ChatGPT

Plan for Developing Custom Firmware and Loader for Awair Glow C & Awair Element

Overview: Both the Awair **Glow C** and **Awair Element** indoor air quality monitors were discontinued by the manufacturer in 2022, losing cloud support and mobile app functionality [1]. Without Awair's servers, these devices can no longer report data to users, effectively turning them into e-waste. This plan outlines how to **develop a custom open-source firmware** for these devices – enabling local Wi-Fi access to sensor readings – and how to create a **firmware loader** that uses Bluetooth from an iOS device or Mac to install the new firmware. The goal is to restore full use of the Glow C and Element (temperature, humidity, VOC, etc.) entirely **offline**, with data accessible via MQTT or REST API for integration into Home Assistant or other user applications.

Hardware Analysis and Preparation

Awair Glow C device (combination smart plug, air sensor, and nightlight). The Glow C is a compact plug-in unit with onboard sensors and a motion-triggered LED nightlight [2]. Awair Element (not shown) is a tabletop monitor with a display and additional sensors.

Microcontroller & Connectivity: The Awair devices use a powerful IoT microcontroller module to handle sensors and networking. The Awair Element is built around a Laird **Sterling-EWB** module containing an STM32F412 ARM Cortex-M4 MCU with integrated Wi-Fi (2.4 GHz 802.11n) and Bluetooth (BLE) via a Cypress CYW4343W chip [3] [4]. The Glow C, released in the same timeframe (mid-2019), is presumed to use a similar architecture – likely the same STM32F412 + CYW4343W module – given that its FCC listing shows both Bluetooth and 2.4 GHz Wi-Fi radios present [5]. This means both devices have a capable 32-bit MCU and built-in wireless connectivity to support our custom firmware.

Onboard Sensors: Each device includes a set of environmental sensors (mostly I^2C or UART peripherals) that the custom firmware must support:

- **Awair Glow C Sensors:** Measures temperature, humidity, and volatile organic compounds (VOCs). According to Awair's own disclosures, the Glow C uses Sensirion's **SHTC3** temperature/humidity sensor and the **SGP30** VOC sensor [6]. These are tiny, low-power digital sensors with factory calibration (the SGP30 can output a TVOC index and an eCO_2 estimate). The Glow C also has a **motion detector** (PIR sensor) on the front for its nightlight trigger and an RGB LED for the nightlight itself [2]. Additionally, it has a built-in **relay** that controls the power outlet (allowing it to toggle a plugged-in device on air quality triggers). Our firmware will need to read from SHTC3 and SGP30 over I^2C and read the PIR sensor via a GPIO, as well as control the RGB LED and relay output.

- **Awair Element Sensors:** The Element is a larger device with **more sensors**, including particulate matter and CO_2. It uses a Sensirion **SHT30** for temperature/humidity and **SGP30** for VOC (same family as Glow C) [7]. In addition, it has a **Honeywell HPMA115S0** laser dust sensor for PM2.5 and a **Telaire T6703** NDIR sensor for CO_2 [8]. (It also has an ambient light sensor and a dot-matrix style LED display on the front to show air quality scores). The custom firmware on Element should interface with these sensors (HPMA115S0 typically communicates via UART, and T6703 via UART or I^2C) and drive the LED display if we want to utilize it. For initial firmware development, focus can be on basic sensor readings and Wi-Fi connectivity; support for the display and any extra features can be added incrementally.

Physical Access and Disassembly: To install a new firmware, we first need hardware access to the device's programming interface. Opening the Awair units is a delicate process but achievable with common tools. On the original Glow (2016 model), for example, the wood top-cover could be pried off to reveal four screws, allowing the internal assembly to slide out [9]. The Glow C is similarly well-built and likely uses clips or hidden screws (its top grille or panel can be gently pried open). **Carefully open the Glow C** by prying at the seams (avoiding damage to the enclosure); once opened, locate the printed circuit board and any debug/programming header. On the Glow C's board (and Element's), look for **SWD pins** or a JTAG header for the STM32 MCU. The FCC internal photos and community notes indicate that the main board is labeled e.g. *"AWAIR-LITE-MAIN V2.0"* (for the Element) [10]; on this board, test pads likely expose the STM32's SWDIO, SWDCLK, NRST, GND, and 3.3V. If no header is populated, you may need to solder fine wires to these pads.

