REPUBLIC OF TURKEY
YILDIZ TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

DESIGN AND DEVELOPMENT OF A MULTI-PURPOSED SMARTWATCH

19011091 — Umut Can Sevdi

SENIOR PROJECT

Advisor
Assoc. Prof. Mehmet Amac Guvensan

January, 2024

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

GPIO	General Purpose Input Output
I2C	Inter-Integrated Circuit
IPS	In Plane Switching
LCD	Liquid-crystal display
RPI	Raspberry Pi
SDK	Software Development Kit
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Design and Development of a Multi-Purposed Smartwatch

Umut Can Sevdi

Department of Computer Engineering
Senior Project

Advisor: Assoc. Prof. Mehmet Amac Guvensan

Wearable technologies are becoming increasingly important in today's world. Among them, smartwatches are the most widely accepted technology among consumers. Within the scope of our project, we aim to develop a smartwatch and an associated Android application that communicates with it. The smartwatch is designed as an embedded system without a full-fledged operating system. In addition to basic clock functions such as time, alarm, and stopwatch, the smartwatch will have features such as receiving notifications, handling calls, and controlling the media player on the phone. The touchscreen-operated watch will also include an accelerometer, a buzzer for alarm sounds, and a motor for vibration. On the mobile side, there will be a mobile application that can respond to requests and messages sent by the smartwatch via Bluetooth. The application will allow users to set alarms on the smartwatch and will manage background services such as notifications, calls, and media controls using Android services.

Keywords: Embedded Systems, Touch Screen, C, Raspberry Pi Pico, Smartwatch, Android, Flutter, Dart, Bluetooth

ÖZET

Çok İşlevli Akıllı Saat Tasarımı ve Geliştirilmesi

Umut Can Sevdi

Bilgisayar Mühendisliği Bölümü
Bitirme Projesi

Danışman: Doç. Dr. Mehmet Amaç Güvensan

Giyilebilir teknolojiler günümüzde giderek önem kazanan bir konudur. Bunlardan tüketiciler arasında en çok kabul gören teknoloji ise akıllı saatlerdir. Projemiz kapsamında bir akıllı saat ve bununla iletişim halinde olan bir Android uygulaması geliştirmeyi hedeflemekteyiz. Akıllı saat işletim sistemi olmayan bir gömülü sistem olarak tasarlanmaktadır. Akıllı saatte temel saat fonksiyonları olan saat, alarm, kronometre gibi özelliklere ek olarak; telefondaki bildirimleri, aramaları ve medya oynatıcısını kontrol edebilmesi gibi özellikler bulunacaktır. Dokunmatik ekranla çalışan saatte buna ek olarak ivmeölçer, alarm sesleri için buzzer ve titreşim için motor bulunacaktır. Mobil tarafında ise saatin Bluetooth üzerinden gönderdiği istek ve cevaplara karşılık verebilen bir mobil uygulama buluncaktır. Uygulamaladan saatin alarmları ayarlanabilecektir. Buna ek olarak arka planda Android servislerini kullanarak bildirim, arama, medya araçları gibi servisleri yönetecektir.

Anahtar Kelimeler: Gömülü Sistemler, Dokunmatik Ekran, C, Raspberry Pi Pico, Akıllı Saat, Android, Flutter, Dart, Bluetooth

1
Introduction

With the rise of IoT and wearable technologies, the modern world has become more integrated. We see "constantly connected" products such as glasses, watches, fridges and other household appliances.

In recent years, smartwatches have seen massive customer acceptance and have become popular products. They are not a replacement for phones but a complement to them.

These compact, intelligent devices offer various features that go beyond simple timekeeping. From fitness tracking to communication, smartwatches have evolved into multifunctional gadgets that enhance our connectivity, productivity, and overall well-being.

Most smartwatches currently in the market often provide the following features besides the basic smartwatch features.

1. Notifications and Communication
2. App Integration
3. Fitness Tracking and Health Monitoring
4. Voice Control and Virtual Assistants
5. Water Resistance

1.1 Market Analysis

The global smartwatch market size is projected to grow from $29.31 billion in 2023 to $77.22 billion by 2030, at a CAGR of 14.84%[1]. The COVID-19 pandemic increased health consciousness among consumers who became more cautious and emphasised

the maintenance of their well-being[1]—this increased demand for monitoring tools for the body's health.

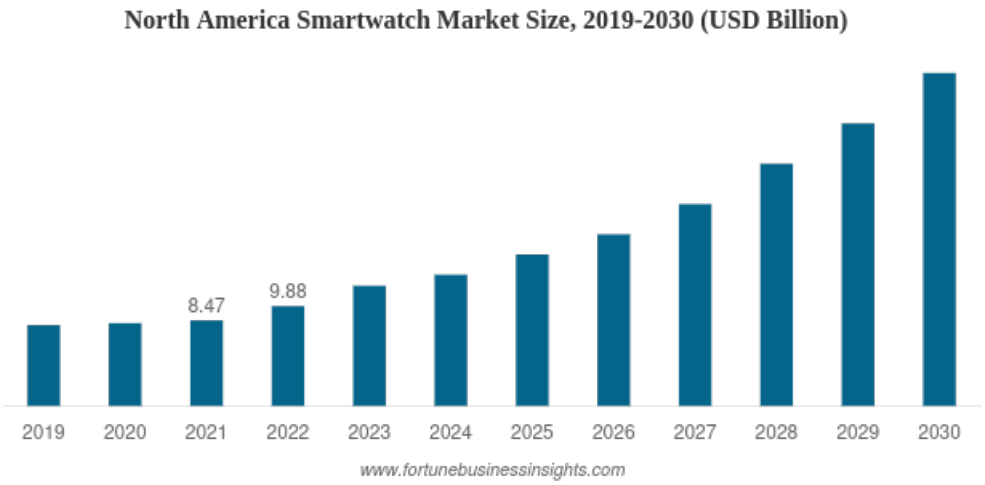

Figure 1.1 North America Smartwatch Market Size[1]

As of 2024, the market size is expected to be around 171 Million units[2], where the most prominent market is North America. According to the market analysis report at Mordor Intelligence, the top-selling brands are[3]:

1. Apple Inc.
2. Samsung Electronics Co. Ltd
3. Fitbit, Inc.
4. Garmin Ltd.
5. Fossil Group, Inc.
6. Huami Co., Ltd.
7. Huawei Technologies Co., Ltd.

1.2 Feature Comparision[4]

Most Smartwatches in the market use a simplified version of existing mobile operating systems. Let us compare top-selling products feature-wise from different brands:

1.2.1 Apple Watch 9

- 18 hours Battery Life
- GPS Support
- Water-Proof
- Fitness Tracker with Heart-Rate Monitoring
- Voice Assistant Support(Siri)
- Extra services such as Find My Phone
- Supports messaging(via Speech to text)

1.2.2 Samsung Galaxy Watch 6

- 40 hours Battery Life
- 16GB Storage
- Sleep Monitoring
- Fitness Tracker
- Camera
- Android Support

1.2.3 Fitbit Sense 2

- 6 Days Battery Life
- GPS
- Voice Assistant Support(Alexa)
- Fitness Tracker
- Android Support

2
Overview

This project aims to build a smartwatch from scratch and an Android application to manage it. The smartwatch is designed as an embedded system without a full-fledged operating system. The smartwatch contains the following services services:

1. Standard Watch Features:
   (a) Stopwatch
   (b) Alarm
   (c) Clock
2. Bluetooth-related services
   (a) Notification Management
   (b) Call Management
   (c) Event Reminder
   (d) Audio Management
3. Temperature
4. Pedometer
5. Calendar
6. Notepad
7. Multitasking

The project consists of The embedded system and the Android application. Roadmaps of each section are described below.

2.1 Embedded System

For the microcontroller, the Raspberry Pi Pico W[5] is selected. It is a powerful, lightweight and power-efficient microcontroller. Despite not having an operating system, it provides multi-threaded programming, Bluetooth and wireless support.

The 1.28-inch Touch LCD from WaveShare[6] is chosen for the touch screen for its rich gesture support.

The dedicated SDKs for the C programming language. Since they both support CMake[7], CMake was chosen as the build tool. Figma[8] is used for UI design. The 3D case model was built on Adobe Fusion.

1. Hardware Modules
2. Development of application modules
3. Bluetooth service and protocol design
4. Configuring connections to the Android device
5. Soldiering and minimising the size of the hardware and mounting it to a case

2.2 Android Application

On the Android side, the Flutter Framework for the Dart Programming Language[9] is selected for its ease of use and cross-platform capabilities. The mobile application provides various user interfaces to interact and configure the smartwatch. These features are as follows:

1. Add/Modify/Remove the alarms
2. A ToDo list for daily tasks with a reminder feature.
3. A step counter UI to show the daily steps.

3
Feasibility Analysis

This section discusses the feasibility studies conducted to outline the general aspects of the project. Research has been conducted to design the project appropriately, and technical preparations have begun. Design decisions to facilitate usability have been determined before the start of the project.

3.1 Technical Feasibility

The technical feasibility section is explained in both hardware and software subsections.

3.1.1 Hardware Feasibility

1. **Raspberry Pi Pico W**[5] A powerful, lightweight and power-efficient microcontroller that was built by the Raspberry Pi Foundation.

Figure 3.1 Raspberry Pi Pico W

It is also a cost-efficient product, which is replaceable when facing a hardware issue. The table 3.1 shows the device's technical specifications.

Hardware	Specifications
CPU	Up to 133 MHz Dual Core Arm Cortex M0+ Processor
SRAM	264kB
Flash Memory	2MB
IO	USB 1.1
	2×SPI
	2×I2C
	2×UART
	3×12-bit ADC
	16×controllable PWM
	26×GPIO pins
Extras	Accurate clock and timer
	Temperature sensor
	Built-in Wireless and Bluetooth support

Table 3.1 Pico W Features

2. **1.28inch Touch LCD** from WaveShare[6] is a circular LCD touch screen with a SDK. SDK provides various features such as gesture capture, pixel capture, buffer swapping, and a basic font library with examples.

Figure 3.2 Waveshare 1.28 Touch LCD

The table 3.2 shows the device's technical specifications.

