International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.152, November 2026

Design and Implementation of an ESP32-based Portable Digital Multi-Tool with Finite State Machine Navigation

Anvay Hemant Mayekar
Dept. of ECS
SAKEC, Mumbai

Jeel Nimish Mashru
Dept. of ECS
SAKEC, Mumbai

Param Prakash Mehta
Dept. of ECS
SAKEC, Mumbai

Mohammed Hamza Afzal Memon
Dept. of ECS
SAKEC, Mumbai

Saeeda Chimthanawala
Asst. Professor, Dept. of ECS
SAKEC, Mumbai

ABSTRACT

This paper presents the design, firmware structure and simulation-based evaluation of a rechargeable multi-function digital instrument built around the ESP32 WROOM-32. A clock, a centisecond-resolution stopwatch, a configurable countdown timer, a non-volatile tally counter and an information screen are tied together by a six-state Moore finite state machine (FSM) with a central menu hub. Three push buttons are conditioned entirely in software, and a 3.7 V 500 mAh LiPo cell with a TP4056 charger makes the unit standalone. The stopwatch and timer derive elapsed time from a stored reference timestamp rather than per-loop increments. No physical bench measurements are reported. The evaluation combines four labelled kinds of evidence: an Arduino IDE compilation (25.83% of program storage, 7.27% of static RAM), a Wokwi functional walkthrough (13 of 14 checks passed; counter persistence not demonstrated in the simulator), an algorithm-level simulation driven by synthetic inputs, and a datasheet-based power estimate. The reference-timestamp scheme avoids loop-dependent drift by construction, and an illustrative simulation under stated assumptions shows accumulating schemes drifting by seconds over 10 minutes; analysis shows that timer expiry is detected within one loop period and that presses shorter than that period can be missed. A datasheet-based estimate of 37–104 mA implies roughly 4–13 hours of runtime from the 500 mAh cell. Validation on the physical device remains future work.

General Terms

Embedded Systems, Design, Simulation

Keywords

ESP32, finite state machine, Moore machine, SSD1306, I2C, OLED, software debounce, TP4056, LiPo, NVS, Wokwi, portable instrumentation, stopwatch, countdown timer

1. INTRODUCTION

Engineering work and field diagnostics regularly call for a few basic digital tools: a clock, a way to measure short durations, a countdown timer and an event counter. Separate gadgets are cumbersome and a smartphone brings notifications, a shared battery and distraction, so consolidating these functions into one dedicated unit with a purpose-built physical interface is attractive.
The ESP32 is well suited to this role: unlike legacy 8-bit architectures such as the Intel 8051 or Atmel AVR, it pairs dual Xtensa LX6 cores at up to 240 MHz with an extensive peripheral set, built-in non-volatile storage and power management, all in one SoC [1], while Arduino-compatible tooling keeps development approachable without sacrificing low-level access.
This paper describes the build of the ESP32 Multi-Tool, with most of the discussion centred on the software: the FSM navigation model, the button-handling pipeline and the timing subsystems. Section 2 surveys related work; Section 3 covers the hardware; Section 4 presents the firmware and FSM model; Section 5 gives the timing analysis; Section 6 reports the simulation-based evaluation, which is not a set of bench measurements; Section 7 discusses limitations; and Section 8 concludes.

2. RELATED WORK

Work on the ATmega328P, surveyed by Beningo [2], showed that FSM-driven menu systems are feasible on 8-bit hardware, but 2 KB of SRAM and the absence of on-chip NVS meant that persistent data required an external EEPROM.
Moving to ARM Cortex-M brought 32-bit arithmetic and hardware floating point to battery-powered instruments [3], at the cost of HAL complexity and a steeper configuration burden. Kolban's survey [4] gives a broad overview of the ESP32 platform; much of the openly published ESP32 literature emphasises network and cloud connectivity, whereas standalone, formally specified FSM-driven instruments with integrated battery charging were harder to find among the sources reviewed.

Table 1. Comparison with a Related Single-Function Embedded Timer

Feature	This work	Ajao et al. [5]
Controller	ESP32 (dual-core, 240 MHz)	PIC16F887 (single-core, 20 MHz)
Display	SSD1306 OLED 128×64	16×2 LCD + 7-segment
Functions	5 (clock, stopwatch, timer, counter, info)	1 (countdown timer + alarm)
Navigation	Formal 6-state Moore FSM	Not formally specified
Persistent storage	NVS flash (by design)	Not reported
Power source	Rechargeable LiPo (portable)	Continuous 5 V USB (not battery)

Closer to this work in application context, Ajao et al. describe a microcontroller-based countdown timer and alarm system for conference presentations, built around a PIC16F887 with a 16×2 LCD, a cascaded seven-segment display, a three-colour LED indicator and a buzzer [5]. That design implements a single timing function rather than a menu-driven set of modes, is USB-powered rather than battery-powered, and reports neither persistent storage nor a formally specified navigation model (Table 1).
The contribution of this work is not any single technique, since Moore-machine menus, software debounce and flash-backed persistence are individually well documented. It is their combination in one formally specified, portable, battery-powered instrument spanning five functions, together with an evidence-labelled evaluation, which the related work surveyed here does not present as a whole.

3. HARDWARE ARCHITECTURE

3.1 System Overview

The hardware is partitioned into three functional tiers: power management, processing core, and input/output interface (Fig. 1).

