ENCE361 Step Counter README

T. Linton, J. Legg

May 2025

Table of Contents

1 Hardware Overview

This project implements a modular step counter system built around the STM32C0 series microcontroller on the NUCLEO64 development board, interfaced with a RCAP board [1] [2]. The system is designed to detect steps in real time and provide live goal progress to the user via an OLED display. The user can interact with the step counter using a joystick, potentiometer, and 4 push-buttons.

1.1 Peripheral Board (RCAP)

It includes a 6-DOF inertial measurement unit (IMU), a buzzer, a joystick, 4 LEDs and a potentiometer. The functionality that the RCAP board has been implemented with is as follows:

- Buttons
  - When *SW1* is pushed:
    - The *Step Count* is increased by 80 steps.
    - This is used for testing and is easily removed.
  - When *SW2* is double pushed (within 300 *ms*):
    - The *Step Counter* enters *Test Mode*.
- Joystick
  - Moving the *Joystick* left and right (x-axis):
    - Changes what is displayed on screen.
  - Moving the *Joystick* up and down (y-axis):
    - Changes the units displayed on screen.
    - Changes the *Step Goal* in *Test Mode*.
  - Pushing the *Joystick* in for over 1 second when in *Goal Progress Mode*:
    - Enters *Set Goal Mode*.
    - The user can then change the *Step Goal* using the *Rotary Potentiometer*.
    - Saving the new *Step Goal* is done by pushing the *Joystick* for at least 1 second.
    - Reverting the to the original *Step Goal* is done by pushing the *Joystick* for less than 1 second.
- Rotary Potentiometer
  - Selects a new *Step Goal* between 500 and 15,000 steps.
- Screen
  - Current *Step Count*.
  - The *Step Goal*.

    - Distance travelled (kilometers & yards)
    - Whether *Test Mode* is enabled.
- LEDs
    - Four LEDs light up in anticlockwise order.
    - One LED for every 25% of the step goal being reached.
    - The first LED uses pulse-width modulation (PWM) to increase in brightness gradually increases brightness for the first 25% of the goal.
- Inertial Measurement Unit (IMU)
    - Provides 3-axis accelerometer data for step detection.
    - Data is filtered and used to calculate movement magnitude.
- Buzzer
    - Lights up for one second when the S*tep Goal* is reached.
    - The *Buzzer* uses PWM at a fixed and duty cycle.

2 Modularisation

2.1 Main Application

The Step counter is split into 7 main tasks/modules:

1. *Buzzer*
2. *LEDs*
3. *Buttons*
4. *Display*
5. *Joystick*
6. *Read IMU*
7. *Read ADC*

Splitting the program into these separate tasks is essential. The *Buzzer* and *LEDs* tasks are not required to happen as frequently as the *Read IMU* and *Buttons* tasks.

The Step Counter's variables that are required globally (*Current Display State, Step Count, Step Goal*, etc.) are stored within the *State Machine* module as a *StepCounterStateMachine* struct as shown in Appendix A - State Machine Struct. They are read and updated using getter/setter functions respectively. This level of abstraction is beneficial as no other module needs to know the details of how these fields are stored or managed. Instead, other modules can call functions such as *stateMachine_NextState(), stateMachine_PreviousState(), stateMachine_IncrementStepCount(), stateMachine_DecrementStepCount()* and *stateMachine_StepCountGetter()*.

If every module that incremented the step count needed *if/*else statements to check if the goal had been reached and if the user is in *Test Mode*, the code would be inefficient and difficult to maintain. This way, if the step increment logic requires changing, it only needs to be changed in one place.

A full dependency graph is shown below in Figure 1. The blue modules are high-level tasks, and purple modules are helper modules. Orange represents lower-level modules that interact with the Hardware Abstraction Layer. The *Hardware Abstraction Layer* is drivers provided by ST Microcontrollers [3].

