Smart Turn - Technical Guide

Students: Josh Casey(21361783) & Jakub Czerniejewski(21466494)
Supervisor: David Sinclair

Abstract. Smart Turn is an IoT device that acts as a safety solution for cyclists and scooter users of the road that utilises an LED turn signal with turn-by-turn navigation. It consists of wearable LED indicators controlled via a touch sensor mounted to the handlebars while also having a display showing navigation. The aim is to reduce accidents and phone thefts while making cyclists more visible and phones less accessible while also decreasing distractions that may be caused by the phone. The implementation combines Arduino hardware with an Android Mobile application in Flutter serving as a bridge via Bluetooth between the hardware and the navigation system.

Date Completed: 01/05/2025

Table of Contents

Introduction

Smart Turn is an IoT product focused on improving the safety of road users, it achieves this by allowing the user to signal their change of directions using LED turn signals. This removes the ambiguity for other road users, especially in a busy city environment. On top of that Smart Turn has a mounted screen which is used for turn-by-turn navigation, this feature reduces distractions as it only displays vital information compared to a phone while lessening the threat of phone theft which is on the rise in major cities.

Motivation

Roads in cities such as Dublin can be quite dangerous, especially for non-car users of the road. Narrow streets, a heavy amount of traffic and a constant rush of city life can lead to people being unsure of what can happen next. Smart Turns hopes to reduce that ambiguity by allowing users to express their next move with turn signals.

Our motivation for this project is quite simple, to make road users such as cyclers and electric scooter riders more visible on the road while lessening the distractions they might encounter by using their phone for navigation.

Phone theft has become commonplace in cities such as London where according to the BBC a phone is stolen every six minutes this includes phones being snatched from cyclists. Smart Turn will reduce the risk of phone theft as the screen gives navigation information to the user while their phone is kept away safely.

Research

We got the idea for a wearable cyclist vest from our supervisor's website, during our meeting with David Sinclair we expanded the idea to incorporate turn signals and a screen which shows navigation directions to the user. Those are fed from a mobile app. All the components would communicate using Bluetooth. Our supervisor also recommended the Adafruit Flora to us as it can be sewn onto clothes.

A lot of effort went into the hardware research of this project as IoT devices need to be compatible with each other.

We split the project hardware into 3 parts:

Front Assembly:

For the main microcontroller we chose the Arduino Uno due to its large amount of pins, its support for 5V and 3.3V logic, small size and a huge community on top of that it's easy to power with a 9V battery.

The touch sensor was a bit more tricky as there is a large selection out there, we ended up choosing the Adafruit version of the MPR 121 due to the fact it has simple logic, a lot of docs and is both highly accurate and sensitive while being able to be wired to the Uno.

For the screen, we were initially looking at a small LCD display but we chose instead the Adafruit 1.9 320x170 Color IPS TFT Display - ST7789, since it is an IPS panel it is bright and has full-colour availability and enough pixels for all the information we will be outputting. On top of that it supports TFT functions which make it easy to consistently output onto the screen as text can be displayed directly and shapes can be drawn and filled with colour too. This display can also be wired up to a Uno.

For the battery we went with rechargeable 9V batteries with 4500mh capacity, during testing we found that one of them can power the assembly for around 5 hours continuously and charges fairly quickly.

For our Bluetooth module, we choose the venerable HC-05 which uses Bluetooth classic to communicate and uses a Software serial which is ideal for the Uno and it uses 5V which is what the Uno outputs with ease.

Rear Assembly:

We have already decided to go with the Flora for our microcontroller as it can be sewn onto clothes with conductive thread while also being easy to power and having a large enough amount of pins.

Since we are using the Flora we chose Adafruit sewable NeoPixels to pair with it, they are very bright, have a lot of documentation on Adafruit's website and were meant to be used together with the Flora as they also use conductive thread.

Finding a suitable battery for the Flora was a bit of a challenge as it requires 3.3V which narrowed out options greatly, in the end, we chose a Li-Po 3.7V rechargeable battery from RS PRO, sadly we did need to buy an adapter for it as the output connector did not match the Flora.

