Group 16 Final Report

Riley Symon 48962845
Grace Kaye-Blake 38604492
Jonathan Edwards 27033004

As a team we were tasked with programming a personal fitness monitor to count a user's steps and estimate distance travelled. The program was run on a TIVA microcontroller with a BoosterPack peripheral board. This program records activity data via an accelerometer and displays information via the peripheral board's OLED screen. This display is controlled by the up, down, leeft, and right buttons on the TIVA board and the BoosterPack peripheral board.

Identification of Tasks and Analysis

For milestone 1 there were specifications that had to be implemented. These specifications were as follows:

- The accelerometer must be sampled at regular intervals, and mean values of the samples must be calculated at regular intervals.
- The accelerometer data must be stored in three circular buffers, for x, y, z directions.
- When the program starts, mean sample values must be calculated as the reference orientation.
- From initiation, the raw x, y, z accelerations must be displayed on the OLED display
- From initiation, pressing DOWN/BTN1 must recalculate the reference orientation.
- Pressing UP/BTN2 must cycle the OLED display to show raw x, y, z accelerations, to x, y, z multiples of gravity on Earth (9.81 ms^{-2}), to x, y, z units of metres per second (ms^{-2}). Pressing the button repeatedly must cycle between these displays in the given order.

All of these requirements were fulfilled for the milestone 1 demo.

Shortly after the completion of milestone 1, the project specification was altered significantly. Milestone 2 was removed and incorporated into the final result. This was due to the covid-19 lockdown closing the university campus. The project specification was shortened and simplified to reflect the new home learning environment we had to adapt to.

This also meant we were unable to collaborate as a team in person, and had to use online tools to complete the project. This added another challenge to the project, as some of these online tools proved challenging to use.

Following milestone 1 we determined the code could be made more modular. As a team we developed a use case diagram to lay out how our program would operate with any given input. These inputs and the events they triggered were outlined in the updated project specification.

This use case diagram helped with our understanding of how our program would operate and how the various inputs would affect the program's behavior.

For the final project demo, there were, again, specifications that had to be implemented. These specifications were as follows:

- The program must estimate the number of steps taken by a user based on data from the accelerometer on the Orbit board.
- The program must estimate the distance travelled by the user. The total distance travelled must be calculated from the number of steps multiplied by 0.9 metres.
- The RESET button on the TIVA board must restart the program and clear any stored step and distance values.
- In addition to the requirements above, the display, buttons, and switches on the TIVA and Orbit boards must function according to the requirements given below:
    - At startup the OLED board must display the number of steps counted since the last reset.
    - Pushing the LEFT or RIGHT buttons on the Orbit board must toggle the display between total distance travelled since last reset and number of steps counted since last reset.
    - When the distance travelled is displayed, pushing the UP button on the TIVA board must toggle the units between kilometres and miles.
    - A long press on the DOWN button when the number of steps are displayed must reset steps to zero; similarly a long press on the DOWN button when the distance travelled is displayed must reset the distance travelled to zero.
    - Power cycling the board must reset step count and distance travelled to zero.
    - Setting SW1 to the UP position must put the fitness monitor in a test mode, where the functionality of the GUI can be verified. In this test mode each push of the UP button must increment the step count by 100 and the distance by 0.09 km. Likewise, pushing the DOWN button must decrement the step count by 500 and the distance by 0.45 km. Note that the other functions of the UP and DOWN buttons, namely toggling units and resetting counts, must be disabled while SW1 is UP. The functionality of the LEFT and RIGHT buttons must not be affected by SW1. Setting SW1 to DOWN must restore the normal functionality of the UP and DOWN buttons.
- When the program starts, which may happen after a 'reset' operation or after reprogramming, the step and distance values must be zero.
- The program must have a real-time foreground/background kernel operating on a round-robin basis for background tasks. Robust behaviour should be maintained at all times.

All of these requirements were fulfilled for the final demo.

