Design Brief: Group 5

Gabriel Apap, Damjan Filipovic, Mark Mizzi

March 10, 2023, Document v.6665

Write an executive summary for your project. This should not overflow into the next page, should not contain references, and should be readable by a wide audience.

1 Introduction

Write an introduction, explaining the purpose and background of this project. Give a brief description of the game to be implemented. Also include an overview of this document.

2 System Design

2.1 System overview

This section offers a brief overview of the designed system's features.

The system will make use of several I/O devices, including

1. An LCD used to convey user input or settings options to the user.
2. An indicator LED which turns on when the system is booted.
3. A keypad input device used to choose between settings or to input DTMF symbols.

On boot-up, the system will prompt the user to enter one of two operational modes: settings or normal mode.

In settings mode, the user is presented with numbered options which can be used to configure the system. The user's options are persisted to flash memory for use in normal mode. The options available to the user will include:

1. Inter-symbol spacing, i.e. the delay between two consecutive tones being produced.
2. Symbol length, i.e. the duration for which a single tone is produced.
3. Tone resolution. This setting affects the sampling frequency of the tone produced. A higher tone resolution produces a better quality tone, but is more computationally intensive.
4. Reboot system. This saves the chosen options and resets the system.

When the system enters normal mode, options are loaded from flash memory, and the system behaves as a DTMF encoder, producing tones when the user presses keys.

2.2 System architecture

To implement these features while adhering to good seperation of concerns, the system must make use of several software components. The proposed software components are shown in Figure 1.

Not shown in this figure is the use of several global variables to store the state of the system. These include:

1. A global, thread-safe flag which indicates whether a tone is being generated.
2. A global counter used to keep track of tone generation in between interrupt handler invocations (see below).

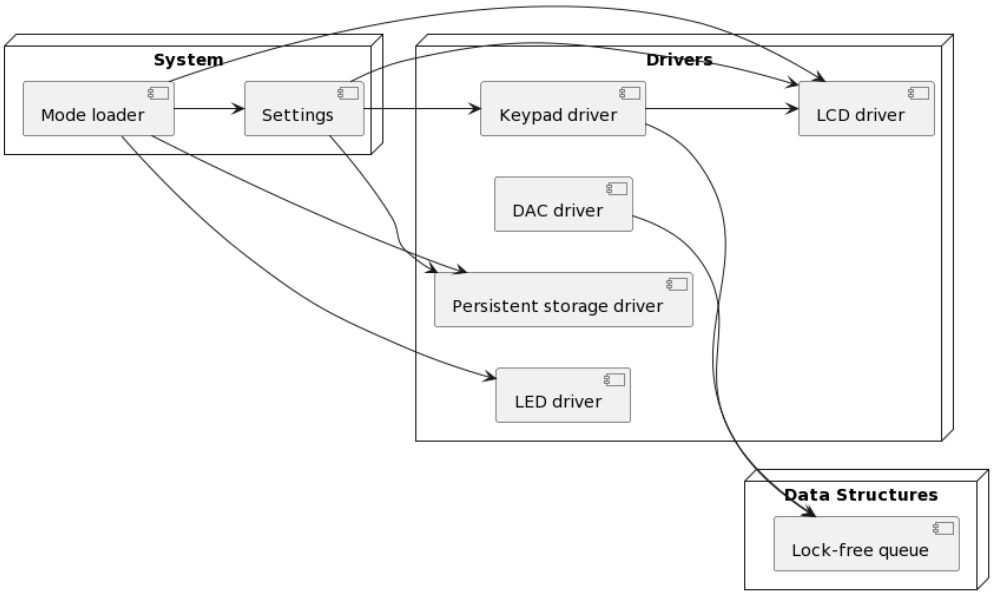

Figure 1: Software components proposed for the system, grouped by functionality. Arrows between components indicate caller-callee dependencies.

3. A global C `struct` holding user configuration of the system. In normal mode, this is loaded from flash memory, and should be read-only. In settings mode, the configuration is loaded from flash memory or initialized with sane defaults depending on whether the user has already configured the system. In this mode, the `struct` can be modified.
4. A global lookup-table containing samples of the sin function. This is loaded in normal mode, and should be read-only.

It is also worth noting that the lock-free queue indicated in the design is globally allocated, and is accessed by several "threads" of execution.

The responsibilities and features of each of the software components indicated in the figure will now be described.

The **mode loader** is the entry point of the system. It turns on the indicator LED (through the use of the LED driver), and presents the user with options for choosing between settings mode or normal mode (through the LCD driver). It must also process any key press which indicates the user's choice (through the keypad driver), and load the code for settings or normal mode (through a function call).

The **settings** component will contain the code for settings mode. It must present the user with options for configuring the system (through the use of the LCD driver), and process the user's choice (through the use of the button matrix driver). When the user chooses to re-boot the system, it must also persist the chosen configuration to flash memory (through the use of the persistent storage driver).

The **DAC driver** is central to system operation, as it contains functions which generate the DTMF tones. The module should provide a function which, given a symbol, enables a timer interrupt handler (depending on what tone is to be produced) and sets the timer duration to the sampling frequency of the tone.

The module should also implement 16 timer interrupt handlers (one for each tone) which can produce tone samples and drive the DAC when a tone is being generated. The interrupt handlers will be similar in behaviour, and will share a lot of their code. However there are two benefits in having an interrupt handler for each tone. Firstly, there is no need to keep track of which tone is being generated in between invocations of the interrupt handler. This reduces the amount of global state used by the system. Secondly, specializing the code depending on the tone to be produced allows the compiler to aggressively optimize the handlers in question, e.g. applying constant division optimization.