The Bluetooth module for the Flora also had to be very specific as most of them run on 5V logic and power while the Flora only outputs 3.3V that's why we chose Adafruit Bluefruit LE UART Friend which uses BLE and can run on Hardware Serial which is what the Flora requires due to its limited amount of pins.

Application:

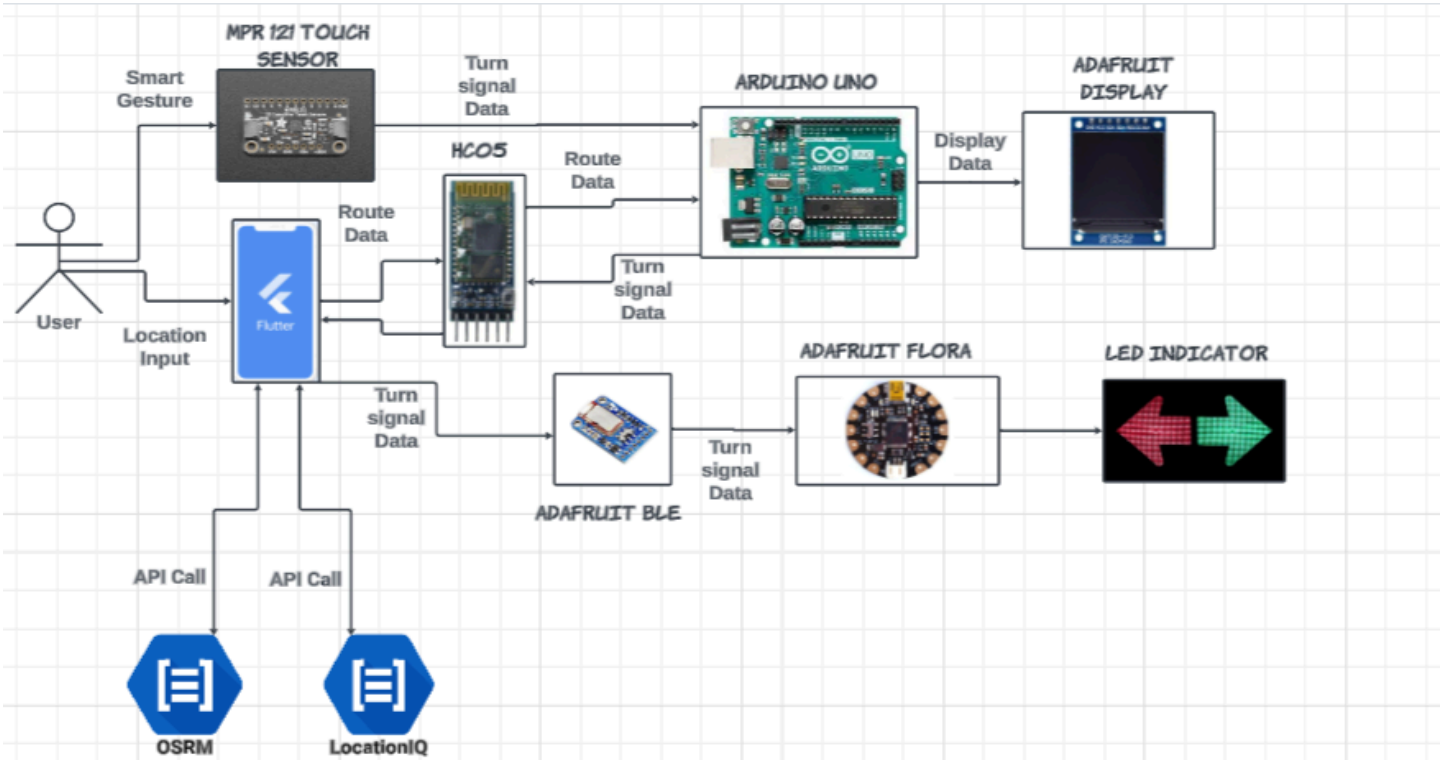
There are a lot of mobile app frameworks out there but we chose Flutter as it has a lot of support, is well-maintained while having one of the biggest package ecosystems on top of that it is written in Dart which is more

performant than Java script and it currently supports Android versions all the way back to Android 8.