- **Identify Debug Interface:** Using a multimeter and the STM32F412 pinout, identify the SWD pins on the PCB. It's common for SWDIO/SWDCLK to be on a Tag-Connect footprint or test pads. Connect an ST-Link or J-Link programmer to these pins. **Verify MCU Lock State:** Attempt to read the chip's memory. If the firmware readout protection is disabled, take the opportunity to dump the original firmware for reference. (This can help in understanding the device's original behavior or retrieving calibration data, though it's not strictly necessary for writing new firmware.) In case the MCU is read-protected (likely true for a production device), you can still erase it via the programmer to install custom firmware – the lock just prevents reading the existing code.

- **Power Considerations:** Both devices are mains-powered (Glow C plugs directly into an outlet, Element uses an adapter). For safety, do all board-level work with the device unplugged. Use the 3.3V from the ST-Link or a bench supply to power the board during programming instead of live AC. Alternatively, find the board's low-voltage regulator and feed it externally. This ensures you can

program and test the MCU in isolation. Once programming is done, you will reassemble the device for full testing on mains power (Glow C's relay and sensor readings under actual conditions).

Firmware Development Environment

Developing the firmware will require an embedded toolchain and possibly vendor SDKs due to the Wi-Fi/BLE component. We have two main approaches:

1. **Use the Cypress/Laird SDK (WICED or ModusToolbox):** The Sterling-EWB module (STM32F412 + CYW4343W) is *"fully compatible with Cypress's WICED SDK"* [11]. WICED (and its successor, Infineon ModusToolbox) provides libraries for Wi-Fi, BLE, and an RTOS (likely using ThreadX or FreeRTOS under the hood). Setting up ModusToolbox with the STM32F412 target and CYW4343W Wi-Fi/Bluetooth chip support can jump-start development – we can leverage existing drivers for the wireless stack and focus on application code. Laird may also provide sample applications for their module (for example, an AT-command firmware or demo sensor application) which could be repurposed or studied [12] [13].

2. **Use a Custom RTOS or Framework:** If not using WICED, an alternative is to use an open-source RTOS (FreeRTOS or Zephyr) and integrate drivers for the CYW4343W. This is more complex, but there are precedents: for example, the Raspberry Pi Pico W uses the related CYW43439 chip with an ARM core, and an open-source driver (lwIP + cyw43 driver) is available. We could port a similar driver to the STM32. However, given the complexity of Wi-Fi stacks, the **recommended path is to use the proven WICED/ModusToolbox environment**, which already supports the hardware. This saves time on low-level Wi-Fi and BLE bring-up. The development will be done in C/C++ using an ARM GCC toolchain (or the IDE provided by the SDK).

Project Structure: The firmware will be organized with a **small bootloader** (for OTA updates) and the main application firmware. We will reserve a section of flash for the bootloader (responsible for Bluetooth DFU and recovery) and the rest for the application. The application firmware will include tasks for sensor sampling, networking (Wi-Fi + MQTT/HTTP), and any device logic (LED control, etc.). Using an RTOS (WICED's built-in or FreeRTOS) will help manage these concurrent tasks.