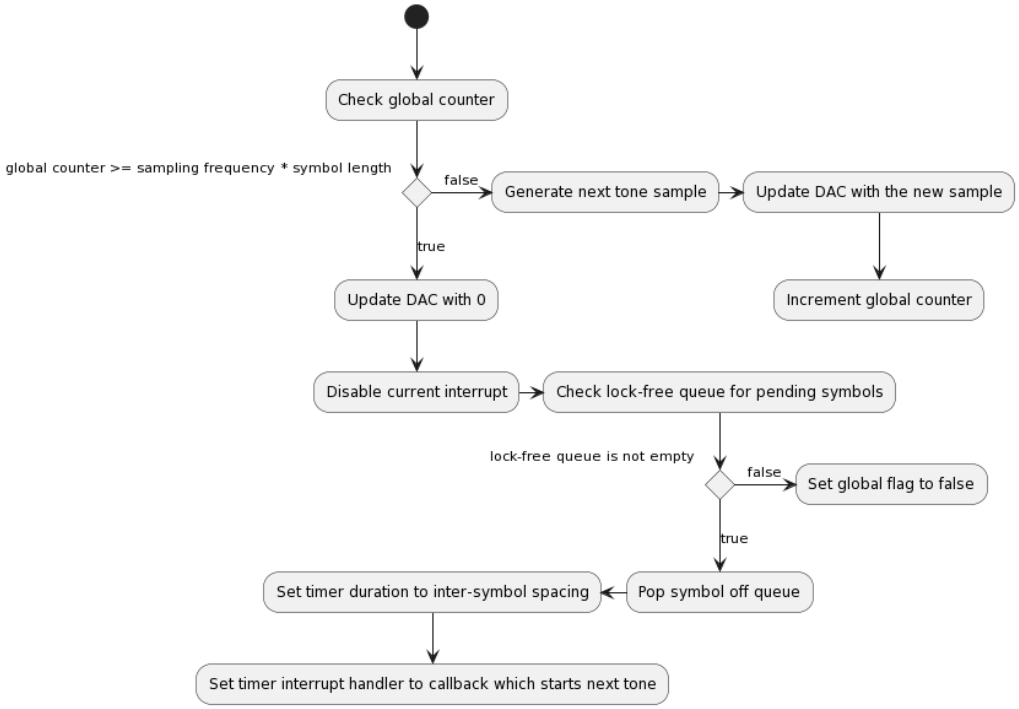

Figure 2: Control flow diagram illustrating behaviour of tone-generating interrupt handlers

The behaviour of each interrupt handler is shown in the control flow diagram in Figure 2. Note the "bootstrapping" behaviour of tone generation. When an interrupt handler detects that tone generation is done, it checks the queue for the next symbol whose tone is to be generated. If there is one, it pops it off the queue, and sets a timer according to the inter-symbol spacing. It also enables a handler for the timer interrupt which starts off the next tone's interrupt handler. Note that the check and pop from the queue must be done atomically to avoid data races.

The **keypad driver** component provides an abstraction layer over the keypad. It must implement a single polling cycle (as a function) for use in other modules.

In addition, it must implement a method that whilst running the polling cycle, performs the following actions immediately on detection of a key press:

1. It immediately outputs the DTMF symbol detected to the LCD screen.
2. It checks if there is a tone interrupt handler enabled. This can be done by atomic test and set on the global flag.
3. If there is no tone interrupt handler enabled, it takes the symbol detected, and enables a tone-generating interrupt handler to generate its tone (using a method in the DAC driver).
4. If there is already a tone interrupt handler enabled, it pushes the DTMF symbol onto the output queue.

This method will be used in normal mode to process key presses.

The **persistent storage driver** provides an abstraction layer over the flash memory. It provides functions to serialize and deserialize the C structs which need to be stored into a byte stream, and allows for loading or storing serialized data from/to flash memory. The use of a wear leveling algorithm when storing data is essential to prevent the flash memory from degrading quickly[1].

The **LED driver** provides methods to turn the indicator LED on or off.

The **LCD driver** provides an abstraction layer over the LCD screen. It must implement methods which allow a DTMF symbol or a C string to be encoded and displayed on the LCD.

The **lock-free queue** is a globally allocated thread-safe data structure. It needs to support the following thread-safe (atomic) operations:

1. Enqueue a DTMF symbol.
2. Check if queue is empty and dequeue. This must be one atomic operation.

2.3 How the polling method is invoked

There are several ways in which the polling cycle procedure that processes user input in normal mode (described above) can be executed.

The simplest method is to run the procedure in application mode, with a set period of busy waiting in between invocations of the method to avoid spurious input. (A human user will keep a key pressed for several clock cycles.)

A more sophisticated option is to run the procedure as an interrupt handler for a timer interrupt, with the timer duration set to avoid spurious input. In this version, the processor has no application code to run and can be put in sleep mode in between interrupt invocations. While more efficient than the busy-waiting approach, this method can still invoke the interrupt handler when there is no input to process.

A final, more efficient approach is to have a keypad-driven interrupt handler. In this approach, the columns of the keypad are set high, and pressing any key triggers a pin interrupt. The interrupt handler then invokes the polling method to handle the key pressed.

Each of these design choices is more sophisticated than the one before it, while re-using much of the same code. The plan is hence to implement each of these designs in turn. When one implementation has been successfully tested, the team will move on to implementing a more sophisticated design.

3 Management

Include a time plan. Show task dependencies. How are these going to be managed?

4 Closure

References

[1] A. Silberschatz, G. Gagne, and P. Galvin, “Operating system concepts,” ch. 11, Wiley, 2018.