Hardware	Specifications
Screen	240 × 240 px LCD Screen
Touch Screen	CST816S Capacitive Touch controller
Display	IPS display panel

Table 3.2 Waveshare 1.28 LCD Features

3. **Arduino TP4056 and Power-Xtra PX 302030 3.7V 120mAh LiPo Battery** Arduino TP4056 is a Protected LiPo Battery Charging Circuit that can be charged via a Micro-USB cable. Power-Xtra PX 302030 3.7V 120mAh LiPo is a capable LiPo battery that will be enough to power all smartwatch components.

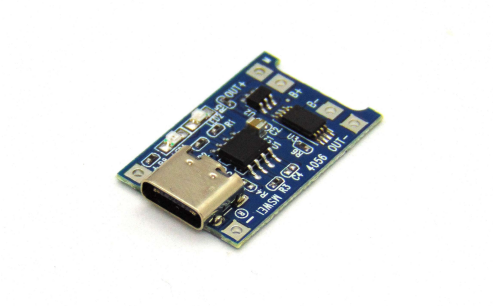

Figure 3.3 Arduino TP4056 Protected Lipo Battery Charging Circuit

4. **HC06 Bluetooth Module** is an external Bluetooth module commonly used in the industry for its simplicity. It is a slave Bluetooth device that transfers and receives data using the UART protocol. It is added later to the project due to the issues faced with the Pico's built-in Bluetooth stack.

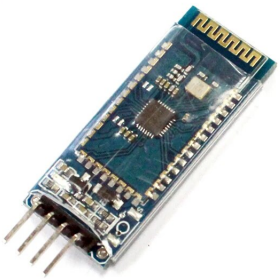

Figure 3.4 HC-06 Wireless Bluetooth Trans-Receiver

5. **MPU6050 Accelerometer** is an accelerometer commonly used with Arduino and Raspberry Pi Pico. MPU6050 provides various information such as acceleration, gyroscope and temperature data.

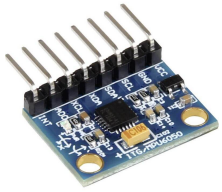

Figure 3.5 MPU6050 Accelerometer

6. **Other Hardware Components** The following products are also used in the project. But they do not have product-specific behaviours.

   (a) Piezo Buzzer

   (b) 5mm Yellow LED

   (c) 10x3 mm Shaftless Vibration Motor

3.1.2 Software Feasibility

All the software described below is Free under GPL or MIT Licenses.

1. **Build Tools:**

   - **Docker**[10] is an open-source containerisation tool for creating sandbox environments. Containerization is the process of packaging a piece of software with all the requirements for its operation. A container is provided with all the necessary dependencies. In the case of this project, the Docker is used as a compilation environment for the Arm64 systems. The Raspberry Pi Pico requires additional environment variables and compiler extensions.
   - **Raspberry Pi Pico SDK**[11] for RP2040-based devices, like the Raspberry Pi Pico, includes essential headers, libraries, and a build system for writing programs in C, C++, or Arm assembly language. The SDK is crafted to offer a programming environment and API that is accessible to both non-embedded C developers and those experienced in embedded C. SDK provides features such as timers, fixed-function peripherals, timers, multi-core programming and interrupt handling.
   - **WaveShare Graphical Library**[12] for the Touch LCD provides high-level abstractions to the SPI and I2C communications between the devices. The library provides basic display functionalities such as drawing points, lines and rectangles, and clearing screens. In addition to display features, it provides gesture and pixel capturing as input.
   - **GNU Utilities**[13] such as GCC is used for compiling the C program.
   - **CMake**[7] is a cross-platform build system that simplifies generating build files for various development environments. Its modular and extensible design enables the integration of third-party libraries and tools, fostering collaboration and code reuse. With its widespread adoption in open-source and commercial projects, CMake has become a cornerstone in modern

software development, promoting consistency and ease of use in the build process. Raspberry Pi Pico SDK and the Waveshare Library use CMake as the build tool.

- **Flutter** [9] is an open-source UI toolkit developed by Google for building natively compiled applications across mobile, web, and desktop platforms using a single codebase. It enables developers to create visually appealing and consistent user interfaces with a rich set of pre-designed widgets. As a versatile solution, Flutter is popular for creating responsive and visually appealing applications with reduced development time and effort. Flutter is used to build the mobile application.
- **Android SDK** [14] is a set of tools and resources provided by Google to enable developers to create applications for the Android operating system. It includes a comprehensive set of libraries, APIs, and debugging tools, allowing developers to build, test, and debug Android apps efficiently. The SDK provides support for various features such as user interfaces, networking, multimedia, and device hardware integration. Android SDK also includes the Android Emulator, which allows developers to test their applications on virtual devices with different configurations. The SDK is used to produce an Android binary.

2. **Developer Software:**

   (a) **Neovim** is a modern and highly extensible text editor designed as an improvement and extension of the traditional Vim editor. It retains Vim's powerful modal editing system, where the editor has different modes for navigation, insertion, and command execution. Neovim, however, introduces additional features and enhancements to address some limitations in Vim, such as a built-in plugin system, LSP support, and multi-threaded I/O. Neovim is the development environment I've used for the C programming language.

   (b) **Tmux**, short for "terminal multiplexer," is a command-line tool that enables users to manage multiple terminal sessions within a single terminal window.

   (c) Android Studio [14] is the official IDE for Android app development. It is built on the IntelliJ IDEA platform and offers a comprehensive set of tools for designing, developing, testing, and debugging Android applications. Developers can write code in languages like Java, Kotlin or Dart, and Android Studio helps manage the project structure, dependencies, and resources efficiently. It also integrates with the Android SDK, allowing

developers to access a vast array of libraries and tools.

(d) Figma [8] Figma is a cloud-based design collaboration tool that enables multiple users to work on design projects in real time. It provides a seamless platform for creating, prototyping, and iterating user interfaces, fostering efficient collaboration among designers and stakeholders. Figma is used to create the User Interface sketches and components used in the embedded system.

3.2 Workforce and Time Planning

The completion process of the project has been estimated to take approximately 90 days. Initially, a meeting was held with our advisor teacher to discuss the expected features of the system, and a plan was created accordingly.

Subsequently, the hardware components were determined and procured. The parts were connected, making them ready for development.

1. On 8/12/2023, the smartwatch was shown to the advisor.
2. On 04/01/2024, the development of the smartwatch was completed.
3. On 02/01/2024, the development of the Android application started.

3.3 Legal Feasibility

All the software described above is Free under GPL or MIT Licenses. They are permitted to be used under the given product.

3.4 Economical Feasibility

3.4.1 Hardware Costs

The hardware cost of each module is shown in the table3.3 below. Prices are as of 10/2023 and may change over time.

Hardware	Cost (TL)
Raspberry Pi Pico W	140
WaveShare 1.28 Touch LCD	529
HC06 Bluetooth Module	197
eSUN 1.75 mm White Pla+ Filament	587
Arduino TP4056 Protected Lithium LiPo Battery Charging Circuit Micro USB	52
Power-Xtra PX 302030 3.7V 120mAh LiPo Battery	88
10x3 mm Shaftless Vibration Motor	15
5mm Yellow LED	1
Total	**1609**

Table 3.3 Hardware Costs

3.4.2 Development Costs

From the project's announcement, approximately 14-16 hours per week have been spent on the project until its release. (14 16)×8 weeks of development makes 112 128 hours of development. Assuming a work day for a software engineer(man-day) is 8 hours, it would take almost a month of salary to complete the project. Since a junior software engineer in Turkey would make 30.000-35.000TL as of 2023, this would be the approximate development cost.

4
System Analysis

The smartwatch project consists of two sections. The system that runs on the embedded device and the Android application.

4.1 Build

The embedded system is written in C using a CMake build system with a modular system design.

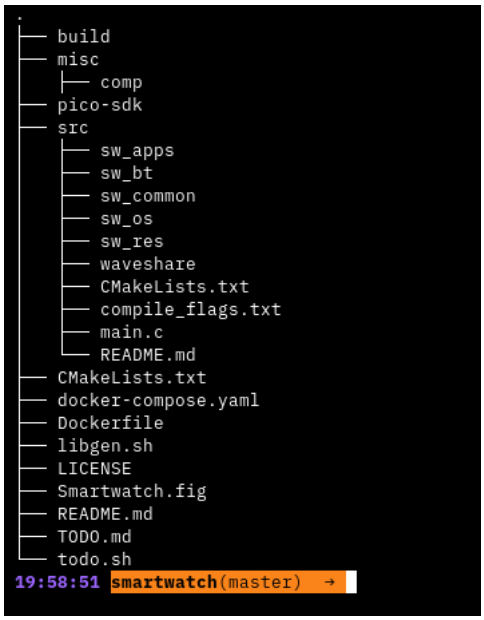

Figure 4.1 File Structure

4.1.1 Installation

Run the following script in a Linux environment with docker and docker-compose installed.

```
docker-compose up -d && docker exec -it picobox bash
mkdir -p app/build
cd app/build
cmake ..
make
exit
sudo cp build/src/smartwatch.uf2 /media/$(whoami)/RPI-RP2/
```

4.2 Data Design

4.2.1 State Struct

The State is a singleton struct that serves as a container for consolidating diverse data about the smartwatch's State and functionalities.

It holds a DateTime type that describes the current date and time. __dt_timer is used to count using Pico's timer module. Flags such as show_sec and is_connected to manage display preferences and connectivity status. __last_connected tracks the last time a packet is received via Bluetooth. The popup and __popup_req enable the handling of user interface elements.

The dev sub-structure includes a GPIO pin stack for robust operation, temperature, accelerometer and gyroscope data. __step_timer facilitates step tracking.

Alarms are managed through an array (list) within the alarms sub-structure, with len denoting the number of alarms present.

A Chrono sub-structure handles chronometer functionality, while the media sub-structure manages media playback details, including the current song, artist, and playback status.

Lastly, a global step count (step) overviews the user's physical activity.