We chose OSM for our map as its open source and Google API is out of reach for a college project due to its pricing.

Design

System Architecture Diagram



-

High Level Overview

Flutter Application

Our Flutter application provides a friendly, easy to follow UI to allow users to connect to the Hardware (HC05) and to input their desired location either by selecting it on the map or using textual input. In the background of the application it also acts as a hardware bridge for the two conflicting Bluetooth types of classic and BLE from the HC05 and Flora allowing them to communicate and pass data.

HC05

Our HC05 Bluetooth component is a vital part of our system controlling the data sent from our flutter app to the Arduino to put data onto our display. It also sends data from the Arduino

back to the flutter app consisting of the data received from the MPR121 touch sensor to be sent to the Flora Ble.

MPR121

The MPR121 is a capacitive touch sensor consisting of 12 pins that we utilised for smart gesture turn signalling. If sensor 11 is touched the left indicator is initiated, 0 initiates the right turn signal and 5 turns off the LED,s. The data from this is passed directly to the Arduino to where it then makes its way to the LEDs via the HC05 -> Flutter App -> Adafruit BLE -> Flora to be displayed.

Arduino UNO & Adafruit display

Our Arduino UNO handles all logic for our hardware components where we declare pins and functions for the HC05 and the MPR121 controlling its power passed, data passed and their functionality. The UNO controls the data on the display, clearing it and formatting it to its correct position.

Adafruit BLE & Flora & LED Indicators

The adafruit BLE is set to receive data incoming from the flutter app to pass the indication turn signals to the Flora which handles the LED arrows controlling where to send power for each arrow signal

OSRM & LocationIQ

The main API's used for the Flutter navigation logic is OSRM[21] and LocationIQ[20]. OSRM handles the turn-by-turn navigation using the selected location and the current user's location to get the turn instructions and the distance to the destination and to the next turn. LocationIQ is used for the geocoding of textual input to latitude and longitude coordinates and is also used for auto-completing the textual input to display the top 5 possible desired locations every 600 milliseconds as text is inputted.

Architectural Components

Flutter Application

3rd Party Libraries & Packages

`Async`: Used for Future class and asynchronous programming
`Convert`: Used for conversion of data for JSON
`Material`: Flutters design for widgets and UI sections
`Geolocator`: Used to obtain users' current location [1]
`LatLong`: Geographic and point calculations of coordinates [2]
`Http`: Used for our API requests [3]
`Geocoding`: Bidirectional geocoding [4]
`osm_ominatim`: Another Geocoding service [5]
`flutter_ble_plus`: Used to handle the BLE connection for the Flora [6]
`flutter_map`: Render map tiles and place points for polylines [7]

`flutter_polyline_points`: Decode polyline points for route [8]
`flutter_background_geolocation`: Update coordinates in the background [9]

Initialization

Our Flutter application for an Android mobile is a stateful system consisting of navigation, Bluetooth, geocoding and interactive routing. Upon the main entry point of the application, there are 2 Bluetooth manager instances to handle the connection of the hardware devices the HC05 and Flora BLE. Before this, however, upon starting the application permissions are requested for the mobile location and Bluetooth services.

```dart
@override
void initState() {
  super.initState();
  _checkAndRequestPermissions();
}
```

This is done upon initialization calling `_checkAndRequestPermissions()` before allowing more access to the application.

```dart
Future<void> _checkAndRequestPermissions() async {
  bool locationGranted = false;
  try {
    LocationPermission permission = await Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
      permission = await Geolocator.requestPermission();
    }
    locationGranted = permission == LocationPermission.always ||
        permission == LocationPermission.whileInUse;
  } catch (e) {
    debugPrint('Error checking location permissions: $e');
  }

  // Check Bluetooth permissions
  bool bluetoothGranted = false;
  try {
    // Maybe change to is supported? -> Already Works tho
    bluetoothGranted = await bl2.FlutterBluePlus.isAvailable;
  } catch (e) {
    debugPrint('Bluetooth permissions: $e');
  }
```

You can see we request permissions using already-in-use packages such as `Geolocator` and `FlutterBluePlus` setting boolean flags to keep track of the active components and then using a private `_permissionsGranted` variable to allow initialization of the app via `_initializeApp()`.

