ChatGPT

Development Brief: ESP32 Industrial Temperature Controller Rewrite

Overview and Objectives

This project involves rewriting an existing ESP32-based industrial temperature controller system to improve its alarm management, user interface, and overall maintainability. The **core functionality and hardware setup** (sensor monitoring, pin assignments, library usage, class structure, etc.) should remain the same as the original system for continuity. The focus is on restructuring the alarm handling, enhancing the alarm indication (siren/beacon relays and LEDs), upgrading the web UI (with Russian localization), refining display and button behavior, fixing Modbus issues, and reorganizing the code for clarity and documentation. The end result will be a cleaner, more configurable system with the same capabilities, plus additional features as specified.

Alarm Handling Restructuring

Define All Alarms at Initialization: Instead of dynamically creating alarm objects when a condition occurs, pre-define all potential alarms during system startup. For each sensor/measurement point and each alarm type (e.g. high temp, low temp, sensor error, etc.), instantiate an `Alarm` object and store it (for example, in a list or map in the `TemperatureController`). This ensures a known set of alarms that can be managed uniformly (enable/disable, configure priorities) from the start. Using a centralized initialization (possibly reading from a config file or a predefined list), create alarms with default priority and threshold values. This may involve refactoring the current `createAlarm` logic to simply mark an existing alarm as active, rather than creating new objects on the fly.

Enable/Disable Alarms per Configuration: Introduce an **enabled/disabled flag** for each alarm. An alarm that is disabled will not trigger any actions or notifications even if its condition is met. The enable/disable status can be determined from configuration – for example, certain alarm types or priorities might be globally disabled, or specific point alarms can be toggled off. Ensure the `Alarm` class has a property for this (if not already present) and check this flag in alarm evaluation logic (i.e., skip or ignore alarms that are not enabled).

Priority Configuration and Management: Allow the priority of each alarm to be configured through multiple interfaces: - **Configuration File:** The system already supports reading alarm configurations (including priority) from a config file (e.g. a CSV or YAML) [1]. Leverage this to set each alarm's priority and enabled flag at startup. For example, the config might map each alarm (by sensor ID and type) to a priority level (Low, Medium, High, Critical) and an initial enabled state. - **Web Interface:** Provide a web page (or extend an existing settings page) to display all alarms in a table, with dropdowns or selectors for priority and toggles for enable/disable. This allows users to easily change an alarm's priority or turn it on/off. Changes made in the web UI should update the running configuration (and ideally persist to the config file). - **Modbus RTU:** Allocate Modbus registers to allow an external system (via Modbus) to read and write the priority and enable status of each alarm. We will use **one byte per alarm** to encode this information. For example, use the high bit (bit 7) as the **enable flag** (1 = enabled, 0 = disabled) and the lower bits (e.g. bits 0–2 or 0–3) to encode the priority level (e.g. 0=Low, 1=Medium, 2=High, 3=Critical). This encoding fits in a single byte and can be expanded if needed (since 4 priority levels require 2 bits, leaving additional bits for future use). Each alarm's config byte can be exposed as a holding register in the Modbus map. The firmware should interpret writes to these registers by updating the corresponding alarm's settings (and possibly saving to config), and provide reads of these registers reflecting the current settings. - **Consistent Priority Definition:** Use a clear enum or mapping for priorities (e.g., 0=LOW, 1=MEDIUM, 2=HIGH, 3=CRITICAL) across the system for consistency [2] [3]. The web UI and config file should use human-readable labels (Low, Medium, High, Critical) which get translated to the numeric/enum values in the code.

Alarm Evaluation Logic: Modify the alarm-checking routine to use the initialized alarms rather than creating new ones. Each alarm object should know its sensor (or source) and threshold/condition. On each cycle of sensor readings: - If an alarm's condition is met (e.g. temperature >= high threshold) and the alarm is enabled, mark or update that alarm as **active** (and set its stage to "ACTIVE" if it was "NORMAL" before). - If an alarm condition clears (goes back to normal range) and the alarm was active, update its status (perhaps mark as **cleared** and resolved). - Ensure that *no new Alarm objects are created or destroyed during runtime* for threshold conditions; instead, just change the state of existing alarm objects. This makes alarm management (sorting, acknowledging, etc.) more straightforward. - Maintain sorting of active alarms by priority and timestamp. The highest priority active alarm should be readily identifiable (e.g., via a method `getHighestPriorityAlarm()`).

Acknowledgment and Auto-Reactivation: Implement the ability to acknowledge alarms and potentially auto-reactivate them if they remain unresolved: - When the user acknowledges an alarm (through the web UI or a physical button), change its stage to "ACKNOWLEDGED" (but it still remains active in memory until the condition clears). Acknowledging means the alarm's audible alert (siren) can be muted, but visual indicators might remain. - Provide a mechanism (configurable timer) to **re-escalate an alarm** that remains in acknowledged state for too long. For example, if a critical alarm is acknowledged but the temperature is still out of range after a certain period, the system can treat it as active again (unacknowledged) to ensure it gets attention. This addresses the requirement to "move acknowledged alarm to active by timer" [4]. Make this timeout value configurable per priority (e.g., critical alarms might re-alert after 5 minutes if still active). - Only allow reactivation if the alarm is still actually in alarm condition (if the condition cleared, the alarm should go to normal/resolved state instead).

In summary, the alarm subsystem will be robust and configurable: all alarms defined upfront, each with an enable flag and priority that can be set via config, web UI, or Modbus, and with proper handling of activation, acknowledgment, and possible re-alert. This addresses the need for a priority-based alarm system and lays the groundwork for the indication logic described next.

