounce period of 5 ticks is added, comparing the time between interrupts and ensuring that it is higher than 5 ticks (1 tick = 40μs), which correspond to 200μs.

One thing that we noticed is that the number of missed deadlines due to overloading, reduced when interrupts were used. Which were due to the reduction of computation time for serving the users input.

References

[1] R. Nimkar, "Switch interrupt in zephyr rtos. a comprehensive guide to configuring and handling interrupts in zephyr rtos." Aug 2022. [Online]. Available: https://medium.com/@csrohit/switch-interrupt-in-zephyr-rtos-bbdaa356207c

[2] "User manual discovery kit with stm32f407vg mcu," May 2025. [Online]. Available: https://www.st.com/resource/en/user$_manual/um1472 - discovery - kit - with - stm32f407vg - mcu - stmicroelectronics.pdf$