As the permissions are granted the app begins to initialise the application tries to connect the 2 Bluetooth devices.

```dart
void _initializeApp() async {
  bool connected = await _bluetoothManager.connectToHC05();
  bool bleFound = await _bleManager.connectBLE();

  if (connected & bleFound) {
    _bluetoothManager.receiveData(_bleManager);
  }

  setState(() {
    _isConnected = connected;
    _connectionFailed = !connected;
  });
```

This is done by utilising the 2 Bluetooth managers using boolean flags `connected` & `bleFound` to indicate a successful or unsuccessful connection

While using the app the user can switch between 2 UI screens, one for the textual input and one for the on-map tap input.

This is controlled using a private variable `_currentIndex` that gets set to the desired page depending on the action or function. As seen the index of `_pages[0]` is the textual initial input screen and `_pages[1]` is for the map tile screen.

Geocoding

For getting coordinates for our desired location from textual we created a `_geoCode()` function that utilises the LocationIQ API for geocoding to extract the coordinates from the text inputted address with the additional Flutter Nominatim and Location packages as a backup for redundancy and diversity in case of failure.

```dart
void _geoCode(String data) async {
  try {

    final locationIQResult = await _fetchLocationIQCoords(data);
    if (locationIQResult != null) {
      _mapScreenKey.currentState?.setDestination(locationIQResult);
      setState(() {
        _currentIndex = 1;
      });
      return;
    }
```

The design idea was implemented due to the issues of Nominatim sometimes returning HTML and the Location package being really unreliable.

Bluetooth

For our connection to the HC05, we utilise the classic Bluetooth manager for a connection connecting it directly to the address of the hardware once again utilising the asynchronous abilities provided using the async package and Future class.

```
// Address to HC05 Bluetooth Module
static const String hc05Address = "00:22:12:01:B3:B1";

// Connect to the HC05 resolving to true if successfull
Future<bool> connectToHC05() async {
  try {
    _connection = await bl1.BluetoothConnection.toAddress(hc05Address);
    return true;
  } catch (e) {
    debugPrint('Error connecting to HC-05: $e');
    return false;
  }
}
```

```
Future<bool> sendData(String data) async {
  try {
    if (!isConnected) {
      throw Exception('Not connected to HC-05');
    }
    Uint8List bytes = Uint8List.fromList(utf8.encode(data));
    _connection?.output.add(bytes);
    await _connection?.output.allSent;
    return true;
  } catch (e) {
    return false;
  }
}
```

For sending data to the HC05 module we first ensure a connection then the String data is converted into bytes to allow the module to understand the messages passed and display them.

Location

For retrieving a location via textual input the user is presented with the top 5 choices that might match using the LocationIQ API. Originally the API was called upon every character change however this led to an abundance of API calls resolving this problem by introducing a timer doubounce to call the API every 600 milliseconds, a similar implementation idea as the Arduino bluetooth message passing.

```dart
void _getPlaces(String data) async {
  if (_debounce?.isActive ?? false) _debounce!.cancel();

  _debounce = Timer(const Duration(milliseconds: 600), () async {
    if (data.trim().isEmpty) {
      setState(() => _places = []);
      return;
    }

    final results = await _fetchLocationIQAutocomplete(data);
    setState(() => _places = results);
  }); // Timer
}
```