Alarm Priority Indication and Relay Control

The device uses **three relay outputs** (assume Relay1, Relay2, Relay3 as per original code) for alarm indication: one for a siren (audible alarm), one for a beacon light (visual strobe), and a spare. The behavior of these outputs must follow the alarm priority logic and be flexible/configurable:

- **Critical Alarms (Highest Priority):** If there is any active *unacknowledged* critical alarm, trigger **both siren and beacon continuously** (siren on, beacon on solid). This is the most urgent state. According to the requirements, a critical alarm that has not been acknowledged should activate both the audible and visual alarms without interruption [5].
- **High Priority Alarms:** If there are no unacknowledged critical alarms, but there is an active *unacknowledged* high-priority alarm (or a critical alarm that is active **but has been acknowledged**), then activate the **beacon continuously, but not the siren** [5]. In other words,

for high-priority unacknowledged alarms (or acknowledged critical alarms that are still active), the beacon stays on as a warning, but the siren remains off (to avoid constant noise for acknowledged critical alarms or high alarms).

- **Medium and Low Priority Alarms:** If all critical alarms are resolved (or none exist) and all high priority alarms are either resolved or acknowledged, but there are active medium-priority alarms (active or even acknowledged), then use a **pulsing beacon** as a lower-level alert [6]. For example, the beacon could flash on for 10 seconds and off for a few minutes in a slow cycle (these timing values should be configurable). This intermittent blinking indicates a non-critical situation that still requires attention. Low priority alarms, if any, could be treated similarly or even with a slower blink or just a log entry depending on requirements. Essentially, any remaining alarms of lower priority (medium/low) that are not urgent can cause the beacon to blink occasionally. The siren would remain off for medium/low priority alarms unless configured otherwise.
- If no alarms are active at all, both siren and beacon remain off (normal state).

Configurable Scenarios Per Priority: The exact relay behavior for each priority and alarm state (active vs. acknowledged) should be configurable. For instance, one might configure that even high priority unacknowledged alarms should sound the siren in some installations. To accommodate this, define a configuration (could be in code or a config file section) that for each priority level specifies what the **siren** and **beacon** do when an alarm of that priority is active and unacknowledged, and what they do when acknowledged (but not cleared). For example, by default: - Critical: Active=**Siren ON + Beacon ON (solid)**; Acknowledged=**Siren OFF, Beacon ON (solid)**. - High: Active=**Siren OFF, Beacon ON (solid)**; Acknowledged=**Siren OFF, Beacon BLINK (slow)** (or solid, depending on interpretation). - Medium: Active=**Siren OFF, Beacon BLINK (slow)**; Acknowledged=**Siren OFF, Beacon BLINK (intermittent)**. - Low: Active/Ack = perhaps no beacon, or a very slow blink, or only an LED indicator. These are just examples – the key is that the code should refer to a table of actions for each priority rather than hard-coding the exact behavior. The requirement explicitly notes a need for “a scenario for each alarm priority with standard options ACTIVE/ACKNOWLEDGED – siren action, beacon action” [7]. Implement this by maybe a struct or mapping (e.g., `alarmBehavior[priority][stage] = {siren: on/off/blink, beacon: on/off/blink}`), possibly configurable via a JSON or at least easily adjustable constants.

Relay Control Implementation: Leverage the existing `IndicatorInterface` or similar I/O control class for relay outputs: - Use the port names for the relays (e.g., "Relay1", "Relay2") that correspond to siren and beacon. For continuous ON/OFF, use the `writePort(name, state)` method. For pulsing, the `startBlinking(port, onTime, offTime)` method of `IndicatorInterface` can be used to implement the periodic flashing (since the class already supports blinking outputs). - For example, for a medium priority acknowledged alarm scenario where beacon should blink occasionally, you might call something like `indicator.startBlinking("Relay2", 10000, 120000)` for a 10-second ON, 2-minute OFF cycle (values configurable). The `IndicatorInterface` will handle toggling that relay [6]. - Ensure that when alarm states change (e.g., an alarm is acknowledged or a new higher-priority alarm arrives), you update the relays accordingly. This likely means in each cycle (or whenever an alarm state changes) calling a function (e.g., `handleAlarmOutputs()`) that: - Determines the *highest priority group of alarms that require attention* (e.g., if any unacknowledged critical, that’s highest group; else if any unacknowledged high or acknowledged critical, that’s next; else if any medium, etc.). - Activates/deactivates the siren and beacon based on that scenario. For instance, if it finds unacknowledged critical, it will ensure siren relay is ON and beacon relay ON (stop any blinking on beacon and set it solid ON). If it finds the next lower scenario, it might turn siren OFF and beacon ON (solid) etc. - Stops any blinking on relays that should now be solid or off (using `stopBlinking` for the port). - Uses `IndicatorInterface` to read current state to avoid unnecessary writes (if not needed). - The spare relay (Relay3) is not explicitly assigned in the requirements, but it could be reserved for future use or configurable for some custom alarm action. For now, it could remain unused or mirror one of the others if needed. We might leave Relay3 as an extra that could be controlled via Modbus or future features.

Visual LED Indicators: In addition to the relays, the device has onboard LEDs (Green, Yellow, Blue, Red as per original code on PCF8575) [8]. These should be used to indicate alarm status by priority: - When an alarm is active, turn on the LED corresponding to that alarm's priority: **Red for Critical, Yellow for High, Blue for Medium** [9]. For example, if there is any critical alarm active, light the red LED (perhaps blinking if unacknowledged vs solid if acknowledged, as a possible extension). If not critical but high alarms present, light the yellow LED, etc. This gives a quick visual indicator of the highest active alarm level. - The Green LED can be used to indicate normal operation (as it was originally, turned on after setup) [10]. It might remain on whenever the system has no active alarms, and possibly turn off or blink when there are alarms (depending on desired behavior). - Implement LED logic in tandem with relay logic in the alarm output handler. For instance, if a critical alarm is active, turn on Red LED, and maybe turn off lower-priority LEDs or use them for other info. If alarms are cleared, ensure to reset LEDs to normal (Green on, others off). - These LED indications should also be configurable if needed, but at minimum hard-code them as above to fulfill the requirement.