Sensor Drivers: Develop or integrate drivers for each sensor component: - For **SHTC3/SHT30 (Temp/Humidity)** and **SGP30 (VOC)**, we can use Sensirion's publicly available I^2C driver code or write our own based on datasheets. Sensirion provides example code for SGP30 to retrieve TVOC and CO_2-equivalent values, including handling of baseline calibration. We'll schedule periodic readings (e.g. every 1 second for T/H, every 2–10 seconds for VOC). - For **CO_2 (T6703)** and **PM2.5 (HPMA115S0)** on the Element, implement UART polling. The T6703 typically outputs CO_2 readings over UART every 2 seconds by default, and the HPMA115S0 can be switched to active mode to stream particulate counts. Ensure to include any necessary warm-up or fan control for the dust sensor (it may need a spin-up period). - Validate sensor data against known-good values (for example, compare temperature/humidity with a reference sensor after flashing firmware to ensure our readings are correct).

Device Functionality: The custom firmware will restore all critical functionality of the devices, but without reliance on any cloud:

- **Reading Sensors Locally:** Continuously gather data from all sensors. Maintain internal state for air quality metrics (e.g. compute an "air quality score" if desired, or simply provide raw values). Ensure VOC sensor's baseline is managed (the SGP30 requires saving a baseline value periodically; we can store this in MCU flash every ~12 hours and reload it on boot to preserve accuracy [6]).

- **Control Outputs:** On the Glow C, control the smart plug relay and the RGB LED. The relay can be exposed to the user (via Home Assistant) as a controllable switch. The LED can either reflect air quality (as the original did with colors) or be user-configurable. Initially, we might set the LED to green/yellow/red based on air quality or simply allow Home Assistant to command it. The motion sensor in Glow C can be used to trigger the nightlight – this logic can be replicated (e.g. turn on LED when motion detected in dark conditions) or left for the user to handle via automations in Home Assistant.

- **Networking:** Initialize the Wi-Fi connection on boot. Since the device will no longer use cloud APIs, it will connect to the user's **local Wi-Fi** network and operate entirely locally. We will set up the Wi-Fi credentials either via a provisioning step (see **Firmware Loader** section) or by having the device start in AP mode if unconfigured (a captive portal to enter Wi-Fi details is an option, though BLE provisioning is simpler for headless devices like these). Use DHCP to get an IP or allow static IP configuration via a config file or BLE command.

- **Local APIs (MQTT/REST):** The new firmware will publish sensor data over **MQTT** and serve a small **REST API**:

- *MQTT:* The firmware will connect to a user-specified MQTT broker (likely on the local network, e.g. a Home Assistant Mosquitto broker). Each sensor reading can be published on a topic (for example: `home/awair_glowC/temperature`, `.../humidity`, `.../voc`). We can also publish a structured JSON to a topic like `home/awair_glowC/air_data` for convenience. MQTT is lightweight and near real-time, which is ideal for Home Assistant integration. We will include an option for the user to configure the broker address, credentials, and topics (possibly set via a config file or via the BLE loader app during setup).
- *REST API:* For simplicity, we can implement an HTTP server on the device (using WICED's webserver library or a lightweight HTTP library). This server can expose endpoints such as `http://<device_ip>/air-data/latest` returning JSON (similar to Awair's original local API [14]). For example, a GET request could return a JSON with fields like temperature, humidity, voc, etc. The Element's data (with CO_2, PM2.5) would include those fields as well. This allows users to `curl` the device for a quick reading or integrate via Home Assistant's RESTful sensor platform if they prefer HTTP. We will keep the REST API read-only for sensor data and perhaps a few control endpoints (e.g. to toggle the relay or set LED color).

- Both MQTT and REST will be strictly local (no cloud). Since this is on a home network, security can be minimal, but we might allow an **optional authentication** (HTTP basic auth or an API key for REST, and user/pw for MQTT which is usually handled by the broker). By default, the device will be accessible to any client in the LAN.