For selecting an area on the map I encountered the design issue of accidentally touching the screen and selecting an unwanted area as the selected location. I overcame this by implementing a confirmation pop-up bubble to confirm that the selected location was intended by clicking “yes” if that was the case or “no” if it was an accidental input then leaving the selected location as the same as before.

```dart
void _onMapTap(TapPosition tapPosition, LatLng latlng) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('Confirm Location'),
        content:
            Text('Are you sure you want to set this as your destination?'),
        actions: [
          TextButton(
            child: Text('No'),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ), // TextButton

          // Set location and pop context bubble if "Yes" is selected
          TextButton(
            child: Text('Yes'),
            onPressed: () {
              Navigator.of(context).pop();
              _confirmLocationSelection(latlng);
            },
          ), // TextButton
        ],
      ); // AlertDialog
    },
  );
}
```

Navigation

For our turn-by-turn navigation system, we utilise the osrm API. This can be called upon by a map tap selecting a desired location on the map tiles.

```
void _confirmLocationSelection(LatLng latlng) {
  setState(() {
    _destinationReached = false;
    _selectedLocation = latlng;
    _currentInstruction = "Calculating route";
  });

  // Check if the selected location is too close
  if (_userLocation != null) {
    double distance = _calculateDistance(_userLocation!, latlng);
    // 100 meters??
    if (distance < minimumNavigationDistance) {
      setState(() {
        _currentInstruction =
            "Destination too close (${(distance * 1000).toStringAsFixed(0)}m)";
        _selectedLocation = null;
        _routePolyline = [];
        _navigationSteps = [];
      });
      BluetoothManager.instance.sendData("Destination too close");
      return;
    }
  }

  widget.onLocationSelected(latlng);
  _fetchRoute();
}
```

When selecting a location the function will ensure you are not selecting a location too close to your current location and will also check that there is an existence of your current location to prevent any unnecessary API calls. If none of these conditions are met the `_fetchRoute()` function is called containing the API call to osrm

.

```dart
void _startLocationUpdates() {
  _positionStreamSubscription = Geolocator.getPositionStream(
    locationSettings: const LocationSettings(
      accuracy: LocationAccuracy.bestForNavigation,
      distanceFilter: 2,
    ),
  ).listen((Position position) {
    if (!mounted) return;

    setState(() {
      _userLocation = LatLng(position.latitude, position.longitude);
      _isLoading = false;
    });

    _updateNavigation();
```

For getting constant updates and route calculation I used the Geolocator position stream to update every 2 meters moved for the user's current location, changing the original accuracy from `best` to `bestForNavigation` for better performance over precision.

For updating our navigation instructions we ensure that we are still in the bounds of the route and if we are, we format the data using Flutter's `toStringAsFixed()` method for the total distance and next turn distance to see if it is in kilometres or meters as the API returns everything in meters so this is handled accordingly then sent to the Bluetooth module for display.

Our main API call from the _fetchRoute() function ensures that there exists a user location and a selected location before continuing on for an API call to OSRM. This is done using the `http` package for the web call and response and `converted` for the JSON decoding to extract the needed data from the call

```dart
Future<void> _fetchRoute() async {
  if (_selectedLocation == null || _userLocation == null) return;

  final url = "http://router.project-osrm.org/route/v1/cycling/"
      "${_userLocation!.longitude},${_userLocation!.latitude};"
      "${_selectedLocation!.longitude},${_selectedLocation!.latitude}"
      "?steps=true&overview=full&geometries=polyline";

  try {
    final response = await http.get(Uri.parse(url));

    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      final totalDistance = data["routes"][0]["distance"];
      final distanceKilometer = totalDistance / 1000;

      setState(() {
        _distanceToDestination = distanceKilometer;
      });
```

Background Location Updates

In navigation.dart we initialise an observer to monitor the app's state whether it is "in focus" or "out of focus" We also create an instance of the BackgroundGeolocation class with the following parameters, the main focus is on the desired Accuracy which we set to high and that it starts up on boot, we also have a fail-safe that will start the instance if it's not started automatically.