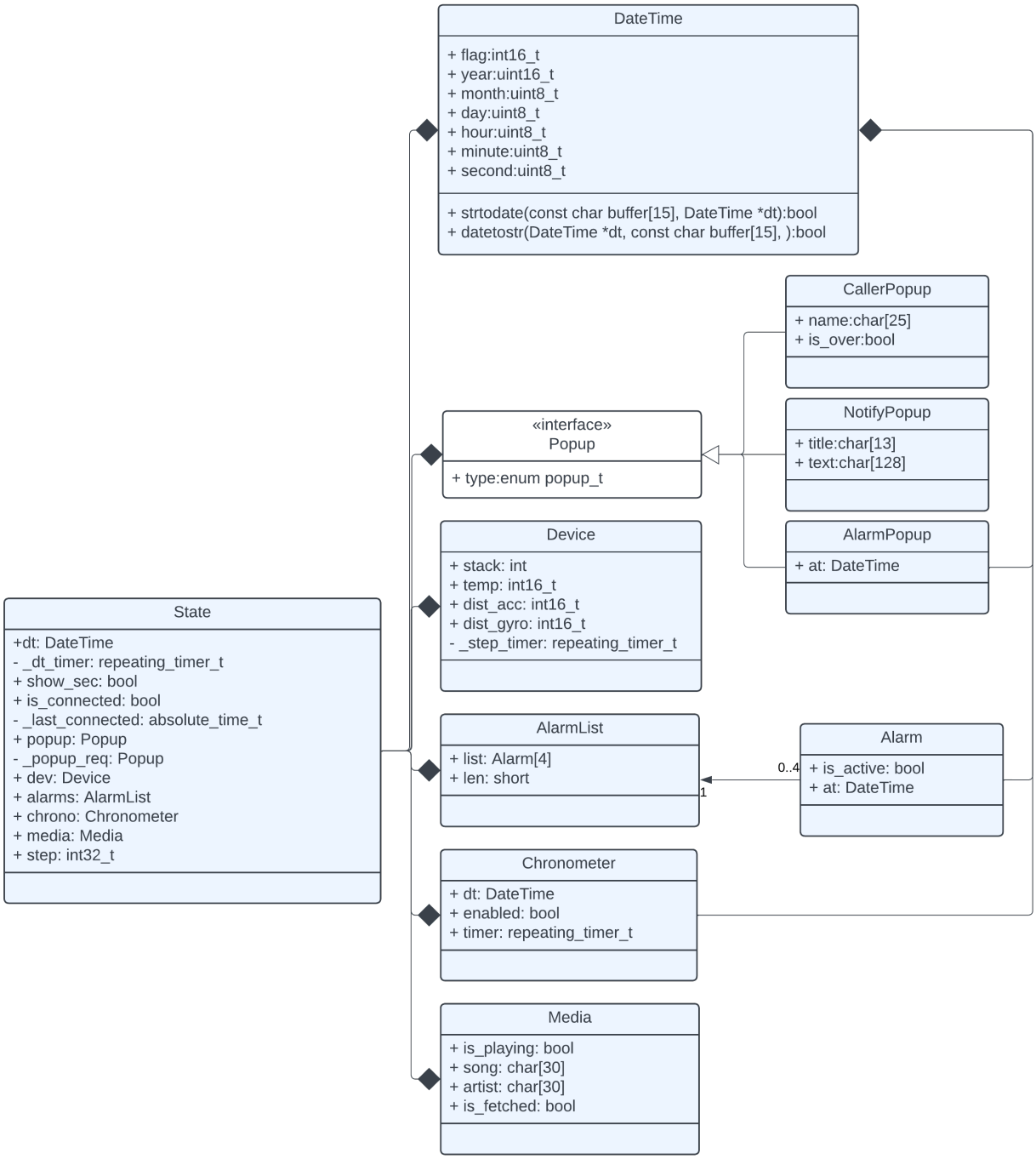

Figure 4.2 State Class Diagram

4.3 Use Cases

The following use case diagram describes the possible user scenarios that could be done on the smartwatch.

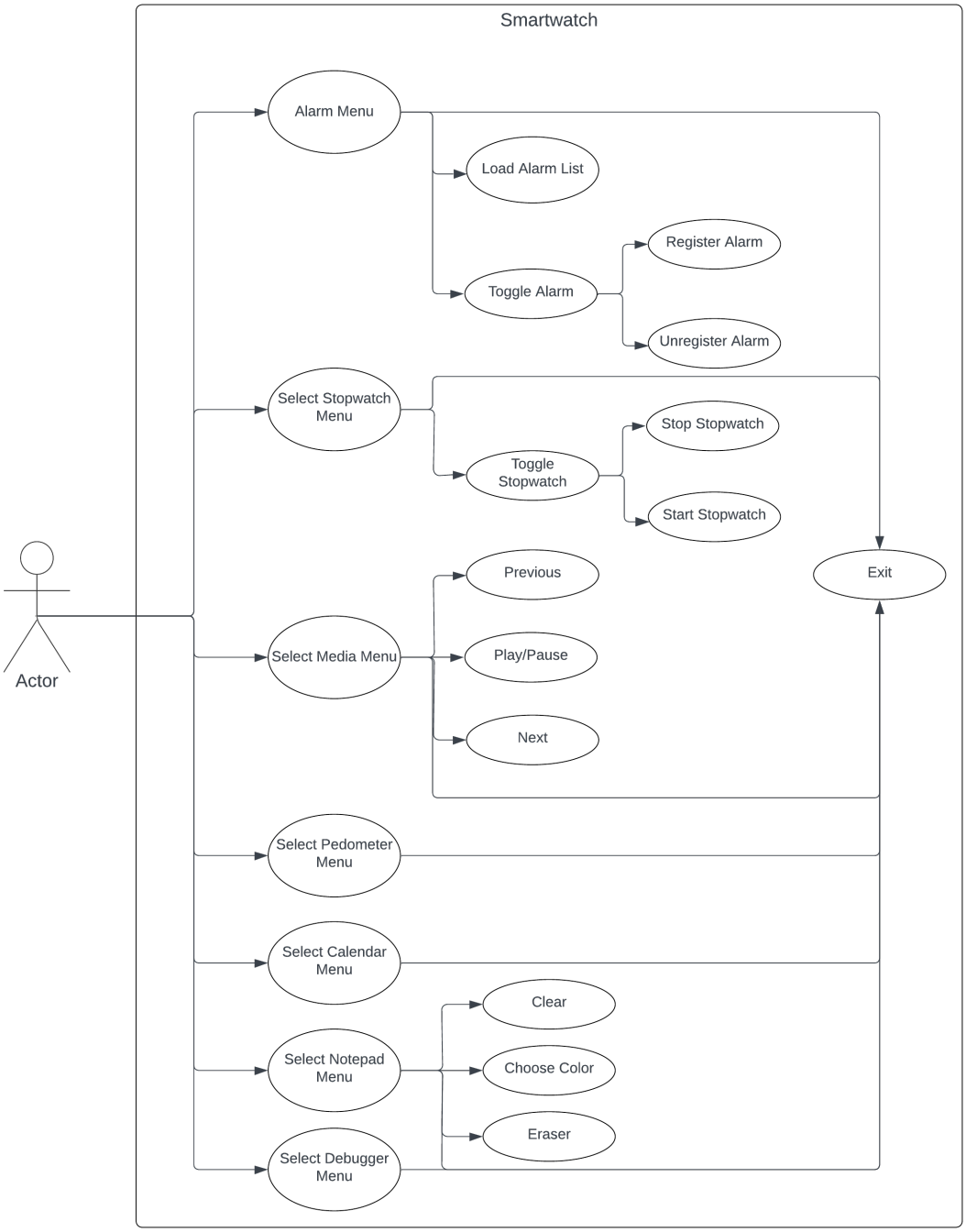

Figure 4.3 The Use Case Diagram

5
System Design

This chapter is divided into different sections to examine individually.

5.1 Hardware Design

All modules have been integrated into the circuit according to the pinout diagram. The scheme 5.1 displays the connections between Raspberry Pi and the rest of the modules.

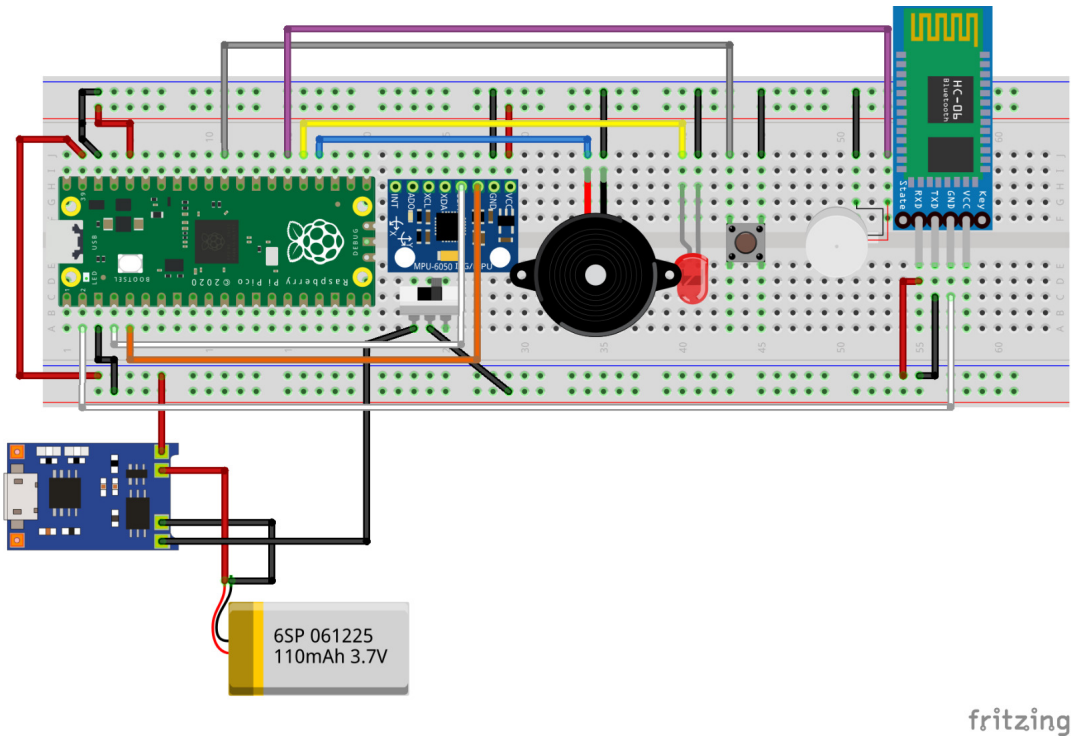

Figure 5.1 Connections on Fritzing

For the Waveshare LCD Module, the table 5.1 is used for the wire connections.