Relay Status via Modbus: Provide read-only Modbus registers or coils that reflect the current state of the siren and beacon relays (and possibly the spare relay). This allows external systems to know if the alarm outputs are active. Possible implementation: - Use **coil registers** (single-bit) for each relay output. E.g., Coil 1 = Siren relay state, Coil 2 = Beacon relay state, Coil 3 = Spare relay state. The Modbus server can map these to the actual output states (the `IndicatorInterface` can provide a `readPort` method for each). - Alternatively, use a holding register with bit flags for the relays (e.g., a 16-bit word where bit0 = Relay1, bit1 = Relay2, bit2 = Relay3, etc.), but coils are more straightforward for binary outputs. - Ensure these are updated in real-time or upon read requests. Since the `IndicatorInterface` likely maintains the latest output states, it can be queried when a Modbus read comes in for those addresses. - Document the addresses of these status registers in the Modbus register map (so users know where to look for siren/beacon status).

In summary, the alarm indication system will be priority-aware and flexible. A highest-priority-first approach ensures that the **loud** notifications (siren/beacon) follow the most urgent alarm present [5], and the behavior for each priority can be tuned. The system will visibly and audibly differentiate critical alarms from lesser ones and allow acknowledgment to mute appropriately. All of this will be user-configurable and reflected both on the device outputs and remotely via Modbus.

Web Interface Enhancements

The built-in web interface needs a significant upgrade for better usability, consistency, and new features. All pages should share a cohesive design and support the new configuration options:

- **Unified Look and Feel:** Apply a consistent theme (styles, color scheme, layout) across all web pages. In the current implementation, pages like the dashboard, sensor list, alarms, history, etc., may have had separate styling. Create a common CSS that all pages use for a uniform appearance (same header bar, fonts, buttons style, etc.). For example, ensure the navigation menu and page container styles are consistent (as seen in the current alarm-history page CSS) – this provides a professional, coherent user experience. A simple approach is to have a base template or include a common `<style>` or CSS file on all pages so that elements like navigation bars, buttons (`.btn-primary`, `.btn-warning`, etc.), and tables have the same styling on every page [11] [12].

- **Live Trend Charts for Temperatures:** Introduce a **Trend View** page that plots sensor temperature readings over time. This should show real-time graphs of each sensor's temperature with markers for alarm thresholds:
- For each sensor (or each important sensor), draw a line chart that updates periodically (e.g., every measurement cycle or every few seconds) to reflect its recent temperature history. Use a client-side JavaScript charting library for smooth visualization (e.g., Highcharts, Chart.js, etc.). The device can serve an endpoint (like an HTTP API returning JSON or CSV data points) that the page uses to fetch new data asynchronously.
- Overlay or indicate the configured high and low threshold values on the chart (e.g., as horizontal lines). This allows the user to see how close the temperature is to triggering alarms.
- Limit the number of data points (e.g., last 1 hour or last 100 readings) to keep performance reasonable. If needed, use a rolling buffer.
- Provide options to select which sensor's data to view if multiple sensors (e.g., a dropdown to pick the sensor).
- The chart should update live without full page refresh – this can be done with JavaScript making periodic AJAX calls to the ESP32. This approach is common for ESP32 web charts [13] and enhances the monitoring capability.
- For implementation, one can follow known examples of plotting sensor data on ESP web pages [14] [15] – typically by including a JS library and periodically pushing new points to it, as RandomNerdTutorials and others demonstrate.
- **Enhanced Alarm History:** Improve the alarm history page to allow better filtering and analysis of past alarms:
- Ensure filtering by priority, type, and status is available and intuitive (the current HTML already has some filtering and sorting functionality in JavaScript) [16] [17]. Extend this by possibly adding:
  - **Date/Time range filter**: so users can view alarms that occurred in a specific timeframe.
  - **Text search**: to find alarms by sensor name or description.
  - The ability to hide acknowledged alarms or cleared alarms to focus on unresolved ones.
- Possibly introduce pagination if the history log is large (though the current code already had a pagination system as indicated by `pageInfo` spans and logic) [18] [19].
- Make the history more readable: e.g., color-code rows by priority (the HTML already uses classes like `priority-high`, etc. for styling rows [20] [21]). Ensure those CSS classes are applied consistently (for example, critical = red background, high = yellow, etc., as seen in the code).
- Include an **export** function (if not already) to download the history as CSV for external analysis, if that's useful (there was mention of export in the code).
- If possible, allow acknowledging alarms from the history page as well (for any active alarms listed), though primarily acknowledgment is done in the main dashboard.
- **Main Settings Page Improvements:** Incorporate system time settings on the settings page. This includes:
- Displaying the current system date and time (and timezone if applicable).
- Options to either sync time via NTP (Network Time Protocol) or set time manually. For example, provide fields to enter date/time or a button "Sync with Browser Time" to capture the user's local time.
- If the device has internet access, an NTP sync option with a server address input can be given. Otherwise, at least allow manual setting.
- Ensure the time setting is saved (likely in the config) and that the system's RTC or SNTP is updated accordingly.
- Time settings are important for timestamping logs and alarm history, so having the correct time configured is critical.
- **Russian Translation of UI:** All text in the web interface must be translated to Russian. This means:

