MicroBadge Software Design Document

1. Introduction

1.1 Purpose

MicroBadge is a software application suite for the BBC micro:bit v2, designed as a digital conference badge. It serves both functional and social purposes; displaying the user's name, hosting small interactive demos, and offering contact sharing via NFC.

This project serves as a conversation starter and technical showcase during events like conferences, meetings, and interviews.

1.2 Scope

This document focuses exclusively on the software implementation of MicroBadge. It covers the architecture, data structures, behavior, and design choices used to implement the badge's app-switching system and core applications using Rust and the `embassy` async runtime.

1.3 Audience

This document is intended for: * Reviewers evaluating its design. * Recruiters or interviewers reviewing technical work. * Anyone trying to learn how to write Rust on an embedded platform.

2. System Overview

MicroBadge is an embedded application for the micro:bit v2. It uses the Embassy async runtime to manage multiple cooperative tasks without a traditional RTOS. The system is modular and consists of an app switcher, an LED display task, button listeners, and multiple interactive apps.

2.1 Runtime and Concurrency

MicroBadge uses Embassy's async executor. It runs the following tasks:

- `display_task` – Consumes frame buffers and drives the LED matrix.
- `button_listener` – One per button (A, B, Start). Waits for input and debounces it before sending an event.
- `app_task` – Runs the currently selected app. Allows apps to yield and re-enter on each loop.

All communication is channel-based using `embassy_sync::channel::Channel`.

[Task UML][./uml/tasks.png]

2.2 App Switcher

The `Switcher` manages app selection and transition. It displays a menu and uses the A, B, and Start buttons to navigate between apps.

Each app implements a shared `App` trait with an async `run()` method. Apps are isolated and run cooperatively, returning control when done.

Current apps:

- *Menu.* The top-level app that allows selecting from installed apps.
- *Badge.* Scrolls a string (e.g. your name) across the LED matrix.
- *Snake.* A basic snake game with food, direction control, and score.
- *NFC Card* (in development). Will present contact info via NFC.

[Switcher UML][./uml/switcher.png]

2.3 Input System

Each button is handled by a separate `button_listener` task. When a button is pressed, it sends a `Button` enum into a shared channel.

Apps listen for button input using the receiver end of the channel.

- A and B buttons are mapped to actions like turn left and right.
- Start is used to confirm or start an app. It is mapped to the capacitive touch sensor logo.
- A debounce delay of 100 ms is used for stability.

2.4 Rendering System

The rendering system uses a frame buffer that is written by the active app and read by the `display_task`.

Apps write into this buffer using a `Renderer` abstraction. Drawing is done in an offscreen buffer that is later pushed to the display.

- The screen is a 5x5 LED grid.
- Per-frame updates allow for animations and dynamic content.
- LED brightness levels are supported.

2.5 Code Organization

The system is split into modules for clarity and reuse:

- `app`. Defines the `App` trait and shared app interface.
- `display`. Low-level display driver and LED control.
- `renderer`. Provides drawing primitives for apps.
- `channel`. Shared async channels for button and frame messages.
- `switcher`. App selection logic and switching behavior.
- `snake`, `menu`, `badge`. App implementations.

- `microbit`. Definitions for button identifiers and device pins.

Each module is self-contained and uses only the shared channels and traits for interaction.

3. Application Features

3.1 Name Scroller

- Scrolls a configured name across the LED display.
- Uses an async timer to advance frames.
- Simple input handling: Any Button returns to the menu.

3.2 Snake Game

- 5x5 LED grid snake game using a wrapped grid (`WrappedU8<0, 4>`).
- Buttons A and B turn the snake left/right.
- Food spawns randomly in empty grid cells.
- On collision with self, enters game-over state and displays score.

3.3 NFC Business Card (WIP)

- Intended to broadcast a vCard or custom URI over NFC.
- Plan to use the BLE softdevice on the chip.
- Currently under development.

4. System Architecture

This system uses Embassy's async runtime to coordinate application execution, hardware input, and rendering on the micro:bit v2 board. It is divided into distinct tasks: input listeners, a display task, and an app task.

The overall architecture is message-passing oriented. Input events and screen updates are communicated over embassy channels.

Application logic is encapsulated in independent modules conforming to a shared `App` trait. The Switcher manages the active app and transitions between them.

4.1 Components

- `main.rs`: Entry point. Spawns system tasks using Embassy.
- `Display`: Renders 5x5 LED frames from a channel receiver using PWM.
- `ButtonListener`: Listens for button presses and sends events via channel.
- `Switcher`: Manages app lifecycle and transitions.
- `App`: Trait for any runnable application module.
- `menu`, `badge`, `snake`, `nfc`: App implementations.

5. Data Structures and State

5.1 Position, Direction, and Snake Body

The board is a fixed 5×5 grid. Positions are stored using a custom `Position` struct, which holds a `ClampedU8` for both `x` and `y` axes, ensuring values remain within bounds.

- `Position`: Represents a coordinate on the board with safe bounds.
- `Direction`: Enum for movement direction: Up, Down, Left, Right.
- `Snake`: Maintains a list of `Position` elements representing the snake's body. The first item is always the head.

Snake direction is updated via input, and movement wraps to stay within the board.

5.2 Message-Passing and Input State

User input is handled asynchronously via Embassy channels.

- Button presses are detected using `button_listener` tasks.
- Events are sent to the `ButtonChannel`.
- Applications read input non-blockingly using `try_receive()`.

This decouples physical input handling from application logic and allows clean, testable state transitions.

6. Component Interactions

6.1 How Components Interact Over Time

At runtime, three core tasks are running:

- `display_task`: Receives rendered frames and presents them on the display.
- `button_listener`: Spawns three tasks, one per button (A, B, Start).
- `app_task`: Owns the app switcher and runs the current app.

All interactions are asynchronous and use message-passing over embassy channels.

6.2 Flow of Control

1. User presses a button.
2. The button task sends a message to the channel.
3. The app reads the button event from the channel.
4. The app updates internal state (e.g., direction or selection).
5. The app prepares a frame and sends it to the frame channel.
6. The display task renders the frame.

This loop repeats, giving a responsive, concurrent embedded UI.

7. Development Environment

7.1 Rust + Embassy

This project uses Rust with the `embassy` async runtime. It provides interrupt-driven, non-blocking execution suitable for low-power embedded devices.

7.2 Tools

- `probe-rs`: For flashing and debugging firmware.
- `defmt`: Lightweight logging for embedded targets.
- `panic-probe`: Panic handler integrated with defmt output.
- `cargo-embed`: For development workflow and flashing.

Development was done on Linux using vim and CLI tooling.

8. Design Decisions

8.1 Why Embassy

Embassy was chosen for its async-first architecture, which maps well to reactive, event-driven embedded applications like games and UI. It allows multiple concurrent tasks without needing an RTOS or blocking code.

8.2 Fixed Board Size

The micro:bit's 5×5 LED matrix is inherently fixed. Game logic and rendering are simplified by using a constant-size grid, avoiding the need for dynamic allocation or scaling logic.

8.3 Data Wrapping and Clamping

Out-of-bounds positions are prevented using custom `ClampedU8` types. These provide safe arithmetic that prevents overflow and keeps all positions within 0–4 inclusive. This reduces bugs and runtime checks in critical loops.

9. Future Work

9.1 NFC Business Card App

An in-progress app will emulate a contact card via NFC. The goal is to allow devices to scan the badge and receive contact information, a URL, or a vCard.

9.3 UI Polish