There were some things that we had to do for milestone 1 that we did not have to do for the final demonstration. For example, we did not have to read from the accelerometer at regular intervals. This meant that we could put the accelerometer reading in the background tasks,

rather than reading it during an interrupt. This is a good thing, since reading from the accelerometer can take a while. However, there were still things that we could keep from the milestone 1 code, like the code that read and averaged the measurements from the accelerometer, and the code that checked that the buttons were pushed.

Scheduler and Inter-task Communication

There is one interrupt used in the program, the `SysTickIntHandler`. This is triggered regularly, every 2.5 ms. Everytime the function is called, the buttons and switch are polled. Every second time, the slow tick boolean is set to true (figure 1). The background tasks are:

- If slow tick is set to true, the accelerometer is read from, and the number of steps and distance travelled are updated.
- The current state is updated.
- The display is updated.

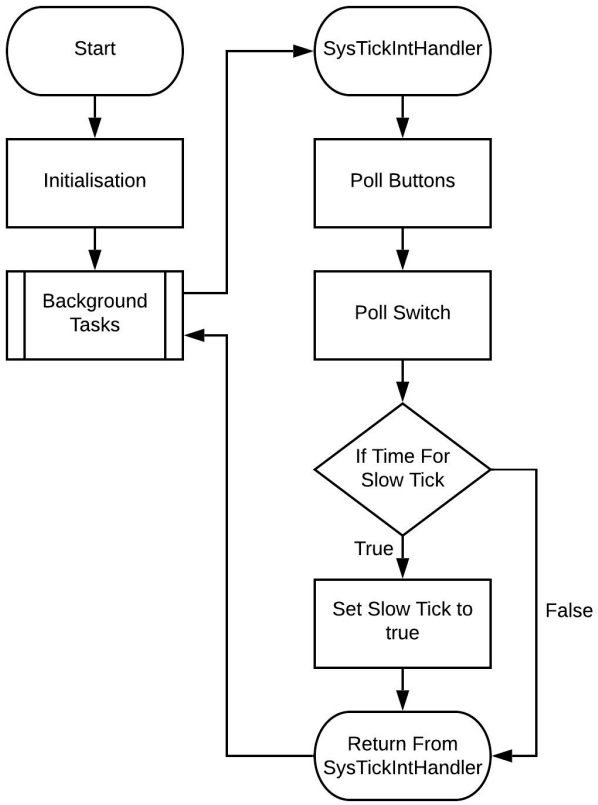

Figure 1: A round robin diagram - describes process during interrupt.

The background tasks are scheduled in a round robin fashion (table 1). The interrupt ensures that the buttons and switch are polled regularly. It does not, however, make sure that a button's state is read before it is overwritten. We tested the device thoroughly and did not find that this was a problem, however if there had been a more time consuming task added to the background tasks, it could become a problem. There is also no debouncing beyond the regular polling, but, again, we did not encounter problems with this.

Table 1: The number of clock cycles of some of the functions important to the scheduler.

Function	**Clock Cycles**
updateTickProcess (a)	78, 709 or 78, 718
progUpdate (b)	109, 281
btnCheck (b.1)	172
displayUpdate (b.2)	109, 052
NORM_2 - Macro (a.1)	69
getAcclData (a.2)	78, 008

Analysis on this table and other time critically intensive functions. The ISR function `SysTickIntHandler` operates at a rate of 50 Hz. This polls the peripheral tasks which occur at a rate of 50 Hz, whereas slow tick tasks run at 25 Hz. The function `updateTickProcess` occurs in the slow tick state i.e when slow tick == 1 and the switch is in NORMAL mode. We can do calculations to see how much space is left in the time slot by, $\frac{1}{25\,(HZ)} \times 20 \times 10^{6} = 800k\,(allowed\ clk\ cycles)$. Since $78k < 800k$ therefore the CPU load is only 9.75 %. Our second slowest process is `progUpdate`, and since this process happens in the background i.e main thread. This process takes what is left over from the timing slot space for every tick. Since There is plenty of space, there is no need to make this code any faster (also this includes the `displayUpdate` which takes up all of the space as shown above).