3.2 Processing Core: ESP32 DevKit V4 (WROOM-32)

The ESP32 WROOM-32 module carries the ESP32-D0WDQ6 chip with dual Xtensa LX6 32-bit cores, 520 KB SRAM, 448 KB ROM and 4 MB SPI flash [1]. It runs at 3.3 V from the DevKit's onboard LDO regulator (input tolerant to 5 V), and since the SSD1306 OLED also runs at 3.3 V, one regulated rail powers both processor and display. The `millis()` calls are backed by the ESP32 hardware timer at 1 μs resolution [6].

3.3 Display: SSD1306 OLED (128×64)

The SSD1306 sits on the I2C bus at address `0x3C`, with SDA on GPIO 21 and SCL on GPIO 22. Because of the way the phosphor stripes are deposited on this panel, the top 16 pixel rows glow yellow while the lower 48 rows glow blue [7]. The firmware reserves the yellow strip for mode titles and uses the blue area for live data. All graphics are rendered through the Adafruit SSD1306 and Adafruit GFX libraries [8], which provide the I2C driver and the primitive drawing routines (text, lines, fills) used throughout the firmware's `draw*()` functions.

Table 2. Estimated Current Consumption by Subsystem (Datasheet-Based, Not Measured)

Subsystem	Current (mA)
ESP32 chip, 240 MHz, radio off	30–68
SSD1306 OLED module	2–25
DevKit regulator, USB-UART	∼5
LED feedback (0 average, flash)	0–6.4
Total (average)	**∼37–104**

3.4 Input Interface: Push Buttons

All three momentary push buttons are wired active-low against internal pull-up resistors: BTN1 on GPIO 13 (select/back), BTN2 on GPIO 25 (up/action A) and BTN3 on GPIO 14 (down/action B). The choice was deliberate: the ESP32 boot ROM samples GPIO 0, 2, 12 and 15 at power-up to select flash mode and IO voltage, so routing buttons through those pins risks corrupting the boot sequence if a button is depressed or floating at power-on. Non-strapping GPIOs eliminate that hazard.

3.5 LED Feedback

A red LED in series with a 220 Ω current-limiting resistor on GPIO 32 gives the user immediate physical confirmation of every registered button press. Rather than halting execution with a `delay()` call, the firmware schedules the LED off-time via a stored timestamp (`ledOffAt`), keeping the main loop responsive throughout the flash period.

3.6 Power Management

1) Energy Source and Charging: The primary energy store is a 3.7 V 500 mAh LiPo cell. The TP4056 linear charger implements a two-phase CC/CV profile: constant current at approximately 1C until 4.2 V, then a tapering-current hold until the termination threshold [9]. Power arrives via Micro-USB. All power conversion for the logic is routed through the DevKit's onboard LDO, which keeps the logic supply independent of the cell's discharge curve.
2) Power Budget: The estimated average draw at full operation is about 37–104 mA from the cell (Table 2). The ESP32 range is the datasheet figure for the CPU running at 240 MHz on both cores with the radio off [1]; the OLED range runs from a mostly dark text screen to all pixels lit, as typically specified for SSD1306 modules; the LED current is calculated as $(3.3 - 1.9)/220 \approx 6.4$ mA assuming a 1.9 V forward drop and flows only during a flash; and the DevKit regulator and USB-UART overhead is an assumed typical value. The TP4056 module is assumed to draw negligible current from the cell outside charging and is not included. The wide range reflects the spread of the datasheet figures and can be narrowed only by measurement. For a 500 mAh cell it corresponds to roughly 4.8–13.5 hours at 100% usable capacity and 3.8–10.8 hours at 80% (Section 6.7). This is a datasheet-based estimate, not a bench measurement of the assembled device.
3) Bill of Materials: Cross-checked against listings from Indian retailers (Robu.in, Robokits.co.in, Robocraze) as of September 2026, a single unit costs roughly INR 710–1055 (ESP32 DevKit 280–450, OLED 100–150, TP4056 module 15–30, LiPo cell 250–300, buttons, LED and wiring 65–125). Prices vary by seller and are representative.

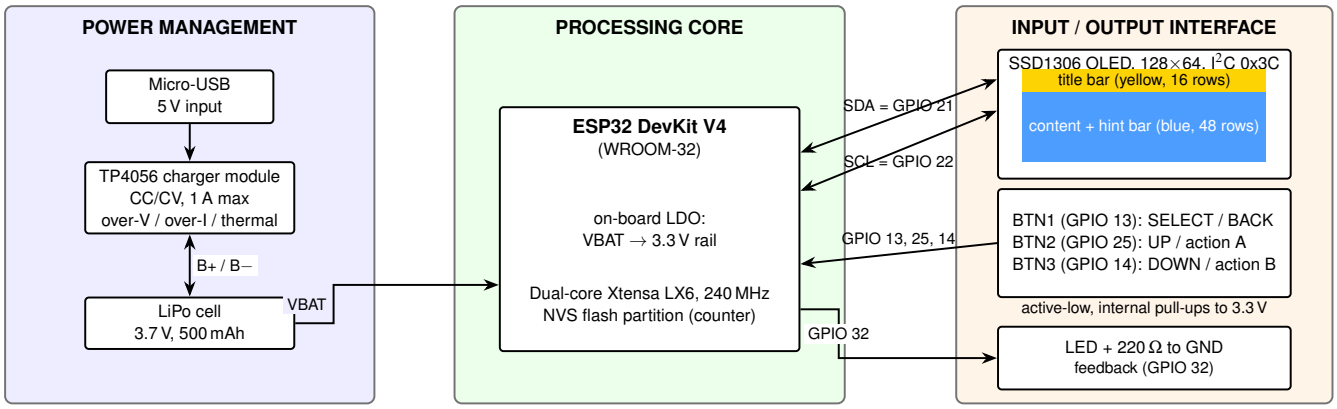


Fig. 1. Hardware functional block diagram of the ESP32 Multi-Tool. The power management, processing core and I/O interface tiers are delineated.