Raspberry Pi Pico	WaveShare Pin
3.3V	VCC
GND	GND
GP12	MISO
GP11	MOSI
GP10	SCLK
GP9	LCS_CS
GP14	LCS_DC
GP8	LCS_RST
GP15	LCS_BL
GP6	TP_SDA
GP7	TP_SCL
GP17	TP_INT
GP16	TP_RST

Table 5.1 Waveshare Pin Connections[15]

Figure 5.2 shows the circuit on a breadboard.

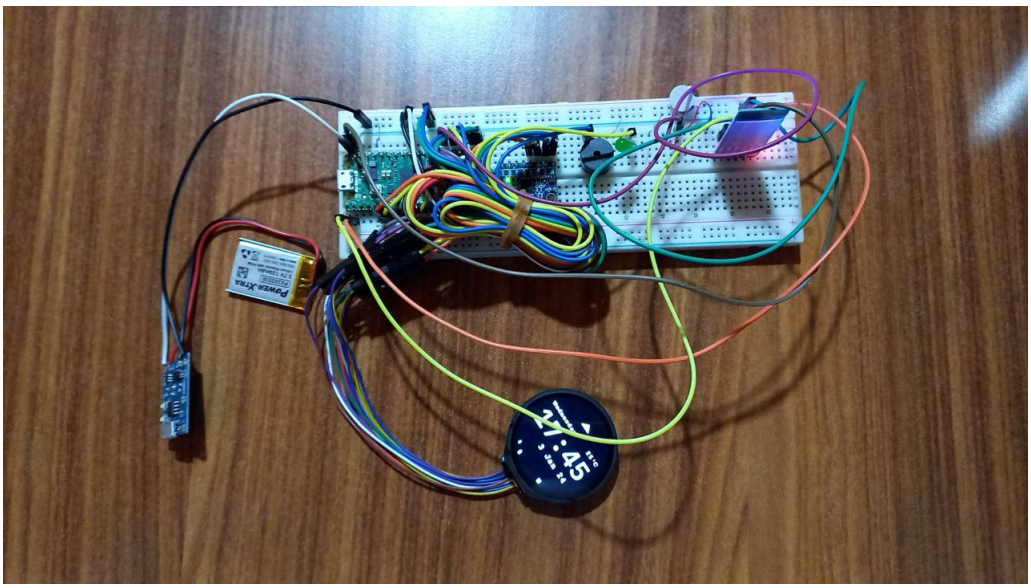

Figure 5.2 Smartwatch on the Breadboard

Figure 5.3 shows after the 3D printed case is installed.

Figure 5.3 Smartwatch with the Case

5.2 Software Design

This section explains the software design choices.

5.2.1 Folder Structure

The project is built modularly and has the following directory structure:

- **sw_apps/**: Contains the logic and display functions related to the Smartwatch's User Interfaces.
- **sw_bt/**: Bluetooth interface of the Smartwatch, which contains the communication protocol and its utility functions.
- **sw_common/**: Common data structures and functions shared among different modules.
- **sw_os/**: Stores functions and data about the global state and hardware abstractions.

- **sw_res/**: Contains the array representations of all fonts, images and miscellaneous items. *resources.h* declares helper functions.

5.2.2 Resources

The resources directory contains all images and fonts. All images for the user interfaces are developed on Figma. Images are then converted to JPG or PNG image formats and exported.

- **Waveshare Image Format**: Exported images had to be converted to byte arrays for our C program. The Waveshare SDK does not support standard image encodings and accepts only the following RGB formats: RGB444, RGB565 and RGB666.

  After the research, lvgl.io[16] was found suitable to convert images to the desired formats. Exported images are then moved under the sw_res directory.

- **Storage Limitations:** Originally, each screen was stored individually with a dedicated image for each possible scenario. This approach initially accelerated development but caused the project to hit the storage limit earlier than expected.

  Pico has 2MB of flash memory. After the competition of the media application, the disk size passed 3MB. Since the compiled binary was too big to fit into the Pico, and the compilation failed.

  RLE Encoding: The images contained a lot of replicated bytes. Initially, the RLE encoding algorithm was found to be fit to solve this problem.

  According to the solution, images would be stored and encoded to trim the binary size. Whenever an image is needed, it is decoded and displayed on the screen.

  The table 5.2 displays the disk usage of the source codes under the resources directory before and after the RLE encoding algorithm.

File	Before Encoding	After Encoding
alarm.c	1.2M	56K
decompress.c	8.0K	4.0K
font.c	856K	104K
media.c	2.0M	144K
menu_alarm.c	1016K	188K
menu_events.c	1016K	184K
menu_media.c	1016K	180K
menu_pedometer.c	1016K	212K
menu_stopwatch.c	1016K	208K
pedometer.c	1016K	80K
popup_alarm.c	1016K	80K
popup_call.c	1016K	84K
popup_notify.c	1016K	104K
stopwatch.c	2.0M	124K
tray.c	52K	52K
watch.c	1016K	4.0K

Table 5.2 Disk Sizes of the Resources

- **Memory Limit**: Raspberry Pi Pico W contains 264KB of memory. The display buffer and its backup buffer take up 112.5KB of memory, which leaves just enough space to encode and decode a single image. Although it might work in theory due to the **memory fragmentation** on the embedded devices, allocating and freeing arbitrary bytes of memory causes *malloc* to freeze the system altogether.

  Memory fragmentation is common in computer systems, including embedded devices like the Raspberry Pico. It refers to the situation where free memory is scattered in small, non-contiguous blocks, making it challenging to allocate large, contiguous chunks of memory. This fragmentation can lead to inefficient memory usage and reduced system performance. Eventually, it may cause the system to run out of memory even when there is technically enough free memory.

  Instead, stack memory and static memory allocation are suggested to prevent this issue.

- **Component(Sprite sheet) Based Design**: One of the biggest problems in the old design was that, despite removing the duplicate bytes during compile time, the memory usage was still higher when decoded.

The display system migrated to a component-based system UI programming to solve this. In a **Component Based UI**, the final image consists of various frequently used components, and them being drawn on top of each other.

Figure 5.4 Menu screen consists of multiple UI components

This approach significantly reduces the memory usage to store the images but costs extra CPU time since images are drawn individually.

- **Sprite sheet Based Design**: In old NES (Nintendo Entertainment System) and similar hardware, game developers often used sprite sheet animation to display animated characters and objects efficiently on the screen. The hardware at that time had limited graphical capabilities compared to modern systems, so developers had to be creative and optimise their use of resources.

Figure 5.5 Nintendo NES Console System

The character is stored in a single image file in the figure 5.6. The image contains multiple *sprites* with constant width and height. The code iterates over the images to quickly switch the buffer.

Figure 5.6 Spritesheet of the Megaman Character

- **Final Image Display**:

  Component and sprite sheet-based images reduced the file sizes so that they became compilable again. Furthermore, since the memory is saved to the static memory, it did not cause fragmentation.

Figure 5.7 Spritesheets used in the Smartwatch

The following code block displays how the Smartwatch captures the right image from the sprite sheets for the title bar and menu screen.

```
typedef struct {
    const unsigned char* img;
    size_t width;
```

```
    size_t height;
} Resource;

static inline Resource _get_sprite(int idx, const unsigned char* res,
                                   const int w, const int h)
{
    return (Resource){res + 2 * (idx)*w * h, w, h};
}

Resource res_get_titlebar(enum screen_t s_title, enum popup_t p_title)
{
    if ((p_title < 0 && p_title >= POPUP_T_SIZE)
        || (s_title < 0 && s_title >= SCREEN_T_SIZE))
        return (Resource){NULL, 0, 0};

    const int w = 160;
    const int h = 30;
    const unsigned char* img;
    if (p_title == POPUP_NONE)
        img = _res_titlebar + 2 * (s_title - 3) * w * h;
    else
        img = _res_titlebar + 2 * (SCREEN_T_SIZE - 3) * w * h
              + 2 * (p_title - 1) * w * h;
    return (Resource){img, w, h};
}

Resource res_get_menu_screen(enum menu_t selected)
{
    const int w = 160;
    const int h = 160;
    if (selected < 0 && selected >= MENU_T_SIZE)
      return (Resource){NULL, 0, 0};
    return _get_sprite(selected, _res_menu, w, h);
}
```

5.2.3 Event Scheduling

The Raspberry Pi Pico's CPU consists of two cores. The cores run independently on a shared memory and communicate over a FIFO queue.

- On Raspberry Pi Pico, when an interrupt occurs, by default, the interrupt handler will run on core 0[17].
- This creates a problem. When a background task such as a chronometer or a Bluetooth service runs, it may block the user interface or vice versa. To solve this problem, the user interface and all background tasks had to be separated. The user interface has been moved to the second core.
- The core one requires a function with the type of *void (*)(void) {}*.

```
static void _core1_cb()
{
    apps_load(SCREEN_CLOCK);
}

int main(int argc, char* argv[])
{
    stdio_init_all();
    os_init();
    apps_init();
    multicore_launch_core1(_core1_cb);

    bt_init();
    while (true) {
        /* ... */
    }
    return 0;
}
```

- The figure 5.8 displays the interrupts happening to both cores. The core 0 receives most of the interrupts and handles them. Due to the frequency of those interrupts, making a single-core application would cause the application to freeze.