- Changing page titles, labels, button texts, menu items, and any messages to Russian language. For example, “Temperature Controller – Alarm History” would be in Russian, all form labels like “Device ID”, “Enable Modbus” etc., should be in Russian.
- Make sure to use UTF-8 encoding (the pages already declare `<meta charset="UTF-8">` which is good [22]). The HTML files should be saved in UTF-8 so that Cyrillic characters display correctly.
- If supporting both English and Russian is desired, one could implement a language toggle, but since the requirement explicitly says **translated to Russian**, we will assume Russian-only interface for the target deployment. In that case, simply replace the strings in the HTML/JS with Russian equivalents.
- Verify layout after translation, as Russian text might be longer in some cases. Ensure the design (buttons, table columns) can accommodate the translated text without breaking.
- Also translate any on-screen messages or alerts (e.g., JavaScript alert messages or placeholder text).
- Note: The Russian **user manual** (discussed later) should mirror the UI, so use consistent terminology in translation (for example, if “Critical” is translated as “Критический” in the UI, use the same in the manual).
- **Additional Web UI Features:**
- **Dashboard/Main Page:** Continue to show key real-time info (like current temperatures of points, and active alarms). Possibly enhance it by highlighting if any alarm is active (e.g., flash the value or show an icon next to the sensor reading).
- **Sensor Pages:** If there’s a page listing sensors or points (as indicated by sensors.html, points.html in the data folder), update those to show new configurable parameters if any (like if sensor calibration or names can be edited, etc.), and translate those to Russian. Ensure consistency in editing features (maybe allow editing threshold values there, which might already be implemented via CSV import or forms).
- **Alarm Settings Page:** We might create a new page (or repurpose an existing one) for configuring alarm priorities as mentioned. This page would list each alarm (point + type) with dropdown for priority and a checkbox for enable/disable. It should also allow saving those changes (which will update config file and registers). This page should be accessible from the main navigation (e.g., a menu item “Alarm Settings”).
- **Logs Download:** If not already present, ensure the “Download Logs” page (mentioned in the file list) works and perhaps include the ability to filter which logs to download (event logs vs alarm logs). Translate that page to Russian too.
- **Consistency:** Implement a common header or include on all pages so that, for example, the navigation bar is defined in one place. This way, adding a new page or changing a menu item only requires one edit. Since the pages are static HTML on the filesystem, one way is to duplicate a consistent `<nav>` HTML in each, but a more advanced approach could be to generate pages via templates. However, given limitations on the ESP32 (no heavy template engine), we might opt for manual consistency with careful replication of changes.

By upgrading the web interface as above, the user will have a much more powerful and user-friendly way to interact with the controller. They can view real-time graphs of sensor data (making the system behave like a mini SCADA/HMI), easily adjust alarm priorities, and navigate a polished, Russian-localized UI.

Display and Physical Controls Behavior

The device includes a physical **OLED display** and at least one **button** (plus the alarm LEDs discussed earlier). The firmware should handle these for improved usability:

- **Display Sleep and Wake:** To prolong the OLED lifespan and reduce power, implement a timeout to turn off (sleep) the display after a period of inactivity. In the current code, an `_oledSleepDelay` mechanism exists (e.g., set by `setOledSleepDelay`, tracking `_lastActivityTime`) [23] [24]. We should use this: for example, set `setOledSleepDelay(30000)` for a 30-second no-activity timeout (or make it configurable). "No activity" can mean no button presses and no alarm state changes in that period. When the timeout elapses, turn the display off (maybe by using `setOLEDOff()` or similar function which likely sets `_oledSleeping` true and clears the screen/backlight).
- **Wake conditions:** Any *button press* or *new alarm* should wake the display immediately. The `IndicatorInterface` has a `_wakeOLED()` method [25] that should be called when we need to wake the screen. Ensure that in the button ISR or poll handler, as well as when an alarm changes to active, we call `_wakeOLED()` so the user can see the new info. Also, when the user accesses the device via web (if we want), but primarily physical triggers are button or alarm.
- Ensure that after waking, normal display content is restored (if it was showing something prior or if we want to show a specific screen on wake, e.g., current status).
- The sleep delay (if -1 = never sleep) can be configurable via settings if desired, but default to a reasonable value (like 1 minute).
- **Startup Diagnostic Display:** On boot, use the OLED to display a brief **self-test/diagnostic** sequence:
- Show a startup screen with the firmware version, and perform checks: e.g., "Initializing... WiFi: OK, Modbus: OK, Sensors found: X" etc. This could be a text summary or icons (for example, use `displayOK()` and `displayCross()` methods to show a big OK or cross symbol for pass/fail of certain checks).
- If all checks pass, perhaps show a big "OK" (the code's `displayOK()` draws a large "OK" on screen) [26] [27], then proceed after a short delay to normal operation. If any check fails or has a warning (e.g., a sensor is missing or config error), display a warning message or an "X" symbol (`displayCross()`) [28] [29] along with a short description of the issue.
- **Pause on error/warning:** If a critical error is found at startup (for example, no sensors detected or unable to initialize SD card if used), consider halting the normal startup and keeping the error on display until the user acknowledges (maybe by button press). For non-critical warnings, maybe just delay a few seconds so the user can read it, then continue. This fulfills *"pause on error/warning"* – essentially, do not just flash an error and continue; let it be noticed.
- For instance, if one sensor failed to initialize, you might show "Sensor 5 error" on the OLED and maybe require a long press of the button to acknowledge before moving on.
- Log these startup diagnostics to the serial console or log file as well for record.
- **Button Short Press Behavior:** Utilize the device's button to navigate information on the display:
- On a **short press** (quick click) of the button, if there are any active unacknowledged alarms, the display should jump to showing the next active alarm's details. Essentially, cycle through active alarms one by one each time the button is pressed briefly [30]. Display the alarm message (e.g., "ALARM: Sensor 3 HIGH temperature 85°C" or similar) along with its priority (maybe color-highlighted text if possible or an icon) and whether it's been acknowledged or not.
- If **no active unacknowledged alarms** are present, then short presses should instead cycle through the bound sensors' readings. For example, show sensor 1's current temperature (and maybe min/max), then next press shows sensor 2, etc. Many sensors may be connected (the original supports up to 60 points), so cycling through all might not be practical; perhaps cycle