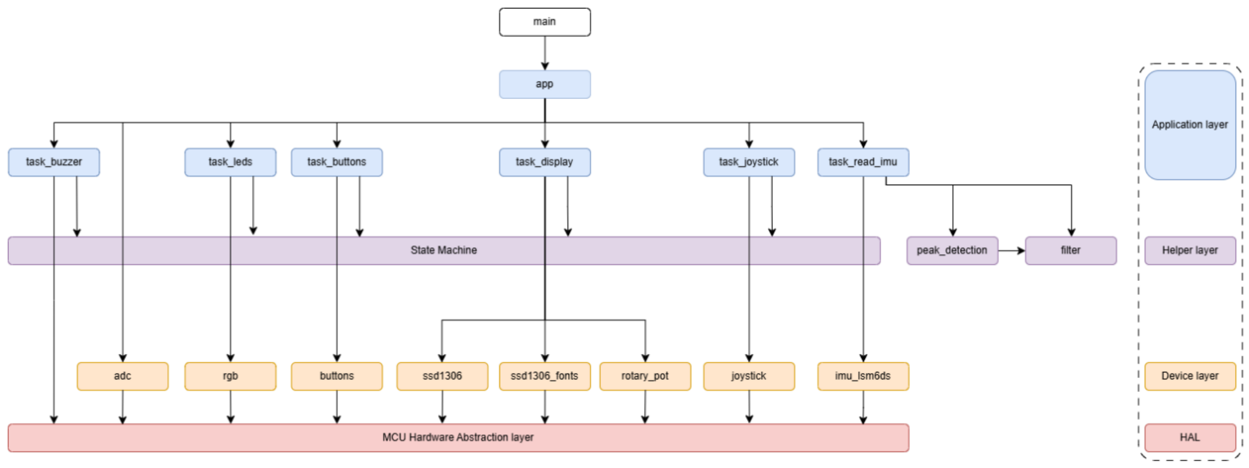

Figure 1: Step Counter hierarchical dependency graph.

2.2 Buzzer Task

The *Buzzer* task module is responsible for managing the state of the *buzzer*. It keeps the low-level PWM implementation separate from the high-level control logic. This means that any changes to the buzzer's behaviour (for example, modifying the beep pattern or frequency) can be made exclusively within the *Buzzer* task module without affecting other parts of the system. This modularisation leads to simplified maintenance, easier testing, and overall, a more robust, scalable application architecture.

2.3 LEDs Task

The *LEDs task* is modularised by isolating low-level PWM and LED control from the logic. It uses *State Machine* getters to retrieve the current *Step Count* and *Step Goal*, computes progress and then controls which LEDs to light or dim via the RGB and PWM modules.

This allows the *LEDs task* to avoid low-level configuration details, keeping the code clean, focused, and easily modifiable with different LED behaviour (e.g. blinking on goal complete) can be added without affecting unrelated modules.

2.4 Buttons Task

task_buttons.c is responsible for polling the buttons with *buttons_update()*, defined within *buttons.c*. Once the status of the buttons is received, the appropriate action is executed by calling functions within the *state machine* module.

Low-level interfacing to the physical buttons is handled within the *buttons* module. This module implements polling and debouncing. It also tracks *SW2* to see if it is double pressed along with the *Joystick* click to see if it is held for more than 1 second. Having this abstracted away from the *task_buttons* module, is helpful as the polling in *buttons.c* can be implemented for any GPIO pin and therefore, more buttons can be easily added.

To increase the modularity, *buttons_CheckDoublePush()* and *buttons_CheckHold()* should be expanded for all buttons like the rest of the functions in *buttons.c*.

2.5 Display Task

The *task_display* module is well modularised by separating what is shown from how it is shown. High-level display logic resides entirely in *task_display*, where the current step counter data is retrieved through *State Machine* getter functions. The function *taskDisplay_Execute()* determines what information to display based on the current system state. It uses helper functions like *taskDisplay_PrintSteps()* and *taskDisplay_Distance()* to build display strings, keeping the logic easy to follow and extend.