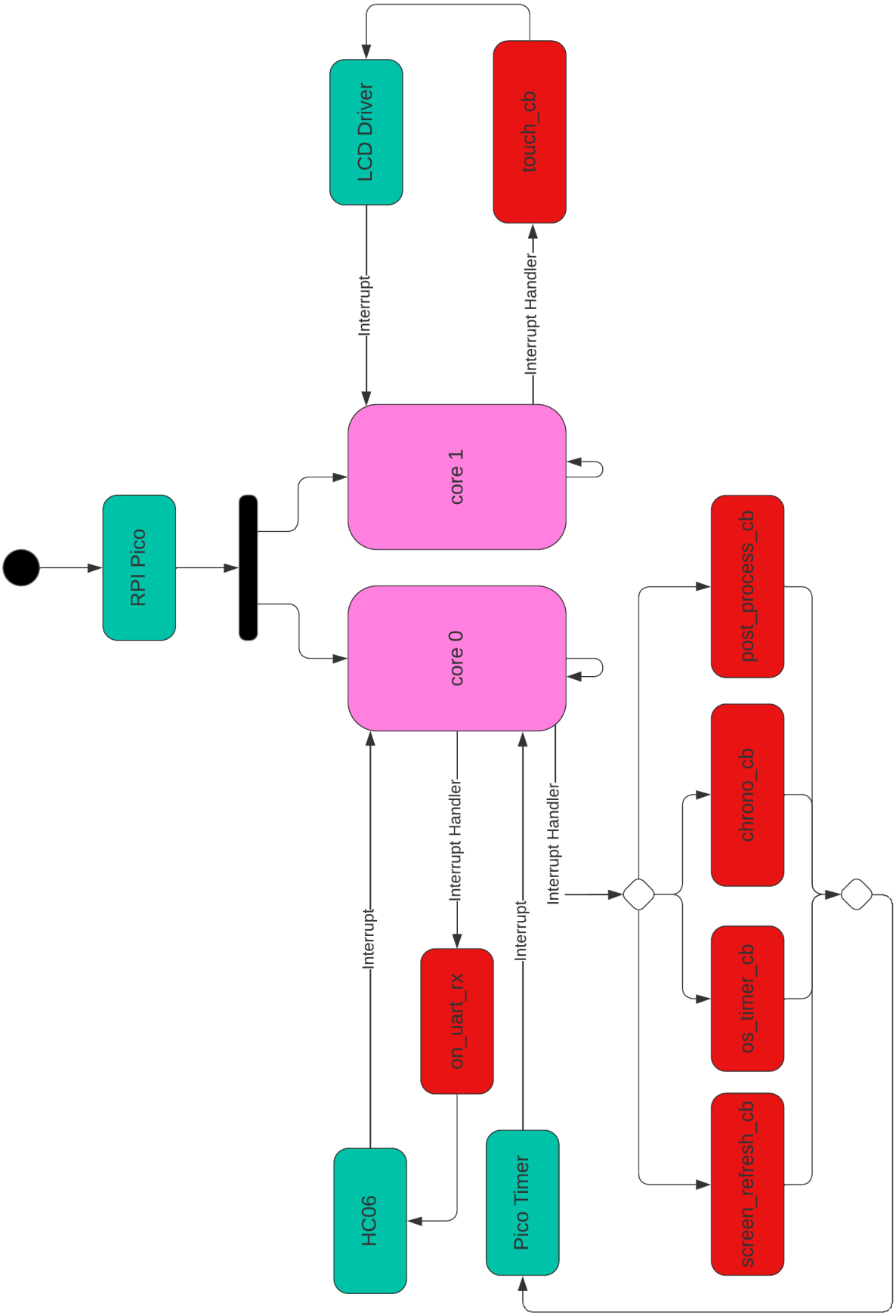

Figure 5.8 Activity Diagram of Interrupts and Background Tasks

5.2.4 The State

The state is a singleton object that stores the data throughout the watch's lifetime. State fields can be found in the figure 4.2. While different cores can independently write to and read from struct fields and their sub-structs, it is essential to note that only one core is responsible for writing a field at a time, ensuring exclusive access to prevent race conditions.

Note: While no hard lock mechanism prevents the race condition, no instance of access may cause such a condition.

5.2.5 The User Interface

The user interface starts by running the *apps_init(void)* function. This function initialises a Display struct instance.

1. **The Buffer:**

```
typedef struct {
    UWORD* buffer;
    UDOUBLE buffer_s;
    UWORD* canvas_buffer;
    bool is_saved;
    enum screen_t sstate;
    enum disp_t redraw;
    int post_time;
    repeating_timer_t __post_timer;
} Display;
```

   - The **buffer** holds the memory for the image being displayed. And **buffer_s** stores it's size.
   - **canvas_buffer** and **is_saved** are for the notepad application.
   - **sstate** stores the active screen.
   - **redraw** holds the data about the next redraw call.
   - **post_time** and **__post_timer** are for the timer to automatically trigger post_procesing.

   A screen can be initialised using the **enum app_status_t apps_load(enum screen_t)**. Before each screen or pop-up loads, the **apps_set_module** function is called, which initialises the touch type for the module type.

2. **The Display Scheduler:** The Display struct keeps track of the current state at any given time. A function call represents each screen state/application. They are stacked on top of each other using the function stack. Whenever a module function is called, the watch enters a new state; when a state ends, it returns its status. At any given time, the currently running module tracks three things:

   - Whether screen's *enum disp_t* is not *DISP_SYNC* or not.
   - Whether the type of the current *sstate* matches with the running function's state.
   - Whether a popup request has been attempted.

   If any of the conditions above are true, it recalculates the buffer and calls the drawing. This mechanism prevents unnecessary redraw calls.

   A redraw can be either *DISP_PARTIAL* or *DISP_REDRAW*. The entire buffer is redrawn on redraw, while only the text or buttons are partially updated.

   A full redraw will be called when a module returns since the current state is not equal to the existing screen.

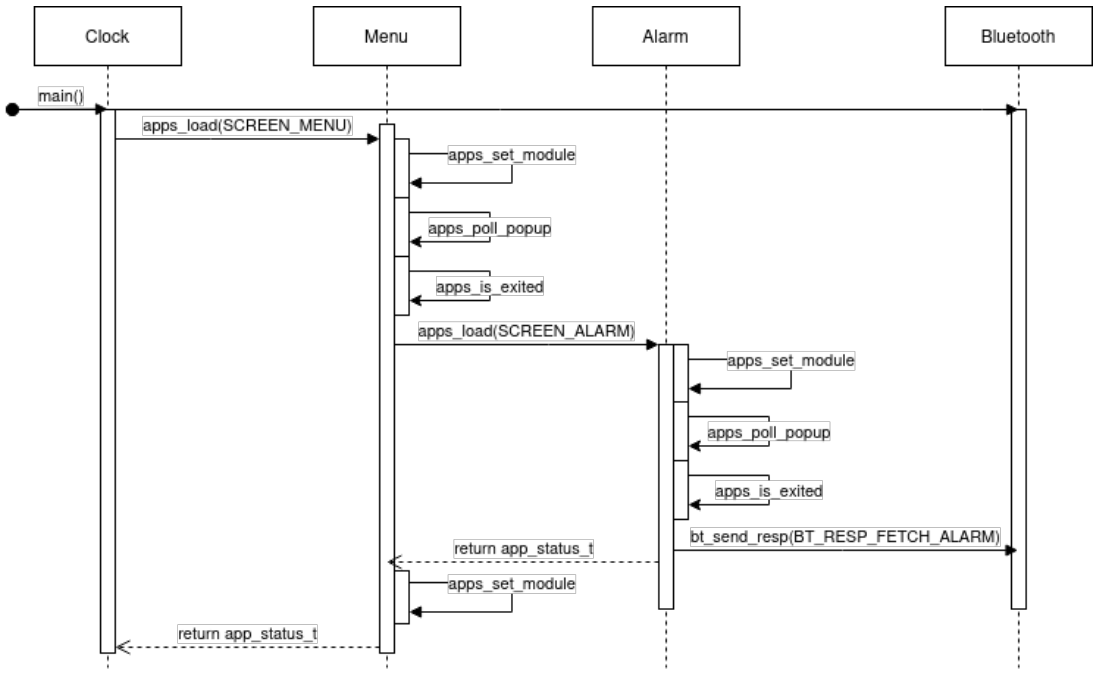

Figure 5.9 Sequence Diagram Of the Module

The sequence diagram 5.9 displays an example in which the user performs the following actions:

- The smartwatch boots, and the user opens the menu.
- User moves to alarm and clicks to open the Alarm application.

- User exits the Alarm.
- User exits the menu.

3. **Tray Processor**: When the Display Scheduler agrees on redraw after the frame is set, the tray postprocessor runs. This function places the tray icons according to the current state of the Smartwatch struct. If no post_process function is called for more than 30 seconds, the interrupt will trigger it automatically. That way, the clock at the top will be synchronised.

Figure 5.10 Tray Processing Example

4. **The Pedometer:** The pedometer is a repeating event that captures the current acceleration and temperature every 50 milliseconds. The square root of every call is inserted into an array. At every 20th call, all elements in the array are analyzed. In our case, the analysis algorithm has an arbitrary threshold, which is **STEP_THRESHOLD**. Anytime a value in the list passes the threshold, a flag is raised until it drops below the threshold again. When it drops, the flag is lowered. Then, the number of times the flag is raised is added to today's score.

   The code below is the pedometer's algorithm.

```
static void _step_count_analyze()
{
    bool peaked = false;
    int peak_count = 0;
    for (int i = 0; i < SAMPLE_SIZE; i++) {
        if (buffer[i] > STEP_THRESHOLD && !peaked) {
            peaked = true;
            peak_count++;
        } else if (buffer[i] < STEP_THRESHOLD && peaked) {
            peaked = false;
        }
```

```
    }
    if (peak_count / 2 > 0) { peak_count /= 2;
    } else { state.dev.step += peak_count; }
}

static bool _step_count_cb(UNUSED(repeating_timer_t* r))
{
    GyroData* data = &state.dev;
    _mpu6050_read_raw(data->acc, data->gyro, &data->temp);
    data->temp = (data->temp / 340.0) + 36.53;
    buffer[cursor++] = sqrt(pow(data->acc[0], 2) +
    pow(data->acc[1], 2) + pow(data->acc[2], 2));
    if (cursor >= SAMPLE_SIZE) {
        _step_count_analyze();
        cursor = 0;
    }
    return true;
}
```

5.2.6 Communication

The Android device and the Smartwatch communicate over Bluetooth.

Bluetooth is a short-range wireless technology standard for exchanging data between fixed and mobile devices over short distances and building personal area networks (PANs)[18]. Unlike the client-server model, which is seen in most network-based communication systems, in Bluetooth, both sides can send or receive independent requests.