4. SOFTWARE DESIGN AND FSM ARCHITECTURE

4.1 Application State Model

The firmware is organised as a Moore finite state machine, meaning that every output is determined solely by the current state rather than by any combination of state and input. Each state is solely responsible for drawing its own complete screen, and no output is produced as a side effect of a transition.

Six application states are defined as the enumeration `enum AppState {ST_MENU, ST_CLOCK, ST_STOPWATCH, ST_TIMER, ST_COUNTER, ST_ABOUT}`, with `state` initialised to `ST_MENU` at boot. The state variable is modified exclusively inside button handler functions; draw functions never touch it. Rendering is therefore idempotent: repeated calls of a draw function in the same state produce identical output.

4.2 Formal State Definition

To make the claim of a formally specified design concrete, the navigation layer is defined as a Moore machine

$$M = (Q, \Sigma, \Lambda, \delta, \lambda, q_0)$$

where $Q = \{q_{menu}, q_{clock}, q_{sw}, q_{tmr}, q_{cnt}, q_{abt}\}$ maps directly to the six `AppState` enumerators; $\Sigma = \{b_1^s, b_2^s, b_3^s, b_1^\ell, b_2^\ell, b_3^\ell\}$ is the input alphabet of short (s) and long (ℓ) press events per button; Λ is the set of six screens rendered by `drawMenu()` through `drawAbout()`; $\delta : Q \times \Sigma \to Q$ is realised by the `handle*()` functions (Table 3); $\lambda : Q \to \Lambda$ is realised by the `draw*()` functions and depends only on Q, never on Σ, which is what makes the machine Moore rather than Mealy; and $q_0 = q_{menu}$ is the initial state at boot.

Table 3 lists the non-trivial transitions of δ. Any input not listed leaves the machine in its current state (a self-loop), which is the default behaviour of each `handle*()` function for an event it does not act on.

The practical effect of Table 3 is a *hub-and-spoke* topology: q_{menu} is the only state with more than one outgoing edge to a distinct destination, and every operational state's exit is back to q_{menu}, subject to the timer interlock described in Section 4.9. This is the topology walked by the firmware (Fig. 2) and is the one depicted in Fig. 2(a).

Table 3. Non-Trivial Transitions of $\delta : Q \times \Sigma \to Q$

Current State	Input	Next State
q_{menu}	b_2^s (Up)	q_{menu} (cursor moves)
q_{menu}	b_3^s (Down)	q_{menu} (cursor moves)
q_{menu}	b_1^s (Select)	q_{clock}, q_{sw}, q_{tmr}, q_{cnt} or q_{abt}
q_{clock}, q_{sw}, q_{tmr}, q_{cnt}, q_{abt}	b_1^s (Back)	q_{menu}
q_{tmr}	b_2^ℓ	q_{tmr} (countdown starts)

4.3 FSM Transition Structure and Main Loop

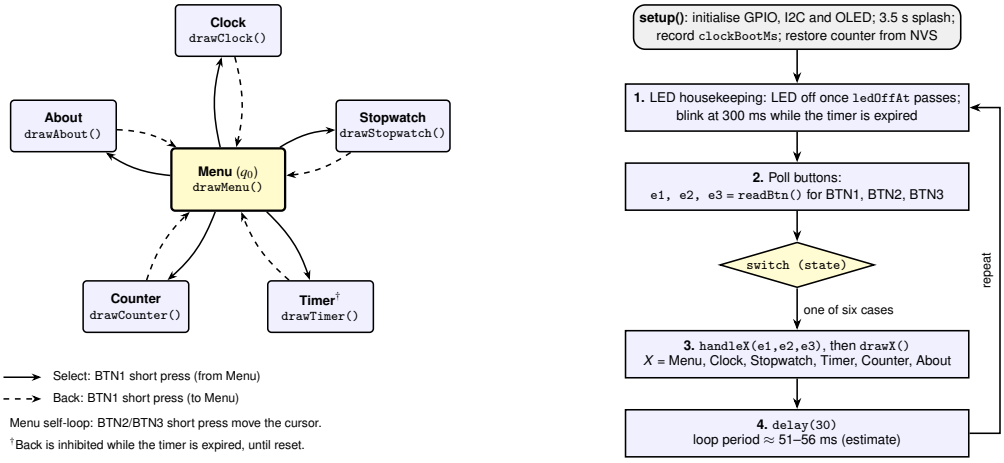
Fig. 2(a) shows the complete state transition diagram. `ST_MENU` acts as the central hub from which every operational state can be reached, and a short press of BTN1 returns the machine to `ST_MENU`. Fig. 2(b) shows the program flow of `loop()`: LED housekeeping, button polling, dispatch on the current state to the matching `handle*()` and `draw*()` pair, and a 30 ms delay.

The dispatch separates transition logic (inside the `handle*()` functions) from output generation (inside the `draw*()` functions): each of the six cases of the `switch (state)` statement calls its `handle*(e1,e2,e3)` and then its `draw*()`, followed by `delay(30)`. The 30 ms delay, together with the display transfer, sets the loop period analysed in Section 5.1 (about 18–20 Hz refresh), which suffices for text-based instrumentation whose content changes slowly.

