Embedded Systems CW2: Group Embreadded Symptoms

Authors:

Rishil Patel (CID: 01560681)

Victor Florea (CID: 01514418)

Aaman Rebello (CID: 01488753)

Anushka Kulkarni (CID: 01567227)

Date: March 26, 2021

1 Introduction

In the given task, we were required to program the real-time behaviour of a micro-controller (ST NUCLEO-L432KC) that operates a modular music synthesiser. The operating system we use in this microcontroller is FreeRTOS. This real-time behaviour would be externally observable to users through the inputs of the synth e.g. keys, joystick, and its outputs e.g. OLED display, sound. In addition, we needed to analyse this behaviour on a software level from the perspective of real-time operating systems.

The behavioural requirements of our firmware are enumerated in the coursework specification. They include external features desirable in a useful synthesiser e.g. no delay between note-press and sound. There are also provisions for communication with other similar synth modules via messages. Also included are constraints on how the firmware should be implemented e.g. use of threads/interrupts.

Therefore, our central jobs in this coursework were the following:

- Characterise the required behaviours of the synth (both initial and advanced) into tasks that the micro-controller would implement concurrently.
- Reason about the initiation intervals and importance of different tasks and, based on this, decide whether to implement each one as an interrupt routine or thread. Decide the relative priorities of the threads.
- Identify necessary variables/data structures - local and global - that the tasks would need to operate with. For global data, ensure that situations like race conditions, deadlock etc do not arise from concurrent access by tasks. There are suitable measures to do this e.g. atomic operations, mutexes etc.
- Ensure that code is efficient, especially in interrupt routines, so that deadline constraints are met.
- Perform an analysis of our software as described in the coursework specification. This mainly pertains to how quantitatively the successful inter-operation of tasks is achieved Results may be found in the part ¡insert¿ of this report.

The lab tasks 1 and 2 were an initial starting point that helped us achieve the above objectives.

2 A Summary of Our Implementation

Our synth module conforms to all functional requirements in the coursework specification. Efforts have been made to conform to the non-functional requirements as much as possible. A video of these features in operation may be found here `https://youtu.be/3AHNhz73ujM`.

The following advanced features have been implemented on top of the firmware at the end of lab 2:

- In addition to sawtooth; *sine*, *square*, *triangular* and *exponential waveforms* have been implemented; these can be configured by adjusting the knob 2 (see figure 1 in the next section). The first four waveforms are standard in many synthesisers. The selected waveform is indicated on the OLED display.
- *Polyphony* is implemented for sawtooth and triangle waveforms. It was not conducive for us with the other waveforms. When multiple notes are played, all of them display on the OLED.
- A *smoothing FIR filter* that averages the last four samples is optionally configurable by adjusting knob 1. It makes the waveform sound more smooth. The presence or absence of this smoothing effect is indicated on the OLED.
- *Tremolo* (low frequency amplitude modulation) is implemented for all waveforms. It is optionally configurable via knob 0.
- *Vibrato* (low frequency frequency modulation) is implemented for sawtooth and triangle waveforms. If available, it can be configured by adjusting knob 0.
- *Echo* is implemented for sine wave only. If available, it can be configured by adjusting knob 0. Presence of any of tremolo/vibrato/echo, as configurable by knob 0 can be seen on the OLED display.
- The effects of knob 2 (waveform), knob 1 (smoothing/no smoothing), knob 0 (tremolo/vibrato/echo) and polyphony can all be superimposed on each other if they are available.
- A preprocessor directive NO_SAMPLE_ISR has been added to the code for easy switch between normal and test mode. If the directive has been defined, the code will be in test mode.

Internal implementation of tasks, resolution of dependency issues and mechanisms to ensure meeting of deadlines are all discussed in later sections of this report.

3 Default configuration for testing Core Functionality

In order to test the core functionality of the synth, please use the settings shown in the image below.