through only sensors that are *bound/active* (i.e., those that have been discovered and are in use). The code has `dsPoints` and `ptPoints` arrays – we can cycle through `point.getBoundSensor() != nullptr` entries.
- The display for a sensor could be something like: "Sensor 5: 37.4°C (OK)" or with an indication if its last alarm state. You might also include threshold info: "HighThr: 50°C, LowThr: 5°C" if space allows, or just the value.
- This feature allows the user to manually scroll through important info when there are no alarms demanding attention.
- **Button Long Press Behavior:** On a **long press** (press and hold for, say, 2 seconds):
- Enter a **"status info" mode** that shows system status details in a rotating manner. Upon detecting a long press, display a summary such as:
    - Line 1: Device name/ID and firmware version.
    - Line 2: WiFi status (e.g., "WiFi: Connected" or "WiFi: Disconnected") and IP address if connected.
    - Line 3: Number of sensor points active (e.g., "Sensors: 12 active").
    - Line 4: Number of active alarms currently, or last alarm time.
    - Maybe Line 5: Free memory or uptime for debugging, if relevant.
- After showing this summary for a few seconds, the display can automatically start cycling through a few detailed screens (round-robin). For example:
    1. **Network Info:** SSID, IP, signal strength.
    2. **Sensor Overview:** maybe list all sensor addresses or a compact list of their current readings in one screen if possible.
    3. **Alarms Overview:** count of each priority alarms (e.g., "Critical:0, High:1, Med:2, Low:0") [31] [32] , or similar info.
    4. **Any other relevant diagnostic info:** e.g., SD card status if logging to SD, etc.
- Each of these screens could show for ~2-3 seconds then automatically advance to the next (thus "round-robin"). After cycling through a predefined set of info (maybe 3–5 screens), it can exit the status mode and return to normal display (or the user can short press to exit immediately).
- If the button is kept held down, maybe continue cycling until release, but it's probably better to trigger on long press release to start the cycle and then automatically exit after one cycle.
- Ensure that during this status display mode, normal alarm display is paused (unless a new alarm comes in, in which case perhaps break out to show that).
- This feature gives a quick way to inspect system health without needing the web interface.
- **Alarm Display and Acknowledgment via Button:** The requirements imply the device should allow acknowledgment of alarms via the button as well. For example, if an alarm is showing on the OLED, perhaps a *long press* could acknowledge it (silencing the siren for instance). Alternatively, a double-click or another button (if available) could be used for acknowledgment. Since we only have one button mentioned, a possible scheme:
- Short press cycles alarms; if an alarm is currently being displayed on the OLED, a **long press while an alarm is shown** could acknowledge that alarm (mark it acknowledged in the system, stop siren). The display could give feedback like "Alarm Acknowledged" and then move to the next alarm or so.
- Implement logic in the button handler: if displaying alarm details and long press detected, call the `acknowledgeHighestPriorityAlarm()` or appropriate function for that alarm [33] . This aligns with needing to handle acknowledgment differently and showing highest priority first [30] .
- **Normal Display Content:** When there are no button presses and no new alarms, decide what the OLED should show in normal idle state. Perhaps cycle through the main readings automatically (e.g., every 10 seconds switch to next sensor reading) or just show a default screen:
- Possibly a rotating display of the top 3-4 sensor readings, or just a welcome screen with device name and maybe the time.

- However, the requirement *"When all the alarms in active status got acknowledged we need to show them in a cycle with some delay (10 seconds – it should be a setting)"* [34] suggests that once alarms are acknowledged, we should cycle through them on the display periodically. This means:
    - If there are acknowledged-but-active alarms (no unacknowledged left), the device will cycle through those alarm messages every 10 seconds so the user is reminded they are still active (just acknowledged) [30].
    - This cycle stops when alarms clear, or if a new unacknowledged alarm comes in (then we focus on that).
- So, in idle normal operation (no alarms at all), perhaps cycle through sensor data or just show "All OK" with current time.
- Use the `IndicatorInterface` text buffer (which supports up to 5 lines and scrolling) to display multi-line info if needed. The `pushLine` method can scroll text lines [35], which might be useful for long sensor names or IP addresses.
- **Implementing Button Press Detection:** Ensure the PCF8575 or GPIO handling for the button supports distinguishing short vs long press. A common approach:
- In the main loop or an interrupt, detect when button goes down and when it goes up, measure the duration.
- For example, on button down, record `pressStartTime`. On button up, if duration = `now - pressStartTime` > 2 seconds, treat as long press; if < 2 seconds (but > some debounce threshold like 50ms), treat as short press.
- If the PCF interrupt is used, note that we might not get continuous hold events, so we might need to poll while it's down to detect long hold.
- Alternatively, use the periodic `indicator.update()` call to poll the button port state (port 15 named "BUTTON" in code) [36]. The `_checkButtonPress()` method (as referenced in TemperatureController update loop) should be expanded to implement this logic.
- Debounce the button to avoid false triggers.
- **Feedback:** Provide user feedback on button actions: e.g., when a long press is registered, maybe blink the screen or an LED to indicate entering status mode; when an alarm is acknowledged via button, maybe flash the blue LED or show a quick "ACK" on display.

With these behaviors, the physical interface of the device becomes much more interactive and informative. The operator can rely on the device's screen and button for quick status checks and alarm acknowledgment even if the web interface is not in use. This is especially useful in an industrial setting where the device is physically accessible.

Modbus RTU Improvements

The system includes Modbus RTU (RS485) for integration with other systems. Two main improvements are needed: fix the config synchronization issue and extend Modbus mapping for new features.

- **Prevent Config Overwrites on Connection:** The current implementation has a bug where connecting a Modbus master or initializing the Modbus server causes the device configuration to be overwritten or reset. This might be due to calling a function like `applyConfigFromRegisterMap()` at the wrong time, which writes default or initial register values back into the config structure or file. We need to ensure that **existing configuration is not lost unless the master explicitly writes new values**.
- Audit where `applyConfigToRegisterMap` and `applyConfigFromRegisterMap` are called. Likely, `applyConfigToRegisterMap` should be used at startup (to load config into Modbus registers), and when config changes (to update registers). `applyConfigFromRegisterMap` should be used only when we want to accept Modbus writes into the system config.