4.4 Button Input Handling and Debounce

When a mechanical button is pressed, the contacts do not settle cleanly: they bounce between open and closed in the first few milliseconds, potentially registering phantom edges. This design debounces entirely in software, avoiding RC filters or debounce ICs. Each GPIO is polled once per loop pass, and a short press is committed only if the measured press-to-release interval reaches a configured minimum. The same framework detects long-press events without additional circuitry.

```
// On edge: falling saves pressedAt, clears
// longFired; rising checks DEBOUNCE_MS and
// (if not already long-fired) sets shortPress.
if (raw != b.lastRaw) { b.lastRaw = raw;
  if (!raw) { b.pressedAt=millis(); b.longFired=false; }
  else if (!b.longFired && (millis()-b.pressedAt)
           >= DEBOUNCE_MS) ev.shortPress = true;
}
// Independently, a held-LOW pin past
// LONGPRESS_MS latches one long-press event.
```



(a) (b)

Fig. 2. (a) Hub-and-spoke Moore machine FSM. Menu is the sole hub; every operational state is reached only through the menu and returns only to the menu. Outputs depend solely on the current state. (b) Program flow of `loop()` in the firmware.

```
if (!raw && !b.longFired &&
    (millis()-b.pressedAt) >= LONGPRESS_MS)
  { b.longFired = true; ev.longPress = true; }
```

Listing 1: Core of the software debounce and long-press logic (function signature and event-structure plumbing omitted for space).

A falling edge saves the timestamp and clears `longFired`; a rising edge returns a short-press event if the press-release interval meets `DEBOUNCE_MS` (50 ms) and no long press has fired. A pin held LOW past `LONGPRESS_MS` (700 ms) fires one long-press event, and `longFired` latches to block repeats.

4.5 OLED Layout Architecture

The layout takes advantage of the yellow-blue colour split (Fig. 3(a)). The 16-pixel yellow strip at the top is a persistent title bar, fixed regardless of mode. Within the 48-pixel blue area below, the bottom 16 rows (48–63) are set aside for a contextual hint bar, separated from the main content by a horizontal line at row 47, with hint text at $y = 50$ and $y = 57$. The Stopwatch, Timer and Counter screens use all three zones; the Menu and About screens use the whole blue area for content, and the Clock screen draws its own rule at row 37 and a status line at row 52. The zones are fixed by four constants: `YELLOW_H = 16` (title bar height), `HINT_SEP = 47` (separator row), and `HINT_Y1 = 50`, `HINT_Y2 = 57` (the two hint text rows).
Every title string is centred horizontally by first calling `getTextBounds()` to measure its rendered pixel width, then positioning the cursor accordingly, which keeps alignment tidy for mode names of different lengths.

4.6 Menu Module

The menu shows up to four entries 12 pixels apart, and a scroll guard adjusts `menuScroll` whenever `menuSel` leaves the visible window. The selected entry is drawn as white fill with black text. BTN2 moves the cursor up, BTN3 moves it down (both wrapping around the five-item list), and a short press of BTN1 confirms the selection.

4.7 Clock Module

The clock uses a base-offset model built on `millis()` and needs no network. At startup, `clockBootMs` records `millis()` and `clockBaseS` is set to zero, so the display begins counting from 00:00:00. On each refresh, the time to display is

$$t_d = \left(\left\lfloor \frac{millis() - clockBootMs}{1000} \right\rfloor + clockBaseS \right) \mod 86400. \quad (1)$$

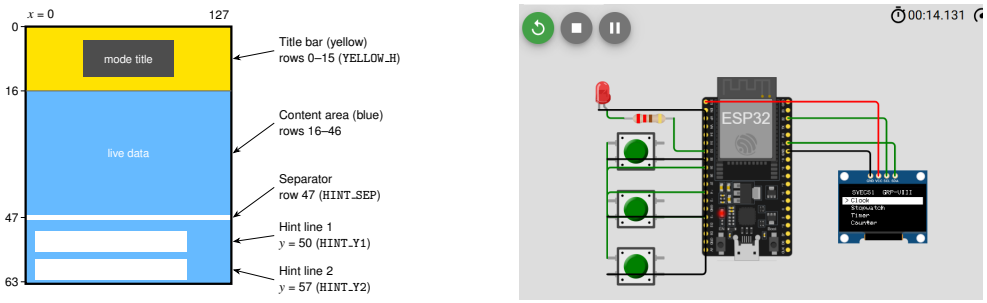
When the user switches time zones, `clockBaseS` is adjusted by the difference in UTC offsets $\Delta_{off} = newOffset - oldOffset$ (seconds) via $clockBaseS = (clockBaseS + \Delta_{off} + 86400) \bmod 86400$; the pre-added 86400 guards against a negative result in either direction. Eight time zones are available (IST, UTC, EST, EDT, PST, GST, JST, AEST), and BTN2 toggles the 12-/24-hour display.

