1. Introduction

Given requirements were to write a program that reads button processes and lights LEDs, with the suggestion that the program should have a purpose. I decided to create a metronome with some additional features – shown in the instruction diagram (fig. 1) and further outlined in the specification below.

Figure 1. Instructions for usage, outlining which buttons correspond to which features

2. Specification

- Metronome that flashes LEDs consistently at a particular tempo, without any noticeable variation or drift.
- The tempo can be adjusted with up/down buttons on the board.
- The sequence of the flashes allows certain beats to be accented for different time signatures.
- The time signature can be adjusted with up/down buttons on the board.
- A tempo can be rapidly changed by using 'tap tempo' - the user taps on the tap tempo button at the target tempo, and the system changes to that tempo automatically.
- The metronome can be re-synchronised (i.e. the beat starts again, at the same tempo, from the moment the synchronise button is pressed).

3. Implementation

System Overview

There are two main execution paths: the loop in the `main` function, which executes indefinitely, and the timer interrupt handler, `TIM2_IRQHandler`, which is raised every 2 milliseconds.

Using a dispatch model for events keeps the program structure much cleaner in the main loop. In practice, these handlers are inlined where possible, as they are only used in one place.

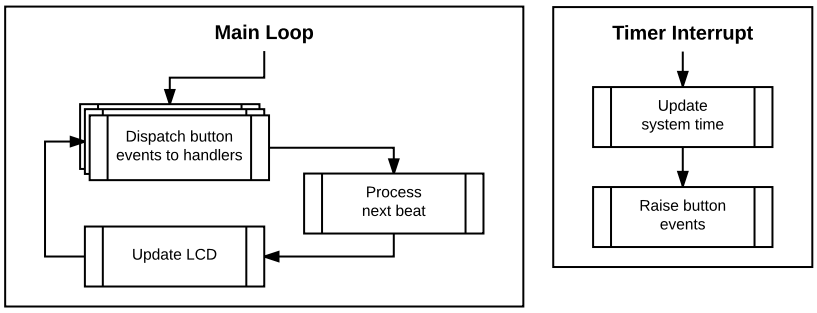

Figure 2. System overview diagram

Individual Functions

`main()`

Responsibility for dealing with particular button events is given to the individual handlers. The rest of the function sets the LEDs according to patterns specified in an array (corresponding to each beat in the time signature).

The LCD is updated only if the update flag has been set by the event dispatcher. This prevents unnecessary LCD rewrites when the program state has not changed.

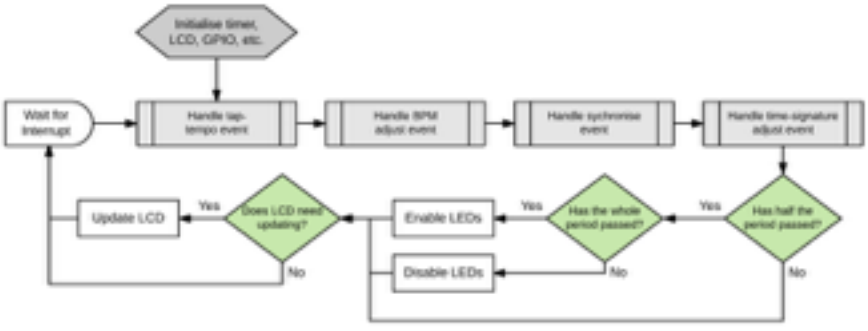

Figure 3. Flow diagram for `main` function

The processor is put into standby after each run of the main loop using the WFI (wait for interrupt) instruction. This is for efficiency – as the program state is triggered by changes in the timer interrupt, nothing will have changed until the next interrupt, so there's no reason to loop indefinitely.