- **Initial Approach**: The Raspberry Pi Pico W has a built-in Bluetooth module implemented by a third-party library called BtStack[19]. The library and Pico W integration are new since the series was released in 2022. Due to the lack of examples and documentation, it's hard to build Bluetooth applications with the built-in library.

  The existing examples of the BtStack found in Github are mostly blocking examples, which are not ideal for this project since they rely heavily on interrupts and asynchronous programming.

- **Introduction of HC06**: HC06 is a Bluetooth slave module generally used with Arduino and the original Raspberry Pi units. Due to the ease of integration, the

number of examples made it the ideal choice for the project. In addition to that, since the HC06 is an external module, the Bluetooth connection can be done without blocking the core.

HC06 has four pins and uses UART for communication:

HC06	Pico W	Description
VCC	VSYS	5V Power supply
GND	GND	Ground
RX	TX	Receiver pin hooked up to the TX pin of the Pico
TX	RX	Transmission pin hooked up to the RX pin of the Pico

Table 5.3 HC06 Pico W Connections

To set up the HC-06 module, we need a USB serial adaptor. Then, we can send AT commands to configure it. On Linux, the USB devices will be connected to */dev/ttyUSB*. Running the following shell script will assign the name and pin code to the HC-06.

```
#!/bin/bash
SERIAL_PORT="/dev/ttyUSB0"
stty -F $SERIAL_PORT 9600
echo "AT+NAMESW Smartwatch" > "$SERIAL_PORT"
echo "AT+PIN1234" > "$SERIAL_PORT"
```

The following code initialises the *uart0* for the UART protocol.

```
void bt_init(void)
{
    bt = (BtFd) {
        .id = uart0, .baud_rate = 9600,
        .tx_pin = 0, .rx_pin = 1,
        .is_enabled = true,
    };
    uart_init(bt.id, bt.baud_rate);
    gpio_set_function(bt.tx_pin, GPIO_FUNC_UART);
    gpio_set_function(bt.rx_pin, GPIO_FUNC_UART);
}
```

- **Synchronous Version**: Pico's UART mechanism can store up to 32 bytes of data before processing. Any data received from that point is overflowed.

```
size_t bt_read(char* str, size_t str_s)
{
    if (!bt_is_readable()) return 0;
    memset(str, '\0', str_s);
    uint i = 0;
    size_t remaining;
    while ((remaining = uart_is_readable(bt.id)) > 0 && i < str_s)
        str[i++] = uart_getc(bt.id);
    if (i > 0) state.__last_connected = get_absolute_time();
    return i;
}
enum bt_fmt_t bt_receive_req()
{
    if (bt.packet_lock) {
        return ERROR(READ_PACKET_LOCK);
    }
    bt.packet_lock = true;
    size_t bytes = bt_read(bt.packet, 240);
    if (bytes > 0) bt_handle_req(bt.packet, bytes);
    bt.packet_lock = false;
    return BT_FMT_OK;
}
```

- **Interrupt Handler Version**: An interrupt handler version is written to solve this overflow issue.

    1. Packet and cursor fields are added to the struct. The packet stores a single received packet while the cursor stores the current index.

        ```
        typedef struct {
            uart_inst_t* id;
            int baud_rate;
            int tx_pin;
            int rx_pin;
            bool is_enabled;
            char packet[240];
            uint cursor;
        } BtFd;
        ```

    2. *is_str_complete* function checks whether the received data is completed. When completed, every received data from the HC-06 ends with `\r\n\0`.

Our protocol always ends with the | character. So, the following function returns true if the string is completed.

```
static bool is_str_complete()
{
    if (bt.cursor < 4) { return false; }
    return bt.packet[bt.cursor] == '\0'
           && bt.packet[bt.cursor - 1] == '\n'
           && bt.packet[bt.cursor - 2] == '\r'
           && bt.packet[bt.cursor - 3] == '|';
}
```

3. The following code block is the UART receive interrupt handler, which reads one byte at a time and increases the cursor until the string is completed. When the string is completed, it handles the received message and resets the bt's buffer and cursor.

```
void on_uart_rx()
{
    size_t remaining = uart_is_readable(bt.id);
    if (remaining > 0 && bt.cursor < 240) {
        bt.packet[bt.cursor++] = uart_getc(bt.id);
    }
    if (is_str_complete()) {
        bt_handle_req(bt.packet, bt.cursor - 2);
        memset(bt.packet, '\0', 240);
        bt.cursor = 0;
    }
}
void bt_init()
{
    /* ... */
    irq_set_exclusive_handler(UART0_IRQ, on_uart_rx);
    irq_set_enabled(UART0_IRQ, true);
    uart_set_irq_enables(bt.id, true, false);
}
```

5.2.7 The Protocol

This section describes the medium of the embedded system and the Android application. Both systems will communicate through Bluetooth using a serialised

common protocol. The requests and responses are denoted with a number followed by their arguments. Protocol uses | character as the separator.

```
REQUEST_TYPE[int]|ARG_1|ARG_2...|
6|3|2210|1930|0945|0855|
```

1. **Requests**: The Smartwatch can receive the following requests:

   - **0 - BT_REQ_CALL_BEGIN**: Request declares that the phone is being ringed. The request's payload contains the caller's full name.

     ```
     0|Name Surname|
     ```

   - **1 - BT_REQ_CALL_END**: Request declares to stop the phone call popup. This request is sent when the phone call is answered on the phone.

     ```
     1|
     ```

   - **2 - BT_REQ_NOTIFY**: Request declares that a notification has been received.

     ```
     2|Title|Description|
     ```

   - **3 - BT_REQ_REMINDER**: Request declares that a reminder has been received.

     ```
     3|NumberOfReminders|title|yyyyMMddhhmm...|
     ```

   - **4 - BT_REQ_OSC**: Request declares that information about the playing media has changed. The second argument is whether the playing media is paused or not.

     ```
     4|t/f|Song Name|Artist|
     ```

   - **5 - BT_REQ_FETCH_DATE**: Request provides the information about the current date to update.

     ```
     5|yyyymmddhhMMss|
     ```

   - **6 - BT_REQ_FETCH_ALARM**: Request provides the information about alarms. The second argument provides the number of alarms to send up to four. Arguments after that provide the alarm's hour and minute one after the other.

     ```
     6|NumberOfAlarms|hhMM|hhMM|hhMM...|
     ```

   - **7 - BT_REQ_STEP**: The request requests the daily step count. A response is sent after the arrival of this message.

     ```
     7|
     ```

- **8 - BT_REQ_HB**: The request is sent after not sending any requests for 15 seconds to test the connection's status.

```
8|
```

- **9 - BT_REQ_CONFIG**: The request is sent to configure various settings on the watch. Brightness is a 3-digit integer; the rest are flag values defined in the struct.

```
9|Brightness|Alarm|Call|Notify|Reminder|
```

2. **Responses**: The Smartwatch can send responses.

- **0 - BT_RESP_OK**: Request is handled successfully.

```
0|
```

- **1 - BT_RESP_ERROR**: An error is occurred during handling.

```
1|
```

- **2 - BT_RESP_CALL_OK**: Answer the call.

```
2|
```

- **3 - BT_RESP_CALL_CANCEL**: Dismiss the call.

```
3|
```

- **4 - BT_RESP_OSC_PREV**: Switch to previous song.

```
4|
```

- **5 - BT_RESP_OSC_PLAY_PAUSE**: Play/Pause the current song.

```
5|
```

- **6 - BT_RESP_OSC_NEXT**: Skip to the next song.

```
6|
```

- **7 - BT_RESP_STEP**: Send the current step amount.

```
7|123|
```

5.3 Android Application

The Android application is written in Dart programming language using the Flutter framework. For the application, the following packages are used as external dependencies.

Dependency	Version
bluetooth_classic	+0.0.1
cupertino_icons	+1.0.2
flutter_charts	+0.5.2
get	+4.6.6
intl	0.19.0
nowplaying	2.1.0
phone_state	1.0.3
permission_handler	+11.1.0
reflex	+0.0.2
shared_preferences	+2.2.2
system_media_controller	+0.0.2

Table 5.4 Android Dependencies

The application has three menus embedded into the *main_view*. The main view contains a button to connect to or disconnect from the device.

```
enum Page {
  alarm(idx: 0, text: "Alarm", icon: Icon(Icons.alarm)),
  reminder(idx: 1, text: "Reminder", icon: Icon(Icons.task_alt)),
  steps(idx: 2, text: "Steps", icon: Icon(Icons.directions_walk));

  final int idx;
  final String text;
  final Icon icon;

  const Page({required this.idx, required this.text, required this.icon});

  getNavigationItem() => BottomNavigationBarItem(icon: icon, label: text);

  static Page? of(int id) => Page.values
                                    .firstWhere((page) => page.idx == id);
}
```

1. **Alarm Menu**: User can save up to 4 alarms and edit or delete them.
2. **Reminder Menu**: User can save infinite reminders, edit or delete.

3. **Step Menu**: The user can graphically view the pedometer statistics about today and history.

5.3.1 Bluetooth Controller

The Bluetooth controller class abstracts the communication between the device and the phone. It listens and routes incoming requests and provides a helper function to send requests. Bluetooth controller requires a *SharedObject* interface to operate.

5.3.2 Storage Controller

Storage controller saves the current state of the alarms, reminders and steps to *Shared Preferences*. The storage controller also contains a custom generic *CrudOperator* class, which stores and fetches any class that implements *StoredObject* interface.

5.3.3 Notification Controller

The notification controller listens to the incoming notification events and sends them to the device if available.

Figure 5.11 describes the class diagram.