Low-level OLED operations (initialisation, drawing, updating the screen buffer) are handled entirely by the ssd1306 driver module. This abstraction means *task_display* doesn't deal with pixel placement or I2C communication. As a result, the display logic is decoupled from hardware, making the system more maintainable. For example, switching to a different screen module would only require changes in ssd1306.c, with zero impact on the display task. This approach ensures each module has a single responsibility, simplifies debugging, and allows each module to be developed and tested independently.

2.6 Joystick Task

The *Joystick task* is modularised by separating input acquisition, state handling, and system actions. *task_joystick.c* handles only high-level logic such as deciding when to change the display state based on the joystick direction. It calls getter functions from *joystick.c* and *State* Machine module, keeping the task free of hardware-specific logic.

Low-level analogue reading and position calculation are abstracted into *joystick.c*, while ADC data is updated externally by the ADC module. This design ensures that the joystick task only interprets intent, not raw signals, making it portable and easier to test or modify independently of hardware or signal processing logic.

2.7 Read IMU Task

The IMU system is modularised by separating responsibilities across distinct components. *task_read_imu* handles raw sensor reading and applies axis offsets but delegates all filtering and statistical analysis to the filter module. The filter module performs exponential moving average (EMA) filtering, magnitude calculation, and maintains a running mean and variance buffer. Step detection is handled entirely by the *peak_detection* module, which uses only the statistical outputs ensuring it remains decoupled from sensor details.

Shared data is accessed with getter functions, avoiding direct coupling between modules. This structure allows each module to be developed, tested, or replaced independently. For example, swapping the EMA filter for a more advanced filter would only require changes within the *filter*.

2.8 Read ADC Task

The main application module (*app.c*) depends on the ADC module only to call *adc_Execute()* with specific timing. This task serves solely to update the ADC values for the *Joystick* and *Rotary Potentiometer*. By keeping this functionality separate from the *Joystick task*, other modules can independently access up-to-date ADC readings without duplicating code or introducing dependencies. This abstraction makes it easier to scale the system such as, adding new analogue inputs requires changes only within the ADC module, not in every component that uses ADC values.

3 Implementation and Firmware

3.1 Filtering

The acceleration data received in 3-axis are filtered using an Exponential Moving Average Filter (EMA). The EMA filter is an adaptation of an Arduino C++ library on GitHub [4] and is shown in Appendix B – EMA Filter.

The formula for the cut-off frequency of an EMA filter is

$$f_c = \alpha \frac{f_s}{2\pi}.$$

Where $\alpha = 1/2^{ALPHA\_SHIFT}$ and f_S is the sampling frequency. *ALPHA_SHIFT* is defined in *filter.c*. Therefore, with *ALPHA_SHIFT* = 3 and a sampling frequency of 100 Hz (the task frequency), the cut-off frequency is approximately 2 Hz.

The advantage of using *ALPHA_SHIFT* rather than α is increased efficiency. This is due to α always between 0 and 1 and thus computationally expensive to multiply by without a floating-point unit (FPU). By choosing $\alpha = 1/2^n$, the EMA update function runs in *O*(1) time without floating point math using a right bit shift. The transfer function is shown below in Figure 2.