4.8 Stopwatch Module

1) Timekeeping Model: Loop iterations do not take a fixed time, so adding a fixed increment to an accumulator once per loop would drift. The stopwatch instead uses a reference model built around two variables: `swBase`, the `millis()` snapshot taken when the stopwatch was most recently started or resumed, and `swAccum`, the total milliseconds accumulated across completed running intervals. The elapsed time on the display is

$$T_{SW} = swAccum + \begin{cases} millis() - swBase & \text{if running} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Because this expression computes elapsed time between a stored reference and the current call, it does not depend on the loop period.
2) State Transitions: The stopwatch moves among three internal sub-states: READY, RUNNING and PAUSED. On the RUNNING→PAUSED transition, triggered by a short press of BTN3, `swAccum` is increased by `millis() - swBase`, `swRunning` clears and `swPaused` sets. On resume, `swBase` is refreshed to the current `millis()` value, starting a fresh interval.



(a) (b)

Fig. 3. (a) OLED screen layout (128×64 pixels); the vertical scale is stretched for legibility, so this is a schematic of the hint-bar layout (Stopwatch, Timer, Counter) rather than a pixel-accurate rendering. (b) Wokwi simulation circuit with the firmware showing the main menu; this is a simulator screenshot, not a photograph of the physical prototype.

No interval is counted twice, and the only error added per pause-resume cycle is millisecond quantisation (Section 6.4).

3) Centisecond Formatting: Elapsed time is shown as H:MM:SS.cs, with the centisecond field obtained by integer division of the 0–999 ms remainder by ten (`cs = (ms%1000)/10`). This is a *display-resolution* property of the formatting routine, not a claim of ±10 ms *measurement accuracy*. The underlying elapsed-time value does not depend on loop-period variability by construction (Eq. 2); its accuracy is bounded by the timer resolution and crystal tolerance of the ESP32 and by how promptly a state change is polled (Section 5.1).

4.9 Timer Module

1) Countdown Model: The timer shares the stopwatch's reference-timestamp strategy but counts down, using four variables: `tmSetMs`, the selected duration in milliseconds; `tmAccumElapsed`, the milliseconds consumed across earlier running intervals; `tmBase`, the `millis()` reference of the current interval; and `tmRemMs`, the remaining time recalculated on every loop pass in the Timer state as `tmRemMs = (long)tmSetMs - (long)(tmAccumElapsed + millis() - tmBase)`. Once this value reaches zero or below, `tmRunning` clears and `tmExpired` is set. The signed cast matters: if the loop runs late on the expiry cycle, unsigned arithmetic would wrap a small negative remainder to a very large positive number.

2) Time Setting and Start: BTN2 and BTN3 decrement and increment the duration in 6-second steps. While running, BTN2 pauses and BTN3 resets. Starting the countdown requires a long press of BTN2 (with a non-zero duration), which resets `tmAccumElapsed`, takes a fresh `tmBase` snapshot and sets `tmRunning`, and which prevents an unintentional start while the duration is still being set.

3) Expiry Behaviour and Safety Interlock: Once the timer reaches zero, the LED blinks at 300 ms intervals using a non-blocking computation against `millis()`, namely `digitalWrite(LED, (millis()/300)%2)`, evaluated once per loop pass. While the expiry state is active, the handler treats a short press of any button as a reset and then returns early. BTN1 therefore cannot navigate to the menu until the alarm has been cleared, an intentional interlock that prevents a finished timer from being left unnoticed.

4.10 Counter Module

The counter stores a signed 32-bit integer. Each modification is written through to NVS immediately: a short press of BTN3 increments the counter and writes it via `prefs.putLong("count", counter)`; a short press of BTN2 decrements it in the same way; and a long press of BTN1 resets it to zero, also written at once. Every change is therefore persisted before the next screen refresh, with no separate save step.

The Preferences library writes to the ESP32's NVS flash partition, which spreads writes across pages (wear levelling) and separately uses redundant entries to guard against unexpected power loss [10]; these are distinct mechanisms. On startup, the firmware opens the namespace with `prefs.begin("app", false)` and checks `prefs.isKey("count")`. If the key exists, `counter` is restored via `prefs.getLong("count", 0)`; otherwise it is initialised to zero and written back. The `isKey()` check distinguishes a legitimately stored zero from the absence of any stored value, so a counter sitting at zero is not mistaken for an uninitialised one on the next boot.

5. TIMING ANALYSIS

5.1 Loop Cadence and Display Latency

The software delay at the end of each main-loop iteration is 30 ms, but the loop period is longer because the I2C display transfer is not instantaneous. At 400 kHz Fast Mode, shifting the 1024-byte framebuffer takes at least $1024 \times 8/400{,}000 \approx 20.5$ ms. This figure is a lower bound: it ignores the acknowledge bit that follows every byte (nine clock cycles per byte, about 23.0 ms in total) and the addressing and control bytes of each transaction, so the actual transfer is probably 23–26 ms. The loop period is therefore approximately 51 ms at the lower bound and probably 53–56 ms in practice, giving a refresh rate of about 18–20 Hz. These are analytical estimates, not measurements. The simulations in Section 6 use 51 ms; their conclusions are not sensitive to this choice because the loop-dependent quantities scale with it (Section 6.5), and loop-period variability does not enter the stopwatch or timer values.

5.2 Debounce Window Selection

A 50 ms debounce threshold was chosen as a conservative margin against contact bounce, which is widely reported to last a few to a few tens of milliseconds for tactile switches. The bounce of the switches used here has not been characterised. The 50 ms threshold is almost equal to the polling period, so the sampling itself performs much of the filtering (Section 6.6).