- **Home Assistant Integration:** Once the device is publishing to MQTT, integrating with Home Assistant is straightforward. We can follow Home Assistant's MQTT Discovery format so that the device announces itself to HA. For example, on first connect, the firmware can send MQTT messages to the HA discovery topic (`homeassistant/sensor/.../config`) for each sensor, so Home Assistant will automatically create entities for temperature, humidity, VOC, etc., with units and device metadata. Alternatively, the user can manually add MQTT sensors in their HA config using the topics documented. If using the REST API instead, the user can configure HA's `RESTful Sensor` to poll the JSON endpoint. However, MQTT push updates are recommended for real-time data.

Development Timeline: Breaking down the firmware coding into stages will help manage progress: - *Stage 1:* Bring up the MCU with a basic WICED project. Verify that we can blink an LED or read a dummy sensor. Then get Wi-Fi working – connect to a test AP and send a simple test packet or ping. - *Stage 2:* Implement sensor I/O one by one. Print sensor readings to serial or debug output for verification. - *Stage 3:* Implement local communication: stand up the MQTT client (publish dummy data initially) and HTTP server. Test these with a PC (e.g. use an MQTT client to subscribe and see data, or hit the REST endpoint in a browser). - *Stage 4:* Integrate everything: combine sensor readings with the MQTT/REST outputs at a reasonable frequency (e.g. publish every 1 minute or on significant change to avoid flooding). Ensure the system is stable (no memory leaks, able to run for days). - *Stage 5:* Add polish: LED behavior, any configuration options, and finalize the bootloader for OTA updates.

Throughout development, **testing on actual hardware** is crucial. Use the Glow C (which the user has) to test the basic firmware – since Glow C has fewer sensors, initial tests are easier (just T/H and VOC). Once stable, extend support to Element's extra sensors. It would be wise to recruit a community member who owns an Awair Element to test the firmware on that device (given the user does not have one physically). The modular nature of the code (abstracting sensor drivers) will allow building one firmware that auto-detects which device it's running on (perhaps via a hardware ID or simply by trying to query the CO_2 sensor and seeing if it exists).

Firmware Loader and OTA Update Mechanism

To make the custom firmware accessible to average users (who may not have debuggers or want to open the device), we will create a **firmware loader utility** that uses Bluetooth. The idea is to use the device's BLE capability to perform a **Device Firmware Update (DFU)** wirelessly from an iOS or macOS application. This requires two components: a **bootloader on the device** that can receive new firmware via BLE, and a **loader app** on the phone/PC that sends the firmware file.

Bootloader Design (Device-side): We will reserve a small section of flash memory (for example, 64KB–128KB) for a bootloader program. This bootloader runs immediately on reset and decides whether to launch the main application or enter DFU mode. The logic may be: if a specific button is held (e.g. Glow C's power button) during reset, or if a special flag is set in flash (perhaps triggered by the loader app), then stay in **DFU mode**. In DFU mode, the bootloader will advertise a Bluetooth Low Energy service that the loader app can connect to. We will implement a simple **GATT-based protocol** to transfer the firmware: - The bootloader defines a custom BLE service (e.g. UUID "AwairDFU") with characteristics for control and data transfer. - The iOS/Mac app connects and sends a "start DFU" command, then streams the firmware binary in chunks (the bootloader writes these chunks to flash memory in the application region). - After transfer, the bootloader verifies the firmware (e.g. using a CRC checksum) and then flags the new firmware as valid

and resets the device to boot into the main application. If anything fails (corrupted transfer, wrong checksum), the bootloader will not overwrite the existing firmware or will remain in DFU mode to retry.