- Remove or disable any automatic calls that push config from Modbus on connect. For example, do not blindly call `applyConfigFromRegisterMap()` on every Modbus client connection or poll. Instead, use it only in the context of handling a Modbus **write request**. The code likely uses `writeHoldingRegister` in RegisterMap to detect writes [37] [38] – when that returns true (value written), then apply that value to the corresponding setting.
- Ensure that when a Modbus master connects and only reads data, it cannot trigger a config save. Reading should be non-intrusive (just fetch from RegisterMap which mirrors internal state).
- Only when a Modbus write is performed (for example, writing a new threshold or changing a setting register) do we update the internal config and then persist it (maybe by calling the existing `savePointsConfig()` or `ConfigManager::save()` after applying).
- Test this by simulating a Modbus master connection and disconnection to ensure no unintended config resets.
- In summary, decouple the Modbus refresh from config save. Possibly maintain a dirty flag – if any Modbus write occurs to config registers, mark config as changed and schedule a save to flash.
- This addresses the *"overwrites config on connect"* problem by confining config writes to intentional actions.
- **Modbus Register Map Extensions:** With new features (alarm priority and enable config, relay status, etc.), extend the register map:
- **Alarm Configuration Registers:** As discussed, allocate registers for each alarm's priority+enable byte. We might organize this as sections:
    - Low Temperature alarm config for each sensor (if applicable) – e.g., 60 registers for 60 possible points.
    - High Temperature alarm config for each sensor – another 60.
    - Sensor Error alarm config for each sensor – another 60.
    - Or combine types per point: for each point, have 3 registers (low, high, error) each containing that alarm's config byte. The mapping should be clearly documented (e.g., "Register 300-359: High Alarm Config for Points 0-59", etc.). Use whatever address range is free or logical (the original RegisterMap likely has ranges for thresholds, current temps, etc. We must pick ranges that do not conflict).
    - Mark these registers as **read/write** (so master can enable/disable or change priority remotely). In `isReadOnlyRegister`, ensure these addresses return false.
    - In the `writeHoldingRegister` handling, when a value is written to one of these addresses, parse the byte (low 8 bits of the 16-bit register) to get enabled flag and priority. Then call a function to update the corresponding Alarm object. Possibly we will utilize `TemperatureController::updateAlarm(point, type, priority, enabled)` (or create such a function) to apply these changes in the system.
    - Similarly, for reads, the `readHoldingRegister` should return the composed byte reflecting current config.
- **Relay Status:** Provide a way to read the siren/beacon status:
    - If using coils, the Modbus library may have a mechanism for coils separate from holding registers. We can use discrete input or coil registers for outputs. If simpler, we can also map them into the holding register space: e.g., use one of the Device Status registers to pack relay states.
    - For example, use a single register where bit0 = siren, bit1 = beacon, bit2 = spare. Or use 3 consecutive registers each with value 0 or 1 for each relay.
    - Mark these as read-only (so masters cannot force control the relays—assuming we don't want external override of alarm relays; if override is needed for testing, we could allow writing to manually activate siren, but that complicates logic).
    - Implement the `readHoldingRegister` for those addresses to return the current output states (by reading the actual output states from `IndicatorInterface`).

    - Alternatively, implement as **Input Registers** if the Modbus stack differentiates (inputs are read-only by design).
- **Time and Settings:** If we added time config, expose those as well (e.g., registers for current timestamp or for timezone offset).
- Update the `RegisterMap` class to include new fields for any new registers, and adjust constants for register addresses (e.g., define `ALARM_CFG_START_REG` etc.). Keep the register addresses aligned and grouped logically for ease of documentation.
- **Testing Modbus:** After implementing, test with a Modbus master tool (like a PC simulator or another device) to ensure:
    - Reading temperatures, thresholds still works as before.
    - Writing a threshold changes the device's threshold (and does not revert on reconnect).
    - Reading/writing the new alarm config registers correctly enables/disables alarms and changes priorities (verify by then checking via web UI or logs that the alarm's behavior changes).
    - Reading relay status matches actual alarm activation (e.g., trigger an alarm and see the modbus coil become 1).
- Document the Modbus map changes clearly (point in the implementation guide and user manual what addresses correspond to what).

By fixing the config flush issue and adding these registers, the Modbus interface will be robust and encompass the new functionality. External systems (like SCADA or PLCs) can not only monitor temperatures and alarm statuses but also remotely configure which alarms are important (priority) and enabled. And crucially, our device won't lose its settings just from a Modbus connection event.

Codebase Refactoring and Maintainability

To facilitate easier future development and clarity, the project's code should be refactored and well-documented:

- **Centralize Hardware Definitions:** Create a single header file (e.g., `HardwareConfig.h` or `Pins.h`) that contains all the initial GPIO pin assignments, constants, and hardware-related magic numbers. This includes:
- ESP32 pin numbers for built-in interfaces (RX/TX pins for RS485, OneWire bus pin, etc., as present in config or code).
- I2C addresses for devices (e.g., the OLED display I2C address, the PCF8575 I/O expander address).
- Port number assignments on the PCF8575 for each named port: e.g., define constants for `PCF_PORT_SIREN = 0`, `PCF_PORT_BEACON = 1`, `PCF_PORT_SPARE = 2`, `PCF_PORT_GREENLED = 4`, etc., so that it's clear and only defined once. Currently, these are set by `indicator.setPortName("Relay1",0)` etc. [39] – we can still use the names, but having #defines or an enum would be helpful.
- Any other system constants like maximum number of sensors (50 DS18B20, 10 PT1000), alarm check interval, default sleep timeout, etc.
- By consolidating these, anyone can quickly adjust pin mappings or change hardware without hunting through code. It also avoids magic numbers scattered in code.
- **Organize Source Code by Functionality:** Restructure the project into logical folders/modules. Currently, it appears all source files are in a flat `src` directory. We should group them, for example:
- `src/alarms/` – contains `Alarm.h/Alarm.cpp` and possibly an `AlarmManager.cpp` if we create one to handle the alarm initialization and logic. This module deals with alarm objects and their states.