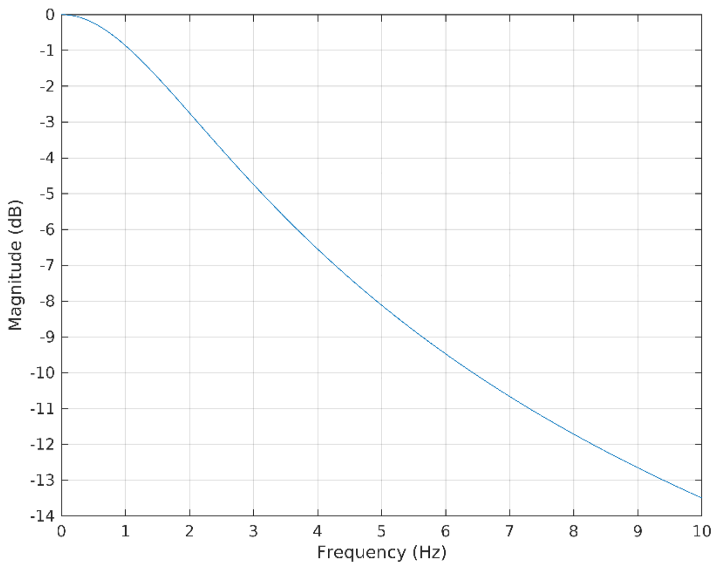

Figure 2: Frequency response of the EMA filter.

While the axes are filtered individually, it is helpful to visualise the impact of the filter on the acceleration magnitude. A plot of the filtered and raw magnitude is shown in Figure 3. Due to the filtering, there is a delay in response (phase shift).

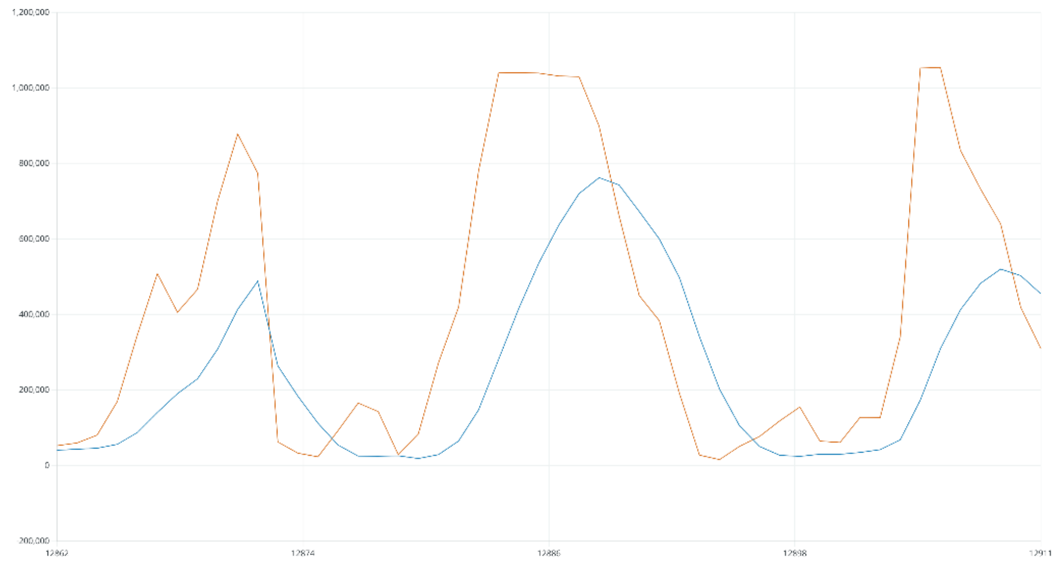

Figure 3: Raw magnitude (orange) vs. filtered magnitude (blue).

3.2 Calibration

To calibrate the IMU sensor, three offsets were recorded. This ensures that when each axis is not under any acceleration, it reads zero. To find the offset values, a heavy low-pass-filter was applied to the raw data and each axis was placed still under no acceleration from gravity. This provided a stable long-term average to adjust the offset easily. The x-axis after offsets were applied and the heavily filtered data are plotted below in Figure 4.