Firstly we initialize a timer then we Override a function that monitors whether the state of the app has changed. If the app goes back into focus (the user has it in the foreground) the timer gets cancelled to stop the execution of the code inside it.

```
if (state == AppLifecycleState.resumed) {
  print("App has focus");
  _heartbeatTimer?.cancel();
} else if (state == AppLifecycleState.paused) {
  _heartbeatTimer = Timer.periodic(Duration(seconds: 1), (timer) async {
    bg.BackgroundGeolocation.getCurrentPosition(
            persist: false, // <-- do not persist this location
            desiredAccuracy: 0, // <-- desire best possible accuracy
            timeout: 30000, // <-- wait 30s before giving up.
            samples: 1 // <-- sample 1 location before selecting best.
            )
        .then((bg.Location location) async {
      _userLocation =
          LatLng(location.coords.latitude, location.coords.longitude);
      Future.delayed(Duration(milliseconds: 100), () {
        _fetchRoute();
      }); // Future.delayed
```

If the app goes out of focus (into the background for example the user switches to a different app or locks their phone) the timer will execute the code every second. Since the BackgroundGeolocation package runs all the time regardless of the state we can get the user's current location, then the state variable _userLocation gets updated and we call fetch route which updates the next step which is sent to the Uno and displayed on the screen.

Arduino UNO

3rd Party Libraries & Packages

`SoftwareSerial` was used for communication from the pins used for debugging and data analysis. [10]
`Adafruit_GFX` is used for the Adafruit display [11]
`Adafruit_ST7789` is for the specific screen display [12]
`Wire` more communication handling [13]
`Adafruit_MPR121` is used for the capacitive touch sensor [14]

Pins

For our hardware and our Arduino uno, we declared our pins for each component.
Bluetooth HC05 takes pin 9 for receiving, 8 for transmitting and 4 for its state pin for connection status.
The Adafruit Display takes pins 10 for chip selection, 7 for resetting the display and 6 for data commands.

Setup

```
void setup() {
  Serial.begin(9600);
  bluetoothSerial.begin(9600);

  pinMode(STATE_PIN, INPUT);

  delay(1000);
  Serial.println("Bluetooth device ready");

  if (!cap.begin(0x5A)) {
    Serial.println("MPR121 not found, check wiring?");
    while (1);
  }
  Serial.println("MPR121 found!");

  // Initialize display
  tft.init(170, 320);
  tft.setRotation(3);
  tft.fillScreen(ST77XX_BLACK);
  tft.setTextColor(ST77XX_WHITE);
  tft.setTextSize(2);
  tft.setCursor(10, 10);
  tft.println("Bluetooth Ready");

}
```

On starting up the Arduino we listen for data incoming on serial port 9600 for analysis and debugging. We also declare initially our screen messages and check for the existence of the capacitive touch sensor. The rotation of the display is set to 3 as the initial display is set vertically however for our purpose it benefited more to display the data horizontally.

HC05

Set up

For the HC05 module, we declare the pins on which the data is received RX (pin 9), the data is sent TX (pin 9) and the state/connection status STATE (pin 4).

Connection

To track the connection state of the module we keep track of a variable assigned to the state `bool currentState = digitalRead(STATE_PIN);`

```
if (currentState != lastState) {
  tft.fillRect(0, 30, 190, 40, ST77XX_BLACK);
  tft.fillRect(0, 100, 150, 70, ST77XX_BLACK);
  tft.fillRect(210, 35, 100, 100, ST77XX_BLACK);
  tft.setCursor(10, 40);

  // If a change in state occurs update the display
  if (currentState == HIGH) {
    Serial.println("Device Connected");
    tft.println("Connected");
    tft.fillCircle(150, 45, 10, ST77XX_GREEN);
  } else {
    Serial.println("Device Disconnected");
    tft.println("Disconnected");
    tft.fillCircle(170, 45, 10, ST77XX_RED);
  }
  lastState = currentState;
}
```

If the state changes at all the display will be reset to display a new message and also a state of HIGH indicates a successful connection to the module else showing it having no connection. HIGH dictates if a voltage is received or not kind of like a 1 or 0 status for transistors.