Table 4. Wokwi Functional Walkthrough (Simulation, Qualitative)

Check	Observed	Result
Menu, Clock (12/24-hour, time zone IST to UTC, PST shown), About	As expected	Pass
Stopwatch start, pause, resume, reset	As expected	Pass
Timer set to 12 s, countdown, expiry	DONE, LED lit	Pass
Counter increment to 5, decrement, reset	As expected	Pass
Counter value after simulator restart	Returned to 0	Not verified

5.3 Timer Expiry Precision

The expiry condition is evaluated once per loop pass, so the detection latency is at most one loop period (roughly 51–56 ms), which is acceptable for process-timing and kitchen-timer use.

6. SIMULATION-BASED EVALUATION

This section evaluates the design with four kinds of evidence, and every result is labelled with its type: (i) the *compiler report* of the Arduino IDE, (ii) a functional walkthrough in the *Wokwi simulator*, (iii) an *algorithm-level simulation, used for illustration only,* in which the firmware logic is driven by synthetic inputs, and (iv) *analysis* and *datasheet-based estimates*. **None of these is a bench measurement of the physical device.** No reference clock, logic analyser or current meter was used.

6.1 Compilation Results

The firmware (`esp32_multitool_v6.ino`) was compiled with Arduino IDE 2.3.10 (Arduino CLI 1.5.1) [11], ESP32 Arduino core 3.3.12 [12], Adafruit SSD1306 2.5.17 and Adafruit GFX 1.13 [8], for the ESP32 Dev Module board. The build succeeded, with 338,592 bytes of program storage against a limit of 1,310,720 bytes (25.83%) and 23,836 bytes of global variables against 327,680 bytes (7.27%). The storage limit is the build's application limit, not the nominal flash capacity, and the global-variable figure is a static allocation, not peak heap or stack use.

6.2 Wokwi Functional Walkthrough

The firmware was loaded into a Wokwi ESP32 project [13] with an ESP32 board, an SSD1306 OLED, three push buttons and an LED with a 220 Ω resistor, wired as in Section 3 (Fig. 3(b)). The walkthrough entered each state and exercised the visible controls. Table 4 groups the 14 checks by function. Thirteen passed, three of them (stopwatch resume, stopwatch reset, and counter decrement and reset) confirmed by the students' observation without a retained screenshot. The remaining check, counter persistence across a simulator restart, was *not demonstrated*: the count returned to zero. This does not show that persistence fails on a physical ESP32, since simulator storage may differ from real flash, but persistence is not claimed here as verified.

6.3 Simulation Assumptions

The algorithm-level simulation ports the firmware logic (`readBtn()`, the reference-timestamp timekeeping and the timer expiry test) and drives it with synthetic inputs. The assumptions are:

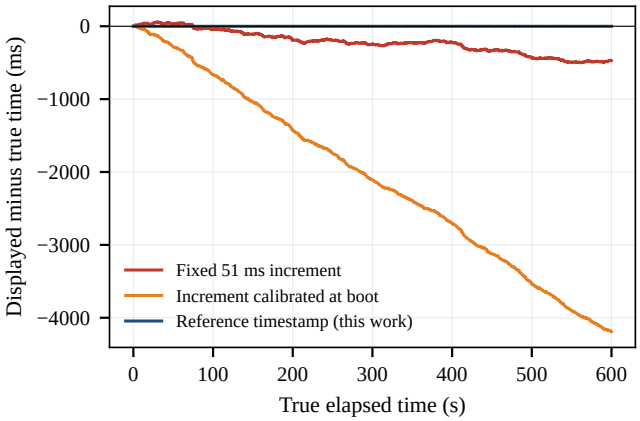


Fig. 4. Illustrative simulation (not a hardware measurement): displayed minus true time for three timekeeping schemes over one simulated 10-minute run.

—**Loop model:** nominal period 51 ms (30 ms delay plus about 21 ms display transfer, the lower end of Section 5.1), 1 ms Gaussian jitter and a 0.5% chance per iteration of a 5–20 ms stall; `millis()` is an ideal 1 ms floor of true time, so crystal tolerance is not modelled.

—**Bounce model:** edge dwell times are exponential with a 0.4 ms mean inside a 10 ms window ("typical") or uniform over 1–60 ms ("severe"); these are assumptions, not measured switch behaviour.

—**Sample sizes:** 200 ten-minute drift runs; 1,500 (firmware polling) or 150 (1 ms polling) trials per setting for the button studies.

Because all inputs are synthetic and the timebase is ideal, the results characterise the *algorithms* under stated assumptions, not the device.

6.4 Timekeeping Error

Analysis. Because Eq. 2 reads `millis()` at the start and at the current time, the displayed value differs from the true elapsed time only by the 1 ms quantisation of `millis()` at each timestamp, independent of how long any loop iteration takes (crystal tolerance, which is not analysed here, adds to this on real hardware). A scheme that adds a fixed increment per iteration instead drifts linearly: if the real period differs from the assumed increment by a fraction ϵ, the error after time t is ϵt, so a 0.1% mismatch gives about 0.6 s over 10 minutes.
Illustrative simulation. Fig. 4 shows one simulated 10-minute run under the loop model of Section 6.3. The fixed 51 ms increment drifts by a few hundred milliseconds to about a second (median 715 ms over 200 runs), and an increment calibrated from the first 20 loop periods at boot drifts by several seconds (median 1.76 s). These magnitudes depend on the assumed jitter and stall rate and should be read as an illustration of the mechanism, not as predictions for the device.

6.5 Latency

This subsection is analytical. The timer expiry is noticed on the first loop pass after the true expiry, so the detection latency lies between 0 and one loop period, with a mean of about half a period: roughly 26–28 ms for the estimated 51–56 ms period. A button release is

seen at the next poll and the screen is updated in the same pass, so the delay from release to visible update is at most about one loop period plus the display transfer time, roughly 75–80 ms in the worst case. Neither figure has been measured.