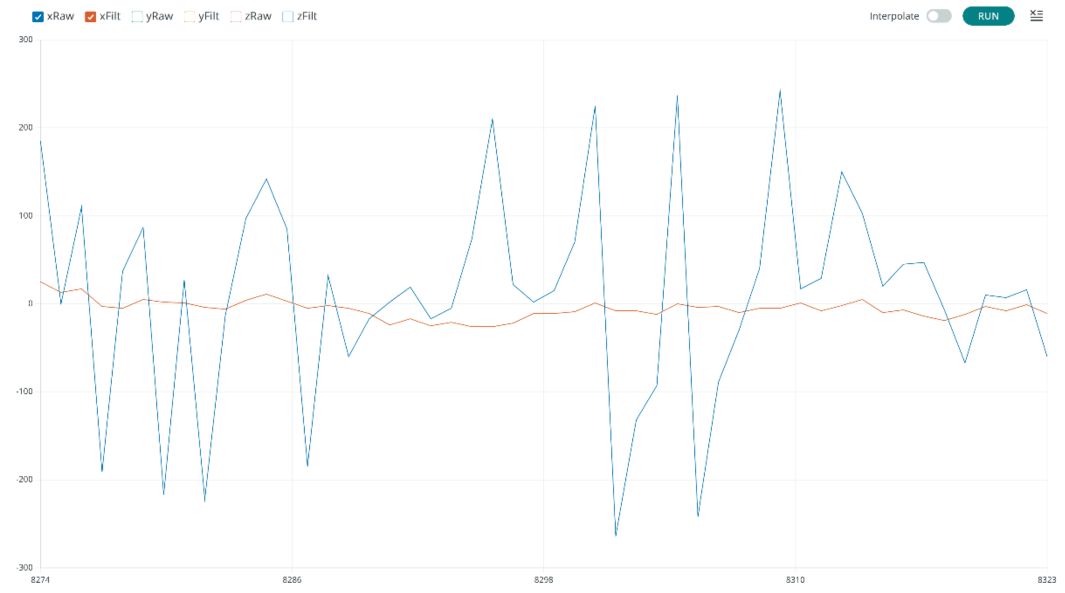

Figure 4: X-axis data with offset applied (blue) & heavily filtered data (orange).

Within the *Peak Detection* module, three variables define how sensitive the device is to steps. Two of which are plotted in Figure 5 and Figure 6. The green line is the variance threshold, set so that it is slightly higher than the variance when standing still. This is so the steps are not counted when standing still. The mean threshold (the yellow line) is set by adding a delta to the mean. Figure 6 shows the variance (orange line)

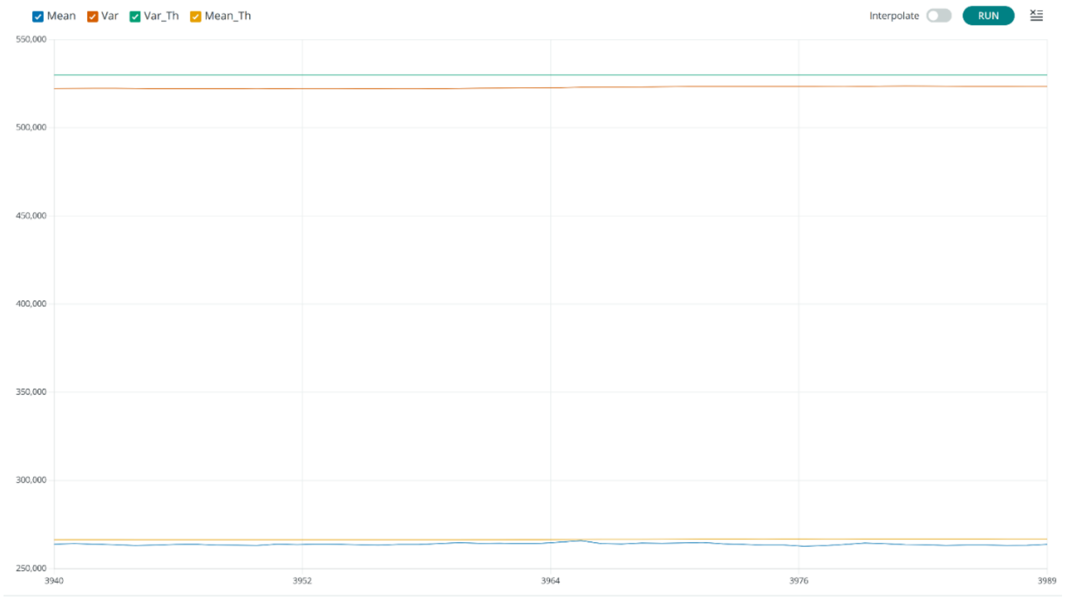

Figure 5: Peak Detection calibration - standing still.