Receiving Messages

For reading an incoming stream of messages from the Flutter application via Bluetooth we check if a stream of bytes is coming in and while there is we append each byte to the `recievedMessage` variable to construct and display our message. We utilise the inbuilt `millis()` Arduino function to keep track of the time passed as the system initialised the track of the time between characters sent and messages.

```
while (bluetoothSerial.available()) {
  char c = bluetoothSerial.read();
  receivedMessage += c;
  lastReceivedTime = millis();

  Serial.write(c);
}
```

For the logic for identifying a new message, we check to see if there is a message `recievedMessage.length() > 0` and also check that the time passed between the sequence of characters is greater than 125 milliseconds, `millis() - lastRecievedTime > messageTimeout.`

```
if (receivedMessage.length() > 0 && millis() - lastReceivedTime > messageTimeout) {
  // Clear previous text area
  tft.fillRect(0, 100, 150, 70, ST77XX_BLACK);
  tft.setCursor(0, 100);
  tft.print(receivedMessage);

  receivedMessage = "";
}
```

MPR121

Initialise

Assigned our variable to be used from the provided hardware library

`Adafruit_MPR121 cap = Adafruit_MPR121();`

Logic

```
if (cap.touched() & (1 << 11)) {
  Serial.println("left");
  bluetoothSerial.write("left");
  //tft.drawRect(210, 40, 50, 30, ST77XX_GREEN);

  tft.fillRect(210, 35, 100, 100, ST77XX_BLACK);
  tft.fillRect(260, 60, 40, 30, ST77XX_GREEN);
  tft.fillTriangle(270, 115, 270, 35, 210, 75, ST77XX_GREEN);
  delay(350);
}
if (cap.touched() & (1 << 5)) {
  Serial.println("off");
  bluetoothSerial.write("off");
  tft.fillRect(210, 35, 100, 100, ST77XX_BLACK);
  delay(350);
}
if (cap.touched() & (1 << 0)) {
  Serial.println("right");
  bluetoothSerial.write("right");
  tft.fillRect(210, 35, 100, 100, ST77XX_BLACK);
  tft.fillRect(210, 60, 50, 30, ST77XX_GREEN);
  tft.fillTriangle(240, 115, 240, 35, 295, 75, ST77XX_GREEN);
  delay(350);
}
```

Check to see if pin 11 is pressed to write a left instruction to the flora.

If capacitive pin 5 is touched it turns off the indication and resets the display.

Upon touching pin 0 a right message is sent, a right message is sent and the right arrow is displayed.

Flora

3rd Party Libraries & Packages

Here are the third-party packages used in the Flora code

`#include <Adafruit_NeoPixel.h>` [15]
`#include <Arduino.h>` [16]
`#include <SPI.h>` [17]
`#include "Adafruit_BLE.h"` [18]
`#include "Adafruit_BluefruitLE_UART.h"` [19]

Turn Signals and BLE

The BLE connection LED on the Flora lights up Red when there isn't a BLE connection and turns Blue once a message has been received.

We check if there is an available BLE connection, read the data and use a case statement to either turn on a turn signal or to turn any signal off. The turnSignalId flag is being set, this is then used to turn on the corresponding turn signal.

```
if (turnSignalId == 1)
{
  showColours(LeftSignal.Color(255, 85, 0), 75, 1); //A
  showColours(LeftSignal.Color(0, 0, 0), 10, 1); //Add

}else if (turnSignalId == 2)
{
  showColours(RightSignal.Color(255, 85, 0), 75, 2);//A
  showColours(RightSignal.Color(0, 0, 0), 10, 2);//Add
}
```

The RGB values are being passed to the show colours function, alongside the delay and turnSingnalId. The show colours function loads the LED Strip colour into RAM and displays it.

Adafruit ST7789 Display

Initialise

Assignment of Adafruit package type to TFT for ease of use,
`Adafruit_ST7789 tft = Adafruit_ST7789(TFT_CS, TFT_DC, TFT_RST);`
Upon powering the system the Adafruit Display is initialised with a set rotation, screen and text colour via the `Adafruit_ST7789` package as variable tft. To display our information in the correct areas we utilised `tft.setCursor(x, y)` and initially displayed a “Bluetooth Ready” status.