- Set the desired volume using knob 3
- Set the waveform to ”Saw”, denoting a sawtooth wave using knob 2
- Set the effect to ”None - None” using knob 1 and knob 0

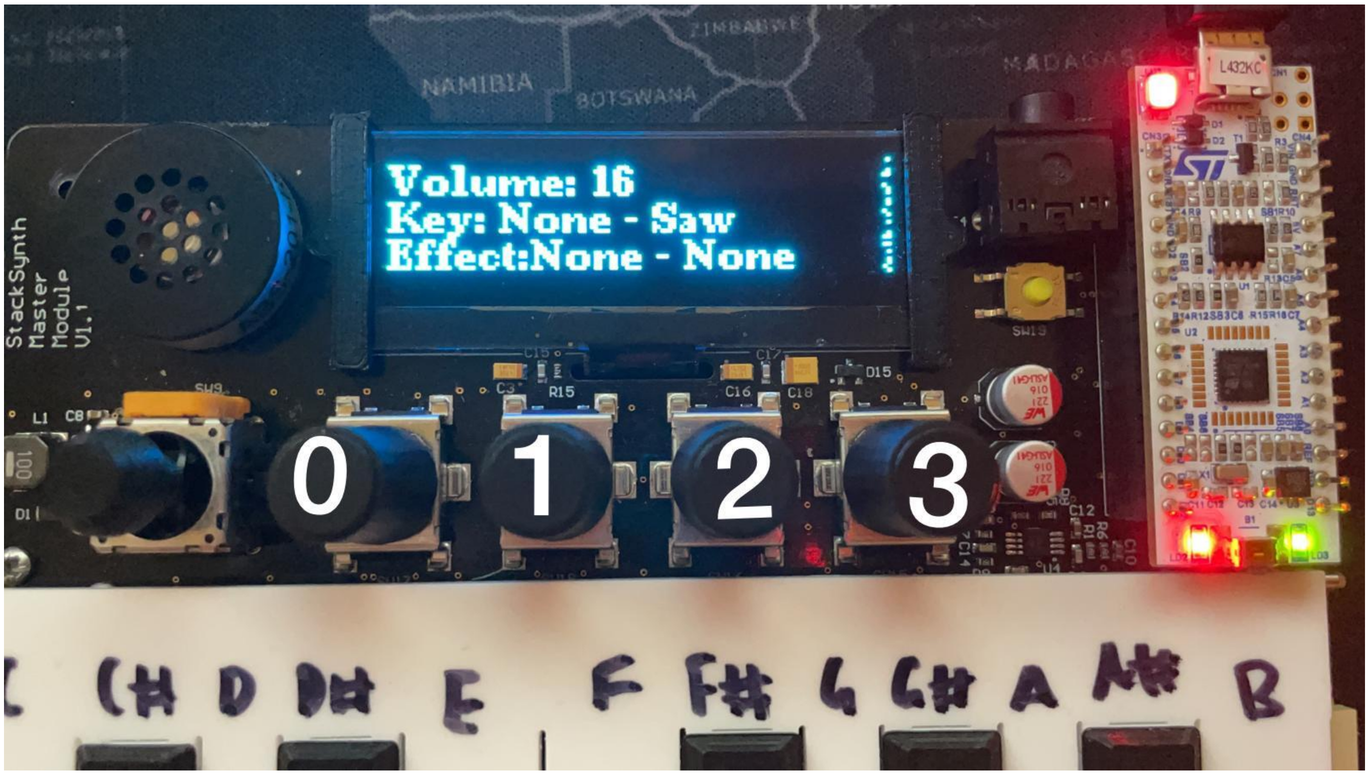
Volume: 16
Key: None - Saw
Effect:None - None

Figure 1: Keyboard with keys and knobs labelled for reference. Display shows settings for core functionality.

4 Tasks

This section describes the tasks performed by the system, in the order of their priority (highest to lowest).

- Updating Output Waveform Phase (implemented by SampleISR)
- MsgIn Task
- ScanKeys Task
- MsgOut Task
- DisplayUpdate Task

	Maximum Execution Time / µs	Minimum Initiation Interval / ms
SampleISR	17	0.045
MsgIn	35	5
ScanKeys	54	20
MsgOut	120	5.6
DisplayUpdate	16777	100

Table 1: Comparison of the Maximum Execution Time, and the Minimum Initiation Interval for tasks

4.1 Updating Output Waveform Phase

This task has the highest priority. There is a need to update the waveform at exactly regular intervals to guarantee a fixed frequency and stable waveform. Since the sampling frequency is 22000 Hz, the initiation interval would be $1/22000 = 4.55 \times 10^{-5}s$ while a note is being played. This is the smallest interval among all tasks. The task is implemented with an interrupt routine: void sampleISR().

SampleISR does not have any arguments or return value. It updates the phase accumulator and sets the analogue output voltage at each sample interval. It adds current step size to the phase accumulator to generate the output waveform.

This function also includes a switch statement to generate different kinds of output waveforms: sine, triangle, square, exponential, and sawtooth (default).