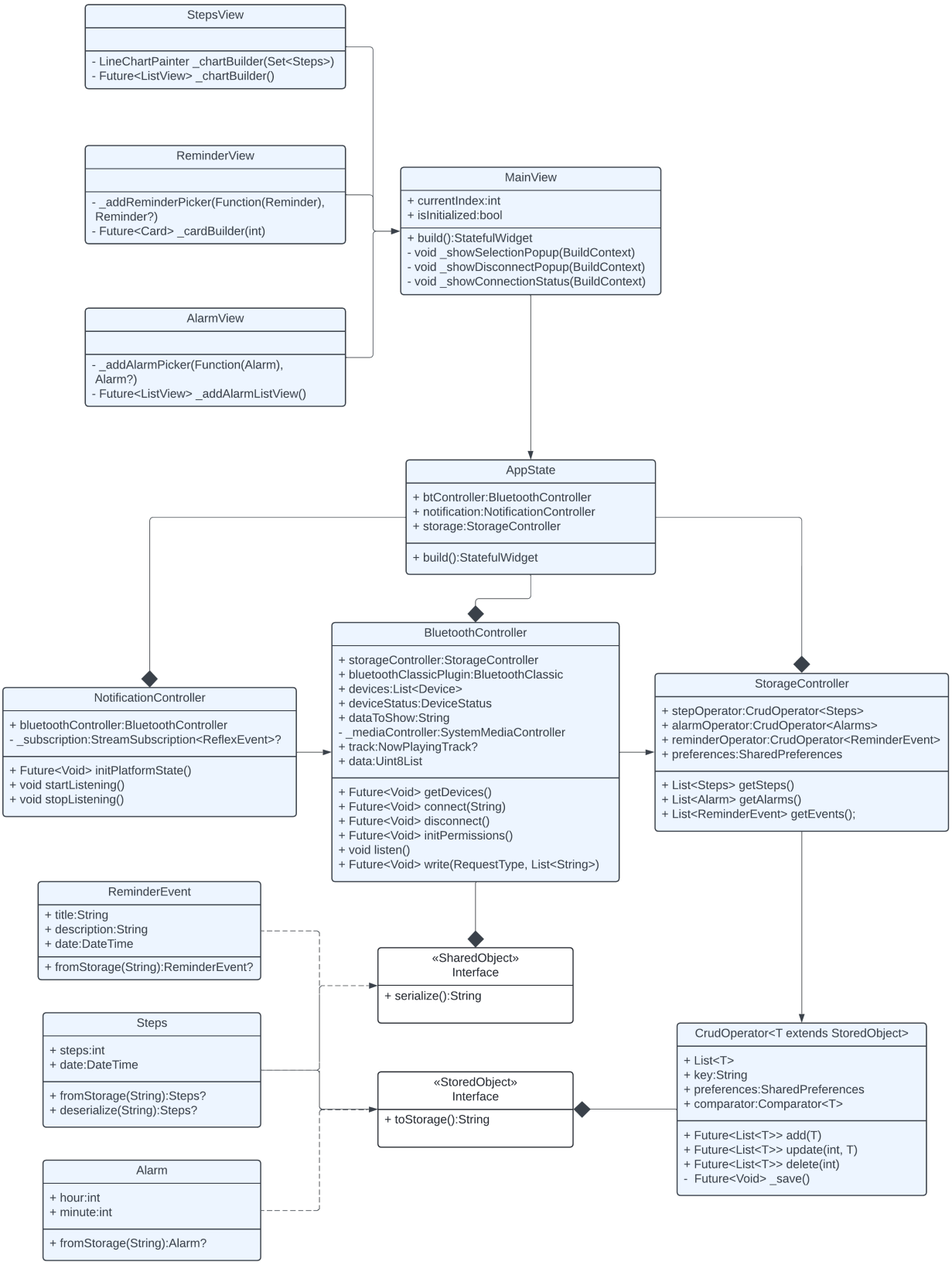

Figure 5.11 Android Application's Class Diagram

6
Application

This section describes how the user interface works from the end user's point of view.

6.1 Embedded System's Interface

6.1.1 Lock Screen

Shuts down the display and the backlight. Calls sleep to the core periodically. The lock screen ends upon receiving a pop-up such as a call, notification or alarm or when the user double-clicks again.

6.1.2 Clock

Displays the current date, day of the week, time in hours and minutes and the temperature. Clicking on the screen toggles between showing a second or not. Swiping up opens the menu. Double-clicking locks the screen.

Figure 6.1 Clock

6.2 Menu

The menu allows users to select one application by clicking on it. Users can swipe left and right to switch to another application. Swiping down returns to the clock.

Double-clicking locks the screen.

Figure 6.2 Example menu

6.3 Applications

Each application contains a window on the top displaying the application name. Users can click the close button to return to the menu.

6.3.1 Alarm

The time values on the alarm screen are obtained via Bluetooth whenever they are updated on the application. If Bluetooth is not connected, the last sent values are used.

Figure 6.3 Alarm Screen

Users can view existing alarms on the screen and toggle them on and off by clicking on them. A background service runs every minute to check active alarms without blocking the screen. When the time comes, a pop-up screen is generated for active alarms.

6.3.2 Stopwatch

The stopwatch app contains two states. Start and stop. The start launches a background task called EVENT_CHRONO, which periodically updates the internal chronometer in the global state struct. The task continues even if the application is closed. The task can be stopped by pressing the Stop button. A tray icon is displayed when the chronometer is active.

Figure 6.4 Stopwatch Button

6.3.3 Mediaplayer

Mediaplayer displays the current song and its artist. Pressing the buttons on the screen sends a request to the connected phone and updates the data. If the song changes, it will be automatically updated.

Figure 6.5 Media Player Screen

6.3.4 Pedometer

The user interface displays the daily steps. A background service counts the value using the accelerometer.

Figure 6.6 Pedometer Screen

6.3.5 Calendar

The calendar screen displays a detailed calendar of the month.

Figure 6.7 Calendar Screen

6.3.6 Notepad

A simple notepad application that stores data throughout the session. Users can take notes on the screen in four colours: white, red, blue, and green. Notepad also provides an eraser and clear screen buttons.

Figure 6.8 Notepad Screen

6.3.7 Debugger

The debugger displays a live feed of system logs to analyze possible bugs.

Figure 6.9 Debugger Screen

6.4 Pop-up Events

This section shows the non-application user interfaces.

Messages from Bluetooth or changes in the system can trigger pop-up events. For this, all screens periodically check for various conditions. If a pop-up is triggered, the current state's data is preserved to recreate on exit(current selected screen, touch screen state, such as Gesture Capture or Pixel Capture). Then, the relevant pop-up screen is loaded. If a pop-up is already active, a comparison is made between pop-ups.

There are four types of pop-ups, namely CALL, ALARM, REMINDER, and NOTIFY, in order of priority. If a more priority pop-up arrives, it replaces the current one.

Pop-ups trigger device GPIOs such as LED, buzzer and motor.

6.4.1 Call Pop-up

Occurs on an event of a call to the connected device. The interface displays the caller's name and creates a menu with accept and dismiss buttons. Pressing either of the buttons will cause the connected device to take action. Pressing the exit button ignores that notification. This notification can be cancelled earlier remotely by answering the call from the connected device.

The buzzer, LED and Motor will be triggered continuously until a button is pressed.

Figure 6.10 Call Screen

6.4.2 Alarm Pop-up

Whenever an alarm is triggered, it creates a pop-up about the time. The alarm pop-up interface contains a single button that cancels the alarm.

The buzzer and LED will be triggered continuously until a button is pressed.

Figure 6.11 Alarm Pop-up

6.4.3 Notification Pop-up

Notifications interface contains a title, text and an exit button. Whenever a push notification is received on the connected device, it is sent to the smartwatch to display.

The buzzer and motor will be triggered once when the notification is received.

Figure 6.12 Notification Screen

6.4.4 Reminder Pop-up

The Reminder interface contains the event date and its title Whenever a saved reminder event occurs, it is sent to the smartwatch to display.

The buzzer and motor will be triggered once when the reminder is received.

Figure 6.13 Reminder Screen

6.5 Android Application

The application stores information about the smartwatch. It captures events such as media, notifications or incoming calls and transfers them to the smartwatch.

6.5.1 Alarm Menu

Contains a list of alarms saved to the Shared Preferences. Users can click the floating action button to add a new alarm up to 4 times. Users may delete or update existing alarms. Any update on alarms will be saved automatically and sent to the smartwatch if available.

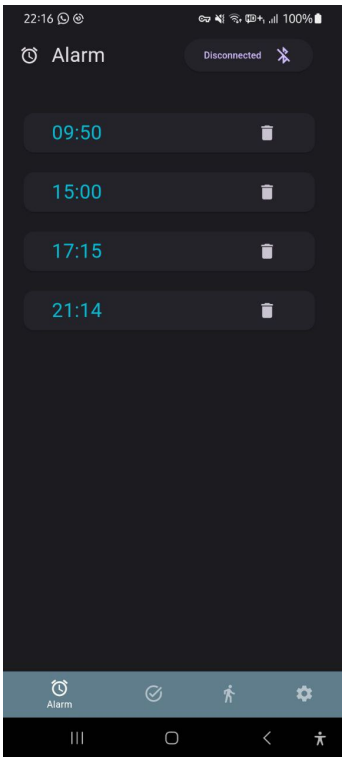

Figure 6.14 Android Alarm Screen

6.5.2 Reminder Menu

Contains a list of reminder events saved to the Shared Preferences. Users can click the floating action button to add a new event. Users may delete or update existing events. If available, the closest three events will be sent to the smartwatch on any update.

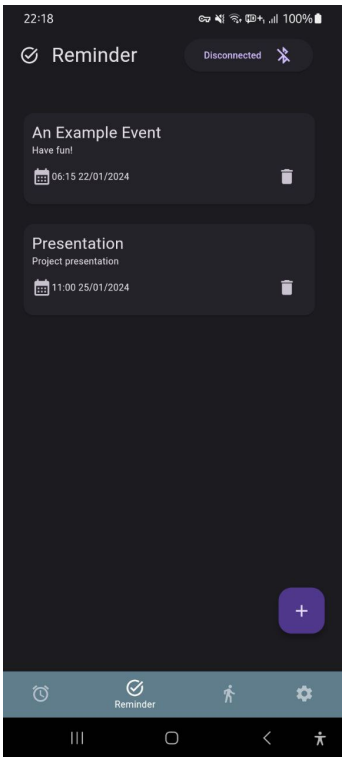

Figure 6.15 Android Reminder Screen

6.5.3 Steps Menu

Displays the daily pedometer statistics taken from the smartwatch. Upon receiving a value, save it to the Shared Preferences to show in the graph and the table.