We can model this process on Nordic's DFU protocol (used for nRF chips), which is well-documented and for which mobile libraries exist. In fact, adopting a known protocol can save effort: for instance, the **Secure DFU** system from Nordic uses BLE to send firmware with a predictable sequence. Our STM32 bootloader can implement a similar scheme (start packet, data packets, CRC, etc.), allowing us to potentially use existing DFU client libraries on iOS. Alternatively, we implement a custom but simple protocol and write our own client logic.

iOS/Mac Firmware Loader App: The user-facing tool will be a small application that allows selecting the new firmware file and uploading it to the device over BLE. Key requirements and steps for the app:

- **Device Discovery:** The app will scan for BLE advertisements. In normal mode, the Awair devices might not advertise openly (they typically only advertise during initial setup or if not configured). Our custom bootloader, when in DFU mode, will advertise with a name or UUID (e.g. "GlowC-DFU" or a service UUID specific to our loader). The app will look for that. If the device is currently running stock firmware, the user may need to manually put it into our DFU mode by first flashing a minimal bootloader via hardware or by other means (see **Initial Flash Options** below). Once our custom bootloader is on the device, subsequent updates can be done purely via the app.

- **Cross-Platform Implementation:** Since the user is open to cross-platform solutions, we have options:

- Use a **single codebase** (for example, a Python script or Flutter app) that can run on macOS and possibly iOS. However, iOS has restrictions (Python scripts won't run on iOS without special apps), so a better approach is using a high-level language for Mac and native for iOS.
- We can develop the loader in **Swift**, taking advantage of Apple's CoreBluetooth framework. Using SwiftUI, we could even create a simple UI that is portable via Mac Catalyst (i.e. the iOS app can run on macOS with minimal changes). This way, we essentially maintain one Swift codebase for both platforms (Catalyst allows an iOS app to run on Mac, or we make a Mac command-line tool separately).

- Alternatively, develop a **desktop application** for Mac (could be in Python using Bleak library or in Swift) and a separate **mobile app** for iPhone. Given the scope, a Swift/SwiftUI implementation with Catalyst might be the cleanest solution to cover both.

- **Workflow of the Loader App:**

- The user launches the app on their iPhone or Mac. The app scans for the Awair's BLE signal.
- The user puts the device in DFU mode (for example, by holding a button while plugging it in, which our bootloader will interpret, or via a special BLE command from the app if the device is running our firmware already).
- The app finds the device's BLE advertisement and connects. It then displays device info (could show the current firmware version).
- The user selects the firmware binary (on iOS, we can bundle the firmware file or use a file picker/ iCloud; on Mac, a file open dialog).

- The app splits the file into chunks (the bootloader will dictate chunk size, e.g. 512 bytes, to fit BLE payloads). It then writes these chunks sequentially via the BLE characteristic. We'll utilize BLE write-without-response or notify to stream efficiently. A progress bar in the app can show percentage complete.
- Once done, the app sends a "validate & reboot" command. The device should reboot into the new firmware. The app can then scan for a normal advertisement from the main firmware (or the user can verify functionality in Home Assistant, etc.).

- If any error occurs (timeout or checksum mismatch), the app will report it and allow retry. Our bootloader will remain in DFU mode until a valid firmware is flashed, so there's no risk of bricking as long as the bootloader itself is intact.

- **Wi-Fi Provisioning via App:** The first time we flash the custom firmware, the device won't have Wi-Fi credentials stored (since it's no longer connected to Awair cloud). We should include a mechanism to set the Wi-Fi SSID and password. The loader app can handle this right after flashing: for instance, once the new firmware boots, it could start a temporary BLE service (in the main app) to accept Wi-Fi config. We could reuse the same app to send the Wi-Fi credentials. Another method is to have the device start in AP mode (hosting a config Wi-Fi network) after flashing, but since we already have BLE in use, it's simpler to do it via BLE. So, the plan is:

- After DFU, when the main firmware runs for the first time and finds no Wi-Fi configured, it advertises a **Provisioning BLE service**. The loader app, after sending firmware, can automatically switch to provisioning mode: send the SSID and password to the device over BLE.
- The device then saves these and attempts to join the Wi-Fi network. The app can wait and confirm (perhaps the device can send a BLE notification or the LED can blink to indicate success).
- Once Wi-Fi is set, the BLE config service can shut down and the device will begin normal operation (publishing data). From then on, the BLE is not needed unless updating firmware again.