SampleISR also handles operations of other knobs such as volume control, different effects (tremolo, vibrato, etc. (see 2)).

It is triggered by an interrupt 22,000 times per second. This is done by attaching a timer to the function in the Setup code. Unlike other tasks, this does not use mutexes since it is an interrupt.

4.2 MsgIn Task

This task has second highest priority. There is a need to play the note as soon as the external message arrives. This would be useful in a situation where two or more modules are connected to each other in an extended keyboard. One of the modules would receive messages from the others to play notes at different octaves. A message would have be processed potentially every time a key is pressed - if multiple keys are pressed together, there would be more than on message.

A MsgInTask thread is created in the setup function. This task reads the message from the serial monitor and checks if the incoming message (of the form ”XYZ”) has its first character(X) equal to ”R” release or ”P” play. If it is a release, it uses an atomic store to set the stepsize to 0, so that it stops playing the key. If it is a play ”P” message, then it uses an atomic store to set the stepsize based on the frequency of the key that is pressed. It gets this information from the remaining two characters of the message, where the first(Y) is the octave, and the second(Z) is the number (from 1 to 12) of the key pressed.

The initiation interval for this task is 5ms, which is calculated using the channel bit rate and size of the character buffer to ensure all the messages received are decoded by this task (see 4.3).

4.3 ScanKeys Task

This task has the third highest priority. It has the functions listed below:

1. Scans knob rotations (for the functionalities described in the previous sections), and updates the knob status accordingly.
2. Scans key presses and updates the current step size accordingly. It allows for a maximum of 8 simultaneous key presses.
3. Based on the keypresses, it computes the message to be sent to the msgOutQueue. For eg., if key A is pressed by the user, and the octave set is 4, it will send the message ”P4A” correctly.

As specified in the functional requirements, performing the above functionalities must not create a perceptible lag between a user pressing a key/turning a knob and the corresponding observable effect e.g. the sound/update of OLED display. However, due to the nature of human perception, this does not place high demands on the overall execution time (it should probably be below 0.01s which is achievable). The task is implemented as a thread named ScanKeys Task.

For points 1 and 2, the thread uses mutexes to access the KeyArray. For eg. it is locked at the start of scanning knob rotations, and unlocked after it has been completed to ensure safe access, as keyArray is a global variable that can be modified by more than one functions in the program. It is also used by DisplayUpdateTask (see 4.5). Refer to 5 for more details on how variables are accessed in different functions.

For point 3, a queue is used as a buffer so that execution isn't blocked if the channel is busy when the function needs to send a message.

A condition has been added to interrupt an incoming message from serial monitor so that precedence can be given to the physical key presses that are read by the ScanKeys. This means if a key is pressed at the same time that a message is sent through the serial monitor, the serial monitor message will automatically be set to ”R” (release) in order to prioritise the former.

The initiation interval of ScanKeys is set to 20ms to be able to detect rapid knob rotations. Since there is a maximum of 8 key presses in an instant, the worst case for ScanKeys task is to complete 8 iterations in 20ms. This interval is compatible with that of MsgIn as described above, so every message sent by this task is decoded by MsgIn task.

4.4 MsgOut Task

This task has the lowest priority. and is implemented via a thread. This thread takes in the messages from the queue and sends them over to the serial port. When it receives new data it is sent to the port and appears on the serial monitor.

Only key presses (”P”) (physical and through the serial monitor) are added to the queue, so the release ”R” messages are never displayed. Due to this, the queue doesn’t ever have to contain more than 8 items (8 is the maximum key presses allowed in an instant).

4.5 Display Update Task

Like the name suggests, the display update task updates the display. This task has the lowest priority - it must update the display every time a key is pressed or a knob is turned.

It reads the global array pressed[] to find out which keys have been pressed, and accesses the global array keyArray to display the corresponding note on the screen. All global variables accessed in this function are protected by mutexes to ensure safe access (as mentioned in 4.3).

The initiation interval for this task is 100ms so that all the other tasks (particularly MsgIn and ScanKeys, which affect the output) can be completed before the display is updated. This is also why this task has the lowest priority.

4.6 CPU Utilisation

The total CPU utilisation is the sum of each task utilisation, and is calculated using the following equation:

$$U = \sum_i \frac{T_i}{\tau_i} \tag{1}$$

Where T is the minimum initiation interval, and τ is the maximum execution time. The utilisation of each task is shown in Table 2. From these, the total CPU utilisation is calculated to be 57.2%.