`TIM2_IRQHandler()`

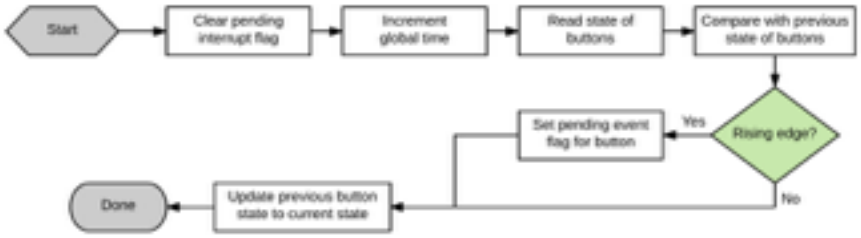

Figure 4. Flow diagram for `TIM_IRQHandler` function

This updates the global system time and polls the button state.

In order to avoid duplicate events being processed when a button is held down, the function only raises an event on the rising edge of a button state change (when it is first pressed, and then not again until it has been released). This architecture has a drawback: it is not possible to detect buttons being held down continuously, which would have been useful to allow the user to adjust tempo more quickly.

`handle_event(event_mask, handler)`

Figure 5. Flow diagram for `handle_event` function

Dispatches to an event handler (function pointer) if an event mask is matched.

Setting the LCD update flag is done in this function to avoid duplicating code in the individual handlers. Setting it in the handlers themselves would eliminate some unnecessary rewrites, but it's rare that an event triggers without the LCD needing an update, so it's not worth the added complexity.

`tap_tempo_recalculate()`

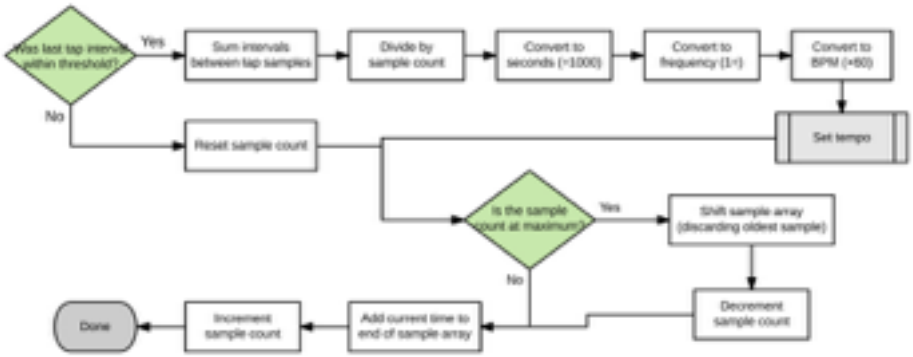

Figure 6. Flow diagram for `tap_tempo_recalculate` function

When the tap tempo button is pressed, a sample is taken of the system time. To determine the tempo, the system takes the average interval between the most recent samples.

Samples are overwritten when the sample array is filled, and discarded past a cut-off time (1.5s).

`synchronise()`

Sets the last beat as having happened exactly one beat period ago. Consequently, the next tick will become the first beat.

`set_tempo(new_tempo)`, `tempo_increase()` AND `tempo_decrease()`

`set_tempo` changes the system tempo and then recalculates the time between beats in milliseconds, the beat period, from this tempo (which is then used by the `main` function). The latter two functions increment and decrement the current BPM respectively, capped between 1bpm and 999bpm.

`timesig_increase()` AND `timesig_decrease()`

Information about the current time signature, such as its label and the sequence of LED patterns, is tracked in arrays. These two functions increment and decrement the program's index for these arrays which specifies which time signature is currently selected – this is then used by the `main` function to look up beat information and to update the LCD.

`led_init()`, `timer_init()`, AND `button_init()`

These just initialise the peripherals ready for use. The LED and buttons use GPIO-D and GPIO-E respectively. The timer used is the TIM2 timer, which also has to have interrupts set-up using the interrupt controller (NVIC).

4. Conclusion

To verify the accuracy of the tap tempo system, I compared the output from the metronome when tapped with the output of a commercially available metronome, and I was satisfied with the result. As an improvement, I would like to use the rotary potentiometers on the board as a way to quickly adjust the tempo. I believe this would require use of the ADC features and I felt it was beyond the scope of the assignment to implement this.