By designing the loader and firmware bootloader together, we ensure a **smooth user experience:** A user should be able to take a Glow C out of the drawer, use our app on their iPhone to *"upload new firmware and connect it to Wi-Fi"* in one flow, and then immediately see sensor data in Home Assistant.

Initial Firmware Installation: One challenge remains: getting the custom bootloader onto the device in the first place. Because the devices currently run stock firmware (which does not know about our DFU protocol), we might need a one-time manual flash. There are a few strategies: - **Manual SWD Flash:** For developers or advanced users, the surest method is to use an SWD programmer as described earlier. Flash the bootloader and firmware binary via STM32CubeProgrammer or OpenOCD. After this, subsequent updates can be done via BLE DFU. This method requires opening the device and hardware access, which not all users are comfortable with. - **Intercepting OTA Update:** An inventive software-only approach is to exploit the device's original OTA mechanism. The Awair cloud used `ota.awair.is` for updates [15]. If the device is still running old firmware and trying to reach that URL for updates, one could **hijack DNS** on the local network to point `ota.awair.is` to a local server and serve a crafted firmware file (our custom bootloader + firmware). However, this requires replicating Awair's firmware package format and is potentially complex (and only works if the device is still actively checking for updates). Awair's email suggested these older devices *"are no longer able to update firmware"*, possibly meaning the update mechanism was disabled or the server is unreachable [1]. So this approach may not be reliable. - **BLE Exploit in Setup Mode:** Another possibility is using the device's setup mode. When first configuring an Awair, the app communicates with it via BLE to send Wi-Fi credentials. If we reverse-engineer that BLE

protocol (perhaps from the Awair app or community), we might find a way to upload code or trigger the bootloader. Some IoT devices have hidden commands in setup mode. This would require significant reverse engineering and is uncertain.

Given these considerations, the **fastest path** to get started is to use the SWD method for at least one device (the user has a Glow C – they can flash that via a programmer). Once we have our custom bootloader and firmware running on Glow C, we can share the binary and instructions for others. Our BLE loader app can then be used by those who *do* open their device to flash the bootloader once. In the spirit of open-source and community, we will document this clearly (with photos of the PCB and where to connect SWD wires, etc.), so enthusiasts can replicate it. Over time, as the community grows, someone might discover a purely software way to flash the initial bootloader (which we can add to the guide if found). But initially, a one-time hardware flashing is the straightforward solution to "adopt" the device into the open firmware ecosystem [16].

Testing, Validation, and Iteration

Once the firmware and loader are developed, thorough **testing** is essential before public release:

- **Bench Testing:** Test the custom firmware on the Glow C hardware on a bench power supply. Simulate different conditions: varying temperature (use a heater/cooler), varying VOC (e.g., bring a VOC source like isopropyl alcohol nearby to see the readings change), and motion triggers (wave a hand in front of the PIR). Verify that the data published to MQTT or shown via REST reflects these changes correctly. Also test the relay by toggling it via an MQTT command or a REST endpoint (ensure the relay clicks and the connected device powers on/off).

- **Connectivity & Stability:** Connect the device to a test MQTT broker and Wi-Fi network. Run a burn-in test where the device publishes data continuously for several days. Monitor for any memory leaks or crashes (enable core debug logs via UART). Also test edge cases: Wi-Fi loss (turn off the router to confirm the device recovers or at least caches data), MQTT broker down, etc. The firmware should handle reconnecting gracefully.

- **Home Assistant Integration:** Set up Home Assistant with the device. For MQTT, ensure that all sensors appear and update. If using MQTT discovery, check that the entities are properly named and units (°C, % humidity, ppb VOC or index) are correct. If possible, compare the readings with a known reference or even the original Awair (if it still displays something) to ensure our values make sense. Minor calibration tweaks might be needed (e.g., the SGP30's eCO_2 might need baseline training – allow it to run ~24 hours to auto-calibrate as per Sensirion spec).