6.6 Button Registration and Debounce

Analysis. The firmware polls each button once per loop pass, so a press shorter than the loop period T is seen only if it spans a poll, which has probability of roughly P/T for a press of duration $P < T$; a press of at least T is always sampled. Registration then requires the measured interval to reach the 50 ms debounce threshold. The 50 ms threshold is almost equal to the 51–56 ms polling period, so every press seen at two consecutive polls already satisfies it. The threshold is therefore rarely decisive at this polling rate, and the filtering arises mostly from sampling.
Illustrative simulation. Fig. 5(a) shows the simulated fraction of presses registered exactly once. At the firmware polling rate, presses of 100 ms or longer were registered in at least 99.5% of trials, 60 ms presses in 93.8% and 30 ms presses in 52.7%, broadly consistent with, and slightly below, the analytical expectation because of loop jitter and the 50 ms threshold. No multiple-event errors occurred under the "typical" bounce model. Polled at 1 ms (dashed), the same algorithm rejects presses shorter than 50 ms explicitly.
Fig. 5(b) applies invented chatter of up to a stated maximum duration at each edge to 200 ms presses. For chatter of up to 40 ms per edge, the firmware algorithm registered exactly one event per press, whereas naive edge counting at 1 ms polling already gave about 1.6 events per press. The algorithm is not immune to arbitrary chatter: because it debounces at the release edge only, chatter longer than the polling period can produce extra events (about 1.5 per press at 150 ms in this model). The chatter model is an assumption, and the real bounce of the switches used remains to be measured.

6.7 Power and Runtime

The power results are an analysis of datasheet-based figures, not a circuit simulation. Summing the component currents of Table 2 gives about 37–104 mA. Fig. 6 converts this to runtime for a 500 mAh cell: 4.8–13.5 hours at 100% usable capacity, 4.3–12.2 hours at 90% and 3.8–10.8 hours at 80%. The upper current bound is deliberately pessimistic, so about 4–5 hours is the prudent planning figure. The usable fraction also depends on the cut-off voltage and the dropout of the DevKit's linear regulator, which has not been checked.

6.8 Summary of Results

Table 5 collects the results with the basis of each; simulated values are illustrations under the assumptions of Section 6.3.

7. ADVANTAGES AND LIMITATIONS

7.1 Advantages

Separating state logic from rendering keeps each mode self-contained and independently modifiable, the reference-timestamp model tolerates added loop overhead without drift, and non-strapping GPIO selection prevents hardware-state-dependent boot faults.

7.2 Limitations

The most important limitation is that no result in this paper is a physical measurement: the simulated timing uses an assumed loop model and an ideal timebase, the bounce model is synthetic, the Wokwi walkthrough is qualitative, the current and runtime figures come from datasheets, and counter persistence was not demonstrated in the simulator. Hardware validation should compare stopwatch and timer values with a reference clock, measure loop period and I2C transfer time with a logic analyser, characterise switch bounce, log the current draw, check the 3.3 V rail at low cell voltage, and repeat power-cycle tests of the counter. Design limitations also remain: there is no hardware RTC, so the clock restarts at 00:00:00 on every power-on; there is no audible output; the debounce filters at the release edge only (Section 6.6); and there is no battery monitoring or low-charge warning.

8. CONCLUSION

This paper described a battery-powered digital multi-tool built around the ESP32 WROOM-32, with five utility modes under a formally specified Moore-machine FSM. The key firmware contributions are a software-only debounce and long-press algorithm, reference-timestamp timekeeping for the stopwatch and timer, NVS write-through persistence for the counter, and a three-zone OLED layout that uses the panel's yellow-blue split.
The evaluation is entirely compilation-, simulation- and analysis-based: the firmware uses 25.83% of the reported program storage and 7.27% of the reported dynamic memory, and the Wokwi walkthrough passed 13 of 14 checks, with counter persistence not demonstrated. Analysis shows that the reference-timestamp scheme does not accumulate loop-dependent drift, that timer expiry is detected within one loop period and that presses shorter than that period can be missed, and an illustrative simulation supports these points. The datasheet-based current estimate of 37–104 mA suggests roughly 4–13 hours of untethered operation, with about 4–5 hours as a prudent planning figure. These are not bench measurements, and validating them on the physical device is the immediate next step. A DS3231 RTC, a piezo buzzer and an ADC battery monitor could be added without changes to the FSM core.

ACKNOWLEDGMENTS

The authors thank the Department of Electronics and Computer Science, SAKEC, for its support.

9. REFERENCES

[1] Espressif Systems. ESP32 Series Datasheet, v3.7. Shanghai, China, 2023. Available: espressif.com.

[2] J. Beningo. *Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques*. Newnes, Rockland, MA, USA, 2015.

[3] J. Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. Newnes, Oxford, U.K., 3rd edition, 2013.

[4] N. Kolban. Kolban's Book on ESP32. Self-published, 2018. Available: leanpub.com/kolban-ESP32.