- `src/sensors/` – contains `Sensor.cpp`, `MeasurementPoint.cpp`, and any sensor-specific code (like OneWire reading). This encapsulates sensor reading logic.
- `src/ui/` – contains `IndicatorInterface.cpp` (which manages OLED, LEDs, button, etc.), and perhaps any display/menu logic.
- `src/modbus/` – contains `TempModbusServer.cpp` and `RegisterMap.cpp`, focusing on Modbus communications.
- `src/core/` (or `src/system/`) – contains `TemperatureController.cpp` (the main application logic), `ConfigManager.cpp` for config file handling, `TimeManager.cpp` if any, etc., and `main.cpp`.
- Update include paths accordingly and ensure the PlatformIO or build system knows about these subfolders.
- This separation improves clarity: for instance, all alarm-related code is together, so a developer working on alarm logic knows where to look.
- **Improve Class Structure (if needed):** Generally preserve the original class design (to not rewrite everything), but consider if any new classes are needed:
- Perhaps an `AlarmManager` class can be introduced to encapsulate the new alarm initialization, sorting, and output handling logic, instead of overloading `TemperatureController` with too many responsibilities. The note even mentioned "TemperatureController class or a new class if advisable" [40] – it might be advisable to have an AlarmManager that TemperatureController uses.
- If so, AlarmManager could contain the list of Alarm objects, and methods like `initAlarms()`, `updateAlarms()`, `handleAlarmOutputs()` etc., thereby keeping TemperatureController.cpp cleaner.
- Similarly, if the web configuration for alarms is complex, maybe a separate handler module for that (though likely not needed).
- Ensure any new class fits logically and does not duplicate existing ones.
- **Doxygen-Compatible Comments:** All source code should be commented thoroughly in a way that tools like Doxygen can generate documentation. This means:
- Every file starts with a header comment describing its purpose, the module, author, etc., using Doxygen tags like `@file`, `@brief` [41].
- Every class and struct has a documentation comment (`/** ... */`) explaining what it represents.
- Every public function and important internal function has comments with `@brief` description, `@param` for each parameter, and `@return` for return values [42]. For example:

```
/**
 * @brief Checks all alarms and updates their state based on sensor
readings.
 *
 * Iterates through all configured alarms, evaluates conditions, and
updates active/acknowledged status.
 * @param currentTime The current timestamp (ms) to compare with last
check time.
 * @return Number of active alarms after update.
 */
int AlarmManager::updateAlarms(unsigned long currentTime);
```

- For complex logic, include additional remarks or examples in the comments. If a function is critical or tricky, document the approach or formula used.

- Use `@note`, `@warning` for any important warnings to future maintainers (e.g., “@warning This function must be called only after sensors are initialized”).
- Document the config file format and web interface integration points as needed within comments.
- The goal is that running Doxygen will produce a neat manual of the code, which aids future developers and is often expected in industrial projects.
- This also implies maintaining consistent naming conventions and clarity in the code itself, which a refactoring can address (for example, ensure naming is self-explanatory, avoid excessively long functions by breaking into sub-functions, etc.).
- **Testing and Debugging Aids:** While refactoring, maintain or add debug log statements (via the LoggerManager or Serial prints) especially around the new features:
- E.g., log when an alarm changes state, when the siren/beacon state changes (“INDICATION: Critical alarm acknowledged, turning beacon on constant, siren off” etc.), so that the system’s behavior can be traced in logs.
- These logs can be wrapped in conditions or appropriate levels (info, warning, error) to be turned off in production if needed.

Overall, the refactoring should *not* change how the system functions from an external point of view, but rather improve the internal organization and readability. Future maintenance or expansion (say adding a new sensor type or a new alarm type) would be easier with this structure and documentation in place. Following these guidelines will produce a clean codebase that adheres to high standards and is accompanied by thorough documentation.

Documentation and Deliverables

In addition to the code changes, the project requires written documentation for both developers and end users:

- **Implementation Guide (Markdown):** Prepare a detailed guide explaining how the new system is structured and how to build or extend it. This should be written in Markdown (for easy viewing on GitHub or other platforms). Include:
- **Project Structure:** Outline the folder hierarchy and the purpose of each directory and key file. For example:
    - `/src/alarms/` – contains alarm logic classes.
    - `/src/ui/` – OLED and indicator handling.
    - etc.
- **Setup Instructions:** If a developer needs to set up the build environment (e.g., PlatformIO or Arduino IDE libraries), list the steps and prerequisites (ESP32 board support, necessary libraries like OneWire, etc.).
- **New Features Summary:** List the major new features (priority config, new UI pages, etc.) and mention which part of the code implements them. For instance, “AlarmManager class handles sorting and prioritization of alarms as described in Alarm Handling Restructuring section.”
- **Data Flow and Class Interactions:** Describe how data moves through the system. E.g., *“Sensor readings are taken in* `TemperatureController::update()` *by calling* `updateAllSensors()`*; then* `updateAlarms()` *is called, which uses AlarmManager to evaluate conditions. If an alarm triggers, AlarmManager updates the Alarm object and TemperatureController then calls* `handleAlarmOutputs()` *to update relays and LEDs.”* This gives a clear picture of runtime logic.
- **File and Folder Creation Commands:** Include a section that literally lists the shell commands used to create the new folder structure and files (helpful if the developer will be doing this manually or wants to replicate it):

```
mkdir src/alarms
mkdir src/ui
mv src/Alarm.cpp src/alarms/Alarm.cpp   # Example of moving file to new
folder
touch src/alarms/AlarmManager.h src/alarms/AlarmManager.cpp
```

and so on. This serves as both a record of what changes were made and a guide if someone were to set up the project structure from scratch.