- **Firmware Loader Tests:** Test the BLE DFU process extensively. Use different devices: an iPhone, an M1 MacBook (with the Catalyst app), etc., to upload firmware. Intentionally test failure modes: e.g., disconnect in the middle of transfer (the bootloader should stay in DFU and allow retry), send a corrupted file (bootloader should reject it). Also test the provisioning flow: after flashing, send wrong Wi-Fi credentials to see if the device handles it (it should either revert to provisioning mode or indicate error – perhaps blink LED red). Then send correct credentials and verify it comes online.

- **Awair Element Testing:** Coordinate with a community member who has an Awair Element to test the firmware on that device. Since the user doesn't have an Element, this might happen in a later phase. Provide that tester with the bootloader and firmware binaries and instructions to flash (likely via SWD if they're willing to open it, or ship them a pre-flashed STM32 if they can solder, etc.). Once flashed, verify Element's CO_2 and PM readings appear and that the device runs stable. Adjust any sensor polling timings if needed (for example, the CO_2 sensor might require a certain warm-up time on power-up).

- **Fine-Tuning:** Gather feedback from initial testers. There may be environmental differences – for instance, SGP30 baseline may differ by region or if the device was unused for long. Provide a way for users to reset baseline or input their altitude for CO_2 calibration (Telaire sensors sometimes need altitude compensation). These can be simple config values set via a small config file or an MQTT command topic (for advanced users).

- **Power and Safety:** Although the devices are wall-powered, ensure the firmware doesn't accidentally toggle the relay erratically (which could power-cycle connected appliances rapidly). Implement a safety lockout (e.g., do not allow more than a certain number of toggles per minute via automation, or expose a parameter for it). Additionally, the Glow C's relay is rated for a certain load – we should clearly communicate to users to stay within the original device's electrical limits (as noted in Awair's documentation) [17] . The custom firmware should default the relay to OFF on boot for safety, until commanded otherwise.

Documentation and Community Release

With a working firmware and loader, the final step is to **release it to the community** and provide comprehensive documentation:

- **Open-Source Repository:** Create a GitHub repository (or similar) for the project. This will host:
- The firmware source code (organized by components: sensor drivers, connectivity, MQTT, etc.).
- The bootloader source code.
- The firmware binary releases (for those who just want to download and flash).
- The loader app source (if open-sourcing it) or at least the binaries for iOS (perhaps TestFlight or an App Store listing, if possible) and Mac. We might provide the Mac version as a direct download if not through App Store.

- Documentation files and diagrams.

- **User Guide:** Write a step-by-step guide (in Markdown and a wiki) for users to follow. This should include:

- Overview of what the custom firmware does and which Awair models it supports.
- Instructions for opening the device (with pictures referencing points from the Reddit advice, e.g. "Pry off the top plate" [18]).
- How to connect the programmer and flash the initial bootloader (if we assume that route). Provide STM32CubeProgrammer screenshots or OpenOCD commands for convenience. (If at release time we have an alternative method without opening, include that as well.)

- How to use the iOS/Mac loader app to flash updates and set Wi-Fi. Include screenshots of the app: scanning for device, selecting firmware, entering Wi-Fi info.
- How to add the device to Home Assistant (provide sample `configuration.yaml` for MQTT or mention the discovery mechanism).

- Troubleshooting section: LED signals (we can use the RGB LED to indicate statuses: e.g. blinking blue for waiting Wi-Fi, solid green for normal operation, red for error), what to do if something fails (e.g. how to factory reset our firmware if needed – perhaps holding the button for 10 seconds could clear Wi-Fi settings and reboot to provisioning mode, etc.).