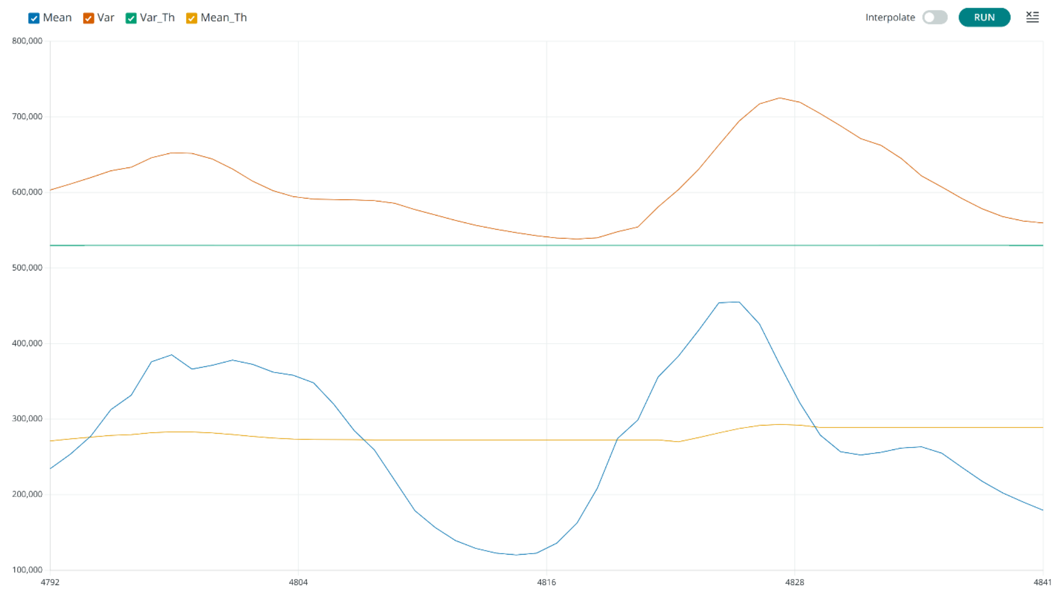

Figure 6: Peak Detection calibration – walking.

3.3 Profiling & CPU Load

As shown in Table 1, the total time to run all tasks (31 μs) is well below the shortest required period (1000 μs), resulting in a CPU load of only 12.2%. Therefore, all tasks can be executed sequentially without missing deadlines. With this much headroom, the system remains robust even if some tasks take slightly longer in their worst-case scenarios. For example, the Buzzer task may take marginally more time when the buzzer is active, as the measured time likely reflects its inactive state.

Table 1: Profiling and CPU load.

Task	Required Frequency (Hz)	Required Period (μs)	Measured Task Time (μs)	CPU Load (%)
SysTisk ISR	100	1,000	2.6	2.60
Read IMU	100	1,000	5.8	5.80
Buttons	100	1,000	3.0	3.00
Joystick	8	125,000	1.9	0.15
READ_ADC	8	125,000	1.6	0.13
LEDs	4	250,000	2.0	0.08
Display	4	250,000	10.6	0.42
Buzzer	1	1,000,000	3.5	0.04
Total			**31.0 μs**	**12.22%**

3.4 Kernel

The Step Counter is implemented with a time-driven interrupt kernel. It uses the *HAL_GetTick()* function and underlying *SysTick* interrupt to schedule the program tasks. Each task maintains its own "last run" timestamp. When the difference between the current tick and that timestamp meets or exceeds the task's required period, the scheduler invokes the task and updates its timestamp accordingly. This approach delivers several benefits.