- **Build & Flash Instructions:** how to compile and upload to the device, including how to upload the SPIFFS/LittleFS data for the web pages (since there are HTML files, mention using the uploader plugin or `pio run -t uploadfs` if PlatformIO).
- **Configuration Guide:** Explain the config file (if it's a CSV or YAML) and how priorities and thresholds can be adjusted there. Document any new config fields introduced (like if we add time zone, etc.).
- **Modbus Map Documentation:** Very important – list the Modbus registers and their functions in a table. For example: | Address | Description | R/W | Range/Format | |---------|--------------------------------------|------|---------------------| | 0 | Device ID | R | 1–9999 | | 1 | Firmware Version (e.g., 0x0101 for 1.1)| R | BCD or format used | | 100-159 | Low Alarm Thresholds for DS18B20 (point 0-59) | R/W | Int16 (°C*100 perhaps) | | 300-359 | High Alarm Config (Priority+Enable) for points 0-59 | R/W | Byte: b7=enable, b0-1=priority (0-3) [5] | | ... | Siren Relay Status (coil 0 or reg 500) | R | 0=off, 1=on | (The above is just illustrative – use actual decided addresses). This table ensures the Modbus integrator knows how to use the new features.
- **Acknowledgment Mechanism:** Document how alarm acknowledgment works now (via web UI button, or physical button long press), so a developer understands that logic.

- The guide should be written in clear English, well-formatted with sections, bullet points, and possibly snippets of code or config for illustration. It's essentially a developer README for the project.

- **Russian User Manual (Markdown):** Prepare a user-facing manual in Russian, which will explain how to use the device and its interface. This should cover:

- **Introduction:** Brief description of the device's purpose (monitor temperature, trigger alarms) – written in Russian.
- **Hardware Overview:** Describe the device components (ESP32 controller, sensors, relays for siren/beacon, OLED display, buttons, etc.) – so the user knows what each part does. E.g., "Устройство оснащено звуковой сиреной и сигнальным маячком на реле, а также цветными светодиодами (красный, желтый, синий, зелёный) для отображения приоритетов сигнализации."
- **Getting Started:** How to power it on, connect to it (if WiFi AP or network needed), initial setup steps. How to access the web interface (provide default IP or mDNS name, etc.).
- **Web Interface Guide:** For each page of the web UI (now in Russian), provide instructions:
    - Dashboard: what information is shown (e.g., current temperatures, active alarms) and how to interpret it. How to acknowledge an alarm from the web (e.g., clicking an "Acknowledge" button next to an alarm if available).
    - Trend Graphs: how to view the temperature graphs, select sensors, understanding threshold lines.
    - Alarm History: how to filter and understand the log of alarms.

    - Settings: how to change settings like WiFi credentials (if applicable), device time, and particularly the **Alarm Priorities page** – explain how to use the dropdowns to set priorities, enable/disable alarms, and what the effect is (for example, disabling an alarm means that condition will not trigger a siren or beacon).
    - Time Settings: if we added on the settings page, instructions on setting the time or enabling NTP.
    - Maybe Modbus settings if the user is expected to configure Modbus address/baud via the UI (as per config.yaml snippet, those exist) – explain in Russian how to set the Modbus parameters to integrate with other systems.
- **Using the Physical Interface:** Explain the button presses and OLED indications (this is crucial for users in the field):
    - e.g., “Короткое нажатие кнопки: просмотреть текущие аварии или температуру датчиков. Длительное нажатие: показать статус системы (Wi-Fi, количество датчиков, и т.д.).”
    - Describe the LED signals: “Красный светодиод горит при критической аварии, жёлтый – при высокой, синий – при средней. Зелёный светодиод означает нормальную работу.”
    - Explain the siren/beacon behavior in user terms: “Сирена будет звучать непрерывно при не подтверждённой критической аварии; маячок будет мигать при менее критических ситуациях…” etc., so they know what the patterns mean.
    - How to acknowledge an alarm via the button (if implemented): “Чтобы отключить звуковую сигнализацию, нажмите и удерживайте кнопку, когда на экране отображается авария.”
- **Maintenance and Troubleshooting:** Provide guidance on what to do if something goes wrong:
    - If a sensor fails (the system should show a sensor error alarm), instruct the user that the device will disable that alarm’s thresholds automatically and perhaps how to replace the sensor.
    - If the device time is wrong, how to correct it.
    - If Modbus communication fails, check wiring, etc. (Basic tips).
    - Remind them how to reset to defaults if needed (if such a procedure exists).
- **Appendix:** Could include the Modbus register map (translated to Russian, if the end-user or integrator needs it).
- The manual should be written in clear Russian, using appropriate technical terms in Russian. All interface elements should be referred to exactly as they appear on the screen (which will now be in Russian).
- Format the manual in Markdown with headings for each section and possibly bullet points or numbered steps for procedures. If possible, include screenshots or diagrams (though the prompt says no need to search for images explicitly; if easy, maybe not required).

Both documents (implementation guide and user manual) are key deliverables. They ensure that future developers can pick up the project easily, and that end-users/operators can effectively use the system. The Russian manual especially will help on-site personnel who may not speak English to understand and trust the system.

Finally, once all code changes are done, thoroughly **test the entire system**: simulate various alarm scenarios to ensure the priority logic works (check that siren/beacon behave as intended in each case [5], and that disabling an alarm truly prevents it from triggering outputs), test web UI functionality (in multiple browsers if possible, and that Russian text displays correctly), test button short/long presses and that the OLED updates properly, and test Modbus interactions with an external tool. The combination of robust implementation and comprehensive documentation will yield a successful rewrite meeting all specified requirements. [5] [43] [13] [42]

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [43] TempController.txt
file://file-24PjGyNybKk6UNhR3Z794v

[13] [14] [15] ESP32/ESP8266 Plot Sensor Readings in Real Time Charts | Random Nerd Tutorials
https://randomnerdtutorials.com/esp32-esp8266-plot-chart-web-server/

[41] [42] 15-410 Coding Style and Doxygen Documentation
https://www.cs.cmu.edu/~410/doc/doxygen.html