- **Community Engagement:** Announce the project on forums like Home Assistant Community and Reddit (r/Awair) where users have been looking for solutions for these now-defunct devices. For example, one Hacker News commenter explicitly wished for an *"offline alternative"* to Awair's cloud dependency [19] [20]. Our project directly addresses that gap by providing an offline, user-owned solution. Encourage those interested to contribute – e.g., testing on Awair Glow (original) or Awair Omni if they have them, as the architecture might be similar and could be added to the firmware with minor tweaks.

- **Future Improvements:** Outline potential future enhancements to inspire community contributions:

- Integrating the Awair Element's LED display to show current readings or an icon without needing the app.
- Adding support for other "sunset" Awair models (like Awair v1 or Glow (original) – the Glow 2016 model monitored CO_2 [21], so it likely has a different sensor setup but perhaps a similar MCU).
- Refining the firmware to use ultra-low-power modes if anyone wants to use it on backup power (less relevant for always-plugged devices, but could be nice to minimize heat generation).
- Possibly supporting **Matter or Thread** in the future, if someone is ambitious, to integrate with modern smart home protocols – the STM32F412 has the capability to handle additional protocols if the community sees fit.

Finally, emphasize the benefits of this project: It extends the lifespan of the Awair Glow C and Element hardware **indefinitely**, free from cloud dependencies. Users regain ownership of their data and can integrate these sensors into any system they want. This not only prevents toxic e-waste but also sets a precedent for reusing IoT devices that manufacturers abandon. With this custom firmware, an Awair Glow C plugged into your wall becomes a fully local air quality node, publishing temperature, humidity, VOC, CO_2, and dust levels straight to your home automation – keeping your home safer and smarter without needing any proprietary services.

Sources:

- Awair's discontinuation notice for Glow/Glow C (device cloud support ended Nov 2022) [1] [22].
- Awair Element hardware info (sensors and module) [3] [7] and Glow C sensor details from Sensirion's announcement [6].
- Laird/STM32 module (Sterling-EWB) specifications [4] enabling Wi-Fi and BLE on these devices.
- Reddit community insights on device disassembly (opening Glow units) [9].
- Awair Glow C product description (features: VOC, temp, humidity sensors; smart plug and motion nightlight) [2].

1 19 20 22 Awair dropping support for Glow and Awair v1 devices | Hacker News
https://news.ycombinator.com/item?id=33253544

2 6 Awair’s Glow C air quality monitor entrusts Sensirion’s best-in-class environmental sensors | Elektor Magazine
https://www.elektormagazine.com/news/awair-s-glow-c-air-quality-monitor-entrusts-sensirion-s-best-in-class-environmental-sensors

3 10 Awair Element inside - Air Quality - Oleksandr Liutyi's WIKI
https://wiki.liutyi.info/display/CO2/Awair+Element+inside

4 11 13 img.eecart.com
https://img.eecart.com/dev/file/part/spec/20240118/fee61d3528cb4a2da3e85137b1bef84f.pdf

5 BITFINDER, . AWAIR GLOW AWAIR0PD2 FCC ID 2AF65AWAIR0PD2
https://fccid.io/2AF65AWAIR0PD2

7 8 AWAIR Element – liutyi.info
https://liutyi.info/awair-element/

9 16 18 Glow C hacking : r/Awair
https://www.reddit.com/r/Awair/comments/11riz7b/glow_c_hacking/

12 Sterling™-EWB IoT Module - Ezurio
https://www.ezurio.com/wireless-modules/wifi-modules-bluetooth/sterling-ewb-iot-module?srsltid=AfmBOoryrDtRSliGvz3q1qMgycRzvSA3pkRgtzqYhuiCVcF8pZT2ZOUb

14 15 Awair Element | tiefpunkt’s notes
http://notes.tiefpunkt.com/hacking/awair-element.html

17 Glow-C Maintenance and Safety - Awair Support
https://support.getawair.com/hc/en-us/articles/360030518054-Glow-C-Maintenance-and-Safety

21 What is Awair Glow?
https://support.getawair.com/hc/en-us/articles/360025380993-What-is-Awair-Glow