To communicate between the interrupt and the background tasks, we used static variables in the initialise and button modules. The static variables needed to store the state of the buttons, the state of the switch, and whether it was time for a slow tick. The button states were stored as a boolean array in the button module, and the state of the switch, and whether it was time for a slow tick were both stored as booleans in the initialise module.

Design and Justifications

There are several modes that the device could be in. These modes were:

- The testing and normal modes
- The step and distance display modes
- The miles and kilometres modes within the distance display mode

We visualised these collections of states as a finite state machine while planning (figure 2), but found that this was not a good way to implement the program. It made more sense to keep track of the “state” with three independent boolean values. The boolean variables were wrapped in a structure, `_CurrentState`. `_CurrentState` also contained variables to keep track of the number of steps taken, and the distance travelled.

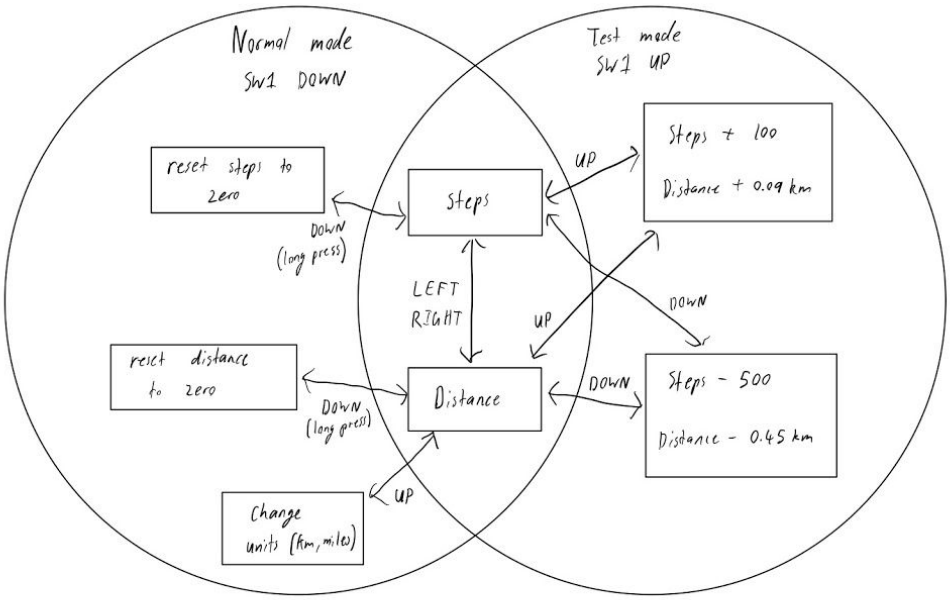

Figure 2: State diagram used in our program design.

As stated in the Identification of Tasks and Analysis section, we also had a strong focus on modularising our project. The modules of the project were:

- The initialisation module, which was run at the beginning of the program to initialise other modules.
- The kernel, model, and control modules, which schedule and run the tasks in other modules.
- A UART driver to communicate with the board to help with debugging.
- The display module, which displayed the number of steps and distance travelled. This module used the provided OLED library.

- A module to read in button pushes.
- A module for communicating with the accelerometer, which depended on the I2C driver.
- The circular buffer module, which stored readings from the accelerometer.

We found that, since the peripherals are quite distinct, it was easy to keep the modules that related to specific hardware cohesive and loosely coupled. It was more difficult to achieve that for the kernel, model, and control modules, since the modules are similar. We did not have a very clear idea at the beginning of where the boundaries were for each of these modules, and this is something we would spend more time on in the planning phase if we were to redo the project.

Conclusion

Our program successfully fulfilled all the project criteria for the final demonstration. The task scheduler uses interrupts to keep the buttons and switches polled at regular intervals, and our background tasks are executed in a round robin style. Although it would have been good to think about the modularisation earlier, the final code is modularised well. From the state diagram we made, we implemented the various states of the program as state variables. This was slightly different from the diagram but was the easiest and most efficient way to implement the desired behavior of the program. We also deviated from the diagram by modularizing our code base.