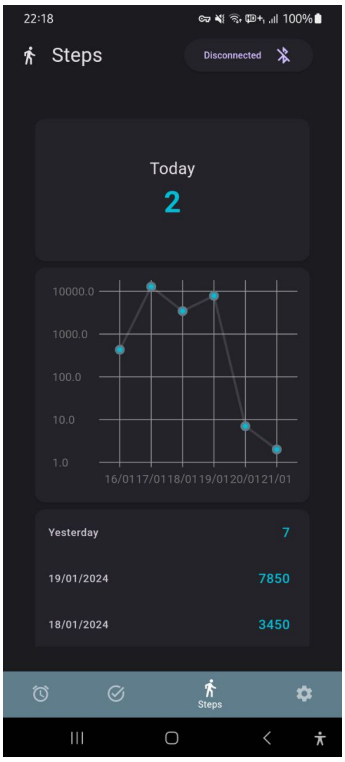

Figure 6.16 Android Steps Screen

6.5.4 Settings Menu

A menu that allows users to configure the notification settings and the brightness.

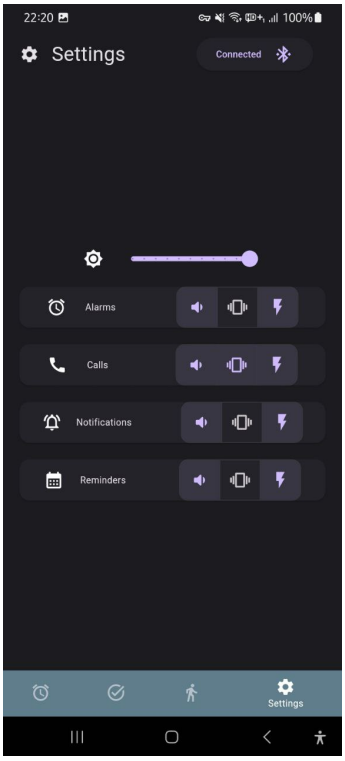

Figure 6.17 Android Settings Screen

7
Performance Analysis

This section examines the performance of the systems outlined and explained in the System Design section. Throughout the programming process of the project, various metrics such as performance, security, and resource utilization have been scrutinized, aiming to achieve the best possible results.

7.1 The Smartwatch

The software is built entirely in the C programming language under the following parameters:

- CMake Version: 3.12+
- C Version: 11
- Pico SDK Version: 1.3.0+
- Flags:
  - -O3
  - -Wextra
  - -Wno-format
  - -Wno-unused-function
  - -Wno-maybe-uninitialized

7.1.1 Resources

The software runs under 264KB RAM, and the binary size is around 2.3MB, which is feasible. However, due to the lack of an operating system, there is no way to trace and monitor the exact memory usage of the application at a given time.

The program utilizes both cores to provide an enhanced user experience without blocking with the help of background tasks. The program heavily relies on the stack memory to improve the performance further.

7.1.2 Power Consumption

The device consumes around 90mAh when active. The battery chosen for the demo product is 120mAh, which means it has around 1.5 hours of battery life. However, the device's case can hold up to 900mAh batteries, which would give up to 10 hours of battery.

7.2 The Android Application

The Android application is written in Dart programming language and Flutter Framework. Flutter is not a native application development framework, and because of that, its performance is worse than the Native alternatives, and its binary size is bigger.

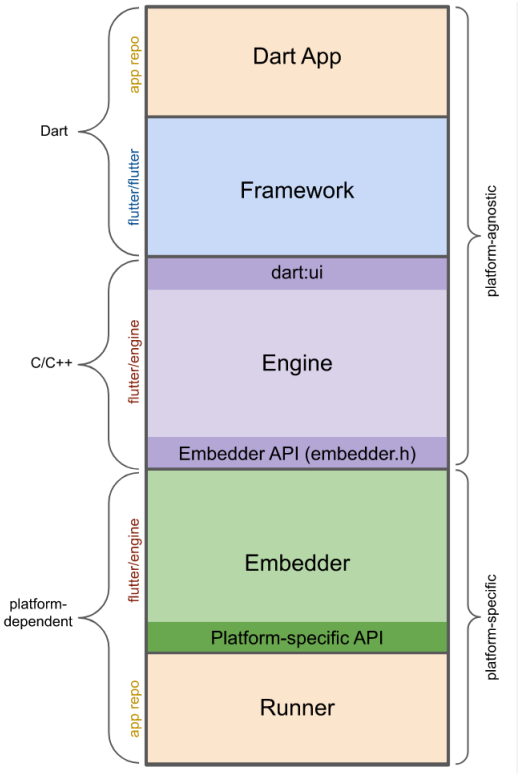

Figure 7.1 Flutter's Architecture

Unlike other cross-platform mobile development frameworks such as React Native, Flutter uses its own rendering engine, which improves its performance since it doesn't

require foreign function calls for each draw call.

But it comes with a drawback. Flutter brings all assets of the application along with it in a binary, whereas a native application would dynamically link the application library at runtime.

The Android application is a relatively minor feature and works as a transition layer between the Android Operation System and the smartwatch[20].

The application takes 450MB of disk space and 200MB memory.

8
Resolution

8.1 The Subject

The subject of this project is the design and development of a smartwatch with integrated functionalities such as touchscreen control, Bluetooth connectivity, multimedia control, notifications, call status monitoring, clock features, stopwatch, alarm, reminder, and step counting. The smartwatch was programmed directly on the microcontroller in the C programming language without using any operating system. A mobile application for Android devices was also created to communicate with the smartwatch and control various features.

8.2 Project Objectives

The primary objectives of this project include:

- **Smartwatch Hardware Design:** Incorporate a touchscreen, buzzer, motor, accelerometer, and microcontroller into the smartwatch design.
- **Smartwatch Software Development:** Develop firmware for the smartwatch using the C programming language without relying on an operating system. Implement a mechanism to concurrently manage touch input, screen interactions, and background processes. Create interfaces for different scenarios, allowing users to control multimedia, receive notifications, and monitor call status.
- **Mobile Application Development:** Design and develop an Android application that communicates with the smartwatch via Bluetooth. Enable transmitting multimedia controls, notifications, and call status information from the mobile app to the smartwatch. Implement a user-friendly interface for seamless interaction with the smartwatch features.

- **Clock Features and Additional Functions:** Implement basic clock functions, including timekeeping, stopwatch, alarm, reminder, and step counting, directly on the smartwatch. Ensure these features continue operating in the background without interfering with other functionalities.
- **Menu System and User Navigation** Develop a menu system on the smartwatch for easy navigation between different interfaces and features.
- **Alert Systems Integration** Integrate alert systems such as vibration and alarm sounds into the smartwatch for notifications and alerts.
- **3D Printed Case Design** Design a compact 3D printable case for the smartwatch to protect the embedded system.

8.3 Future Work

The project was designed to be a prototype designed by a single person within three months. Because of these reasons, there is room for improvement in many areas. I can recommend the following enhancements:

- **Health Monitoring**: Integrate features such as real-time heart rate monitoring and GPS tracking to elevate the health and fitness functionalities of the smartwatch.
- **Interconnectivity:** Explore seamless integration with a spectrum of devices, including smartphones and laptops, to enhance the smartwatch's versatility and user experience.
- **Persistent Data Storage:** Implement a file storage mechanism to retain persistent data. This could include storing timestamps or user notes and ensuring seamless continuity across usage sessions.
- **Improved Communication:** Improve the communication capabilities by adding a video camera, enabling users to answer calls directly from the smartwatch.
- **Security Measures:** Implement security mechanisms to protect the communication between the smartwatch and the mobile application.
- **Modularize the Code Base:** Develop a framework so an external system can be easily embedded into the program as a dependency.
- **Open Source Contribution:** Open-sourcing the firmware and mobile application code to allow community contributions.

- **Add IOS Support:** Find alternatives to the Android-only packages and release the application for IOS.

References

[1] *Smartwatch market analysis*, https://www.fortunebusinessinsights.com/smartwatch-market-106625.

[2] *Smartwatch market size*, https://www.mordorintelligence.com/industry-reports/smartwatch-market.

[3] *Top selling smartwatch brands*, https://www.grandviewresearch.com/industry-analysis/smartwatches-market.

[4] *Smartwatch comparision*, https://www.pcmag.com/picks/the-best-smartwatches.

[5] *Raspberry pi pico w*, https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html.

[6] *1.28inch touch lcd from waveshare*, https://www.waveshare.com/wiki/1.28inch_Touch_LCD.

[7] *Cmake*, https://cmake.org/.

[8] *Figma*, https://www.figma.com/.

[9] *Flutter*, https://flutter.dev/.

[10] *Docker*, https://docs.docker.com.

[11] *Raspberry pi pico c/c++ sdk*, https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf.

[12] *Waveshare lcd driver for rpi pico*, https://files.waveshare.com/upload/3/3e/1.28inch_Touch_LCD_Pico.zip.

[13] *Gnu utilities*, https://www.gnu.org/.

[14] *Android sdk*, https://developer.android.com/studio.

[15] *1.28inch touch lcd from waveshare*, https://www.waveshare.com/wiki/1.28inch_Touch_LCD.

[16] *Image to c header converter*, https://lvgl.io/tools/imageconverter.

[17] *Raspberry pi alarm interrupts on core 0*, https://www.raspberrypi.com/documentation/pico-sdk/high_level.html.

[18] *Bluetooth overview*, https://www.bluetooth.com/learn-about-bluetooth/tech-overview/.

[19] *Btstack bluetooth library*, https://bluekitchen-gmbh.com/btstack/.

[20] *Flutter architecture*, https://docs.flutter.dev/resources/architectural-overview.

Curriculum Vitae

FIRST MEMBER

Name-Surname: Umut Can Sevdi
Birthdate and Place of Birth: 20.12.1999, Adana
E-mail: umutcan.sevdi@std.yildiz.edu.tr
Phone: 0535 867 70 08
Practical Training: Yapi Kredi Teknoloji, egaranti

Project System Informations

System and Software: Pico SDK, Waveshare SDK, Android SDK, Flutter, Dart
Required RAM: 264KB
Required Disk: 2.3MB