Task Name	Utilisation
SampleISR	37.7%
MsgIn	0.7%
ScanKeys	0.3%
MsgOut	2.1%
DisplayUpdate	16.8%

Table 2: CPU Utilisation Percentage for each Task

4.7 Critical Instant Analysis

The lowest priority task for the synth software is the Display Update, which runs once every 100ms, and takes 16.8ms to execute. The Table 3 shows that all the tasks are executed with the interval of the longest task with time to space, as at most it would take 57.8 ms, while the longest interval is 100ms.

	τ_i / µs	T_i / ms	$\frac{\tau_n}{\tau_i}$	$\frac{\tau_n}{\tau_i} T_i$ / ms	$\frac{T_i}{\tau_i}$
SampleISR	17	0.045	2223	37.8	37.8%
MsgIn	35	5	20	0.7	0.8 %
ScanKeys	54	20	5	0.3	0.3 %
MsgOut	120	5.6	18	2.2	2.2 %
DisplayUpdate	16777	100	1	16.8	16.8 %
			Total	57.8	57.8 %

Table 3: Critical Instant Analysis of the system. $\tau_n = 16777$, execution time of DisplayUpdate.

In spite of containing many lines of code, the SampleISR routine executes fast enough to not adversely affect the meeting of deadlines of other tasks.

5 Access, synchronisation and dependencies

Below is a table of all shared data structures in our code used by different tasks and the way in which they are protected to guarantee safe access and synchronisation.

Name	Type	Task	Protection	Read/Write
currentStepSize	volatile uint32_t	sampleISR()	Interrupt	Read
		msgInTask()	Atomic	Write
		scanKeysTask()	Atomic	Write
currentKey	volatile int	sampleISR()	Interrupt	Read
		msgInTask()	Atomic	Both
		scanKeysTask()	Mutex	Both
		displayUpdateTask()	Mutex	Read
pressed	volatile int array	sampleISR()	Interrupt	Read
		scanKeysTask()	Mutex	Write
		displayUpdateTask()	Mutex	Read
incoming	volatile char array	sampleISR()	Interrupt	Read
		msgInTask()	Atomic	Write
		scanKeysTask()	Mutex	Write
		scanKeysTask()	Atomic	Read
globalShift	volatile int	msgInTask()	Atomic	Both
		scanKeysTask()	Mutex	Write
		scanKeysTask()	Atomic	Read
XATOCODE	volatile int	sampleISR()	Interrupt	Both
		displayUpdateTask()	Atomic	Read
WAVECODE	volatile int	sampleISR()	Interrupt	Both
		displayUpdateTask()	Atomic	Read
FIR	volatile int	sampleISR()	Interrupt	Both
		displayUpdateTask()	Atomic	Read
knob 3,2,1	Knob	sampleISR()	Interrupt	Read
		scanKeysTask()	Mutex	Write
		displayUpdateTask()	Mutex	Read

Table 4: Table showing shared data structures, where and how they are used (Read/Write)

We can see that all inter-task operations on shared data are always protected by either an atomic operation, a mutex or being an interrupt function. For example in the table above we can see that even though multiple tasks use currentStepSize, msgInTask() and scanKeysTask() both use atomic operations. This means that the operations in currentStepSize will either fully complete, or not complete at all. Even if there was an interruption from SampleISR, it would not affect the consistency of the variable.

SampleISR is an interrupt and therefore it cannot be interrupted by anything else, so the changes and accesses will always be synchronised.

In the entire program, there is only one mutex instantiated - more are not required. This eliminates the possibility of a deadlock as there is no circular wait-graph. The mutex prevents race conditions in concurrent access to the pressed array by scanKeysTask() and displayUpdateTask(). It prevents displayUpdateTask() from reading data that is partially modified by scanKeysTask(). Concurrent accesses to the knob objects and currentKey are similarly protected by mutexes, although the latter could also be protected by atomic writes and reads. Atomic writes and reads are used to protect accesses to all other variables, which are of the C++ int type.

The mutex creates a blocking dependency between scanKeysTask() and displayUpdateTask(). The latter cannot read pressed until the former has finished writing. Similarly, the former must wait for the latter to finish reading before writing. In the second case, displayUpdateTask() would have a priority at the level of scanKeysTask() due to priority inheritance that is implemented in FreeRTOS.

Since our global constants are read-only they'll never be written, only read so synchronisation doesn't need to be taken into account, and accesses will always be available.