This report provides a factual update on the current state of our investigation into the **Rebble Flutter application** and **Micropebble**. Its purpose is to establish a shared understanding of our findings and use them as a foundation for deciding the best path forward.

Since our first meeting, the Pebble / Core ecosystem has evolved significantly. Our goal has been to remain impartial with respect to third-party directions, focus on technical feasibility, and objectively assess the available options before determining next steps.

Below, we summarize our findings in more detail.

Micropebble (KMM)

We began our work from this repository and focused on understanding the complexity of compiling and running the codebase on iOS.

As expected, since **libpebble3** is a **KMM-based library**, this approach provided the highest degree of compatibility between the shared logic and the presentation layer.

While this reduces architectural risk, we were already aware that **KMM support on Android is significantly more mature than on iOS**. As a result, the primary limitations we encountered were related to **platform parity in the UI**, particularly when using **Kompose**.

To address these issues, we lightly patched the `libpebble3` fork we started from and introduced a few native layers to bridge gaps where KMM-based libraries were missing (for example, networking).

As a result of this work, we now have a reference implementation, available in the GitHub repository we used during development.

Pros

- A single, unified codebase across platforms
- Significantly simpler architecture
- Lower complexity and barrier to contribution for the community

Cons

- UI cannot be fully shared between iOS and Android
- Some required libraries are missing in KMM and need native iOS counterparts
- The full presentation layer still needs to be implemented

Rebble (Flutter) + libpebble3

The second part of our investigation focused on evaluating the effort and complexity of integrating **libpebble3** into the existing **Rebble Mobile** Flutter application, starting from the official repository:
https://github.com/pebble-dev/mobile-app

Our first goal was to bridge `libpebble3` into Flutter and verify basic functionality. We successfully managed to scan for Pebble devices and establish a connection.

However, this approach introduced several challenges.

The first major issue was compiling **libpebble3 as an iOS framework**. This required additional tooling, including automated build scripts, to reliably test and integrate the library.

To make this work, we needed to:

1. Build the KMM framework
2. Run **Pigeon** to generate the Flutter bridge
3. Adapt the Flutter-side implementation

Each change to the KMM code required a **full rebuild**, taking up to **three minutes per iteration**, followed by regeneration of bindings and Flutter-side adjustments. This created a **three-stage failure point**, significantly slowing development. We also encountered suboptimal caching behavior, which further impacted iteration speed.

Additional code changes were required, after which we began bridging functionality using Pigeon. Once the bridge was in place, we had to heavily rework a dedicated file (`LibPebble3FlutterBridge.swift`) to ensure correct behavior.

At this stage, the application can successfully:

- Scan for Pebble watches on iOS
- Connect to devices
- Use `libpebble3` via the compiled framework

Although out of scope, we also investigated the **Android counterpart** to better understand the full cross-platform implications, where we encountered a different but comparable set of challenges.

As with the Micropebble PoC, we provide a reference to the repository used during this work.

Pros

- A partially implemented presentation layer already exists

Cons

- `libpebble3` integration introduces too many abstraction layers
- Debugging a compiled library is significantly harder than native or KMM code
- Flutter Blue Plus appears more mature for BLE but would require additional effort to integrate
- High barrier to entry for community contributors

Our Guidelines and Recommendation

After reviewing both PoCs and the challenges encountered, we met internally to determine the most suitable next step for delivering a mobile application for the Rebble community.

Our primary objective is to:

- Deliver an MVP capable of consuming `libpebble3`
- Avoid unnecessary architectural complexity
- Enable **any developer** to contribute without facing significant technical barriers
- Support long-term community maintenance as an open-source project

We also evaluated the **time-to-feature tradeoff**, comparing the benefits of reusing existing presentation code versus starting fresh.

While starting from the existing Rebble Flutter codebase provides an initial UI foundation, we believe this benefit does **not** outweigh the long-term cost of maintaining a **three-layer architecture** to integrate `libpebble3`.

The overhead of building, debugging, and iterating is high. Even assuming `libpebble3` itself is stable, deeper debugging would likely require additional tooling or headless builds. Flutter, in this context, risks becoming a blind spot and a recurring source of friction—as already experienced during our PoC.

To minimize complexity and lower the barrier to entry, we believe the most sustainable solution is a **full KMP-based application**, with:

- A two-layer architecture
- Native bridging only where iOS parity is missing
- Full support for debugging and breakpoints

We acknowledge that this approach requires building the **entire presentation layer from scratch**, but we are also aware that a UI redesign is already underway, making this a good opportunity to align efforts.

Additionally, the existing Flutter codebase relies on outdated state management and architectural patterns, which would likely require a significant rewrite in the near future anyway.

Conclusion

Both approaches have clear advantages and drawbacks. However, from a **long-term sustainability, maintainability, and community contribution** perspective, we believe the best strategic choice is to focus on a **full KMP implementation**, leveraging `libpebble3` directly and developing a modern, cross-platform UI on top of it.

This approach minimizes architectural complexity, improves debuggability, and creates a more welcoming foundation for the Rebble community to build upon.