First, it decouples timing from execution. This means each task only executes when its timed interval elapses. Given the measured total execution time of 31 μs well beneath the shortest 1 ms period, there is headroom to absorb overruns without compromising the timing of other tasks.

Second, this design scales easily. Adding new tasks only requires implementing a timestamp check with no need to restructure the main loop. Should future requirements demand higher frequencies, the existing scheduler framework is simply updated with the new periods.

Finally, more complex kernels that implement pre-emption are not required for a simple step counter.

4 Conclusion

This project demonstrates a well-structured and modular implementation of a step counter system using the STM32C0 microcontroller and RCAP board. Each hardware feature is abstracted into its own module, allowing for clean separation of responsibilities. The use of getter/setter functions and low-level driver modules ensures minimal coupling between components and improves maintainability.

By adopting a time-driven interrupt scheduler and avoiding unnecessary complexity, the system remains reliable, efficient, and easy to extend. The result is a responsive and user-friendly step counter with a clear and scalable architecture.

5 References

[1] ST, “STM32C071KB Datasheet,” [Online]. Available: https://www.st.com/resource/en/datasheet/stm32c071kb.pdf. [Accessed May 2025].

[2] ST Microelectronics, “UM3353 - STM32 Nucleo-64 boards (MB2046) User Manual,” January 2025. [Online]. Available: https://www.st.com/resource/en/user_manual/um3353-stm32-nucleo64-board-mb2046-stmicroelectronics.pdf. [Accessed May 2025].

[3] ST Microcontrollers, “UM3029 - Description of STM32C0 HAL and low-layer drivers User Manual,” January 2025. [Online]. Available: https://www.st.com/resource/en/user_manual/um3029-description-of-stm32c0-hal-and-lowlayer-drivers-stmicroelectronics.pdf. [Accessed May 2025].

[4] ST Microelectronics, “LSM6DS3TR-C - iNEMO inertial module:,” May 2017. [Online]. Available: https://www.st.com/resource/en/datasheet/lsm6ds3tr-c.pdf. [Accessed May 2025].

[5] F. R. R. Carmona, “EMA,” 2022. [Online]. Available: https://github.com/RafaelReyesCarmona/EMA. [Accessed 10 May 2025].

6 Appendices

6.1 Appendix A - State Machine Struct

```
typedef enum {
    STATE_CURRENT_STEPS,
    STATE_DISTANCE_TRAVELLED,
    STATE_GOAL_PROGRESS,
    STATE_SET_GOAL
} DisplayState;

typedef enum {
    UNITS_STEPS,
    UNITS_PERCENT,
    UNITS_KM,
    UNITS_YD
} UnitDisplayMode;

typedef struct {
    DisplayState current_display_state;
    bool test_mode_enabled;
    UnitDisplayMode unit_mode;

    uint32_t step_count;
    uint32_t goal;

    bool goal_completed;
    UnitDisplayMode previous_unit;
    uint32_t saved_step_count;
} StepCounterStateMachine;
```

6.2 Appendix B – EMA Filter

```
/*
 * IIR filter: y[n] = y[n-1] + alpha * (x[n] - y[n-1])
 * where alpha = 1/(2^ALPHA_SHIFT)
 */
void  filter_IIR  (int16_t  new_x,  int16_t  new_y,  int16_t  new_z,  int16_t*
imu_filtered)
{
      int32_t tempX = (int32_t) imu_filtered[0];
      int32_t tempY = (int32_t) imu_filtered[1];
      int32_t tempZ = (int32_t) imu_filtered[2];

      tempX += ((int32_t) new_x - tempX) >> ALPHA_SHIFT;
      tempY += ((int32_t) new_y - tempY) >> ALPHA_SHIFT;
      tempZ += ((int32_t) new_z - tempZ) >> ALPHA_SHIFT;

    imu_filtered[0] = tempX;
    imu_filtered[1] = tempY;
    imu_filtered[2] = tempZ;
}
```