[5] L. A. Ajao, M. A. Adegboye, E. M. Dogo, S. O. Aliyu, and D. Maliki. Development and implementation of microcontroller-based improved digital timer and alarm system. In *Proc. Int. Conf. Information and Communication*

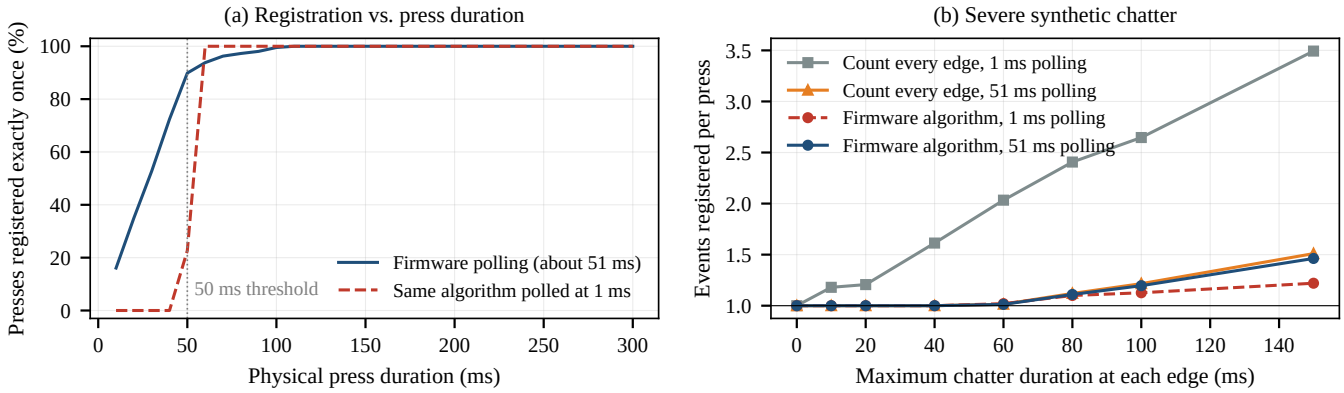


Fig. 5. Simulation results (not hardware measurements): (a) fraction of presses registered exactly once versus press duration; (b) events registered per press under severe synthetic chatter, for the firmware algorithm and for naive edge counting.

Table 5. Summary of Evaluation Results and Their Basis

Item	Result	Basis
Program storage; static RAM	338,592 B (25.83%); 23,836 B (7.27%)	Compiler report
Functional states	13 of 14 checks passed; persistence not demonstrated	Wokwi simulation
Reference-timestamp display error	At most the 1 ms `millis()` quantisation, independent of loop period	Analysis
Fixed-increment drift over 10 min	Linear, ϵt; simulated medians 0.72 s (fixed 51 ms) and 1.76 s (boot-calibrated)	Analysis, illustrative simulation
Timer expiry detection latency	At most one loop period (51–56 ms); mean about 26–28 ms	Analysis
Registration of short presses	Press shorter than the loop period missed with probability about $1 - P/T$; simulated 99.5% / 93.8% / 52.7% for 100 / 60 / 30 ms	Analysis, illustrative simulation
Invented chatter up to 40 ms per edge	Exactly one event per press (naive counting about 1.6)	Illustrative simulation
Average current draw	About 37–104 mA	Datasheet-based estimate
Runtime, 500 mAh cell	4.8–13.5 h (100%); 3.8–10.8 h (80%)	Analysis

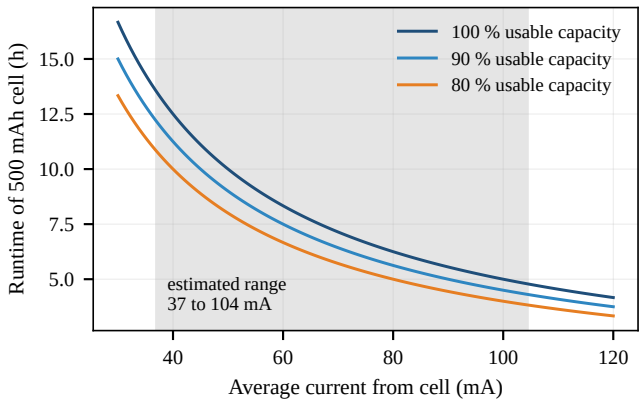


Fig. 6. Datasheet-based analysis (not a hardware measurement): estimated runtime of the 500 mAh cell versus average current for three usable-capacity assumptions; the shaded band is the estimated 37–104 mA range.

Technology and Its Applications (ICTA), pages 184–190, Minna, Nigeria, November 2016.

[6] Espressif Systems. ESP-IDF Programming Guide: General Purpose Timer, v5.1. Shanghai, China, 2023. Available: docs.espressif.com.

[7] Solomon Systech. SSD1306 Advanced Information: 128×64 Dot Matrix OLED Controller, Rev. 1.1. Hong Kong, 2008.

[8] Adafruit Industries. Adafruit SSD1306 and Adafruit GFX libraries. GitHub, 2023. Available: github.com/adafruit.

[9] NanJing Top Power ASIC Corp. TP4056 1A Standalone Linear Li-Ion Battery Charger Datasheet, Rev. 1.2, 2012.

[10] Espressif Systems. Non-Volatile Storage (NVS) Library, ESP-IDF v5.1. Shanghai, China, 2023. Available: docs.espressif.com.

[11] Arduino. Arduino IDE 2 and Arduino CLI, software and documentation, 2026. Available: arduino.cc (accessed Nov. 2026).

[12] Espressif Systems. Arduino core for the ESP32, v3.3.12. GitHub, 2026. Available: github.com/espressif/arduino-esp32.

[13] Wokwi. Wokwi online electronics simulator (ESP32), 2026. Available: wokwi.com (accessed Nov. 2026).