Connection Status

For displaying a connection status to the display, the cursor is set below the initial message and the screen is cleared to erase any old messages via a black rectangle using `tft.fillRect(x, y, x_size, y_size, colour)` to set a clear canvas for a new one. For ease of use a green and red indicator is also used for connection status using `tft.fillCircle(x, y, r)` to display a green circle for a successful connection and red if not.

```
if (currentState != lastState) {
  tft.fillRect(0, 30, 190, 40, ST77XX_BLACK);
  tft.fillRect(0, 100, 150, 70, ST77XX_BLACK);
  tft.fillRect(210, 35, 100, 100, ST77XX_BLACK);
  tft.setCursor(10, 40);

  // If a change in state occurs update the display
  if (currentState == HIGH) {
    Serial.println("Device Connected");
    tft.println("Connected");
    tft.fillCircle(150, 45, 10, ST77XX_GREEN);
  } else {
    Serial.println("Device Disconnected");
    tft.println("Disconnected");
    tft.fillCircle(170, 45, 10, ST77XX_RED);
  }
  lastState = currentState;
}
```

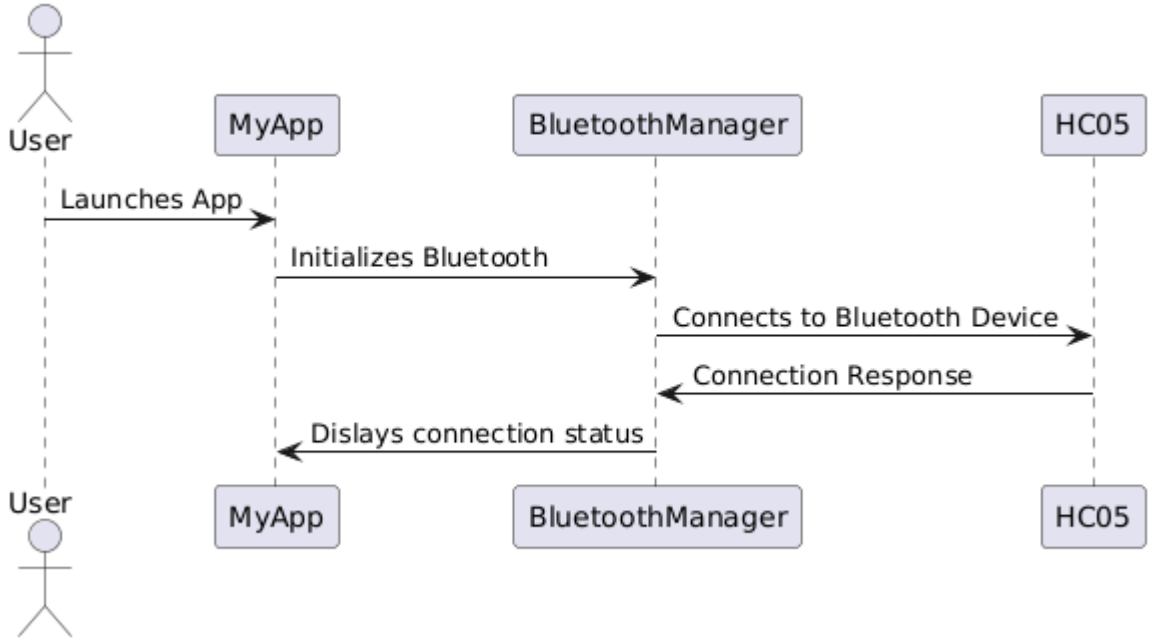
Turn by Turn

Turn-by-turn instructions are displayed in their own unique rectangle area and also cleared and set via changing the pixels to black and redisplaying them every time the turn instruction is updated. Due to the imported `Adafruit_ST7789` package displaying this was as easy as declaring where to put it and printing it using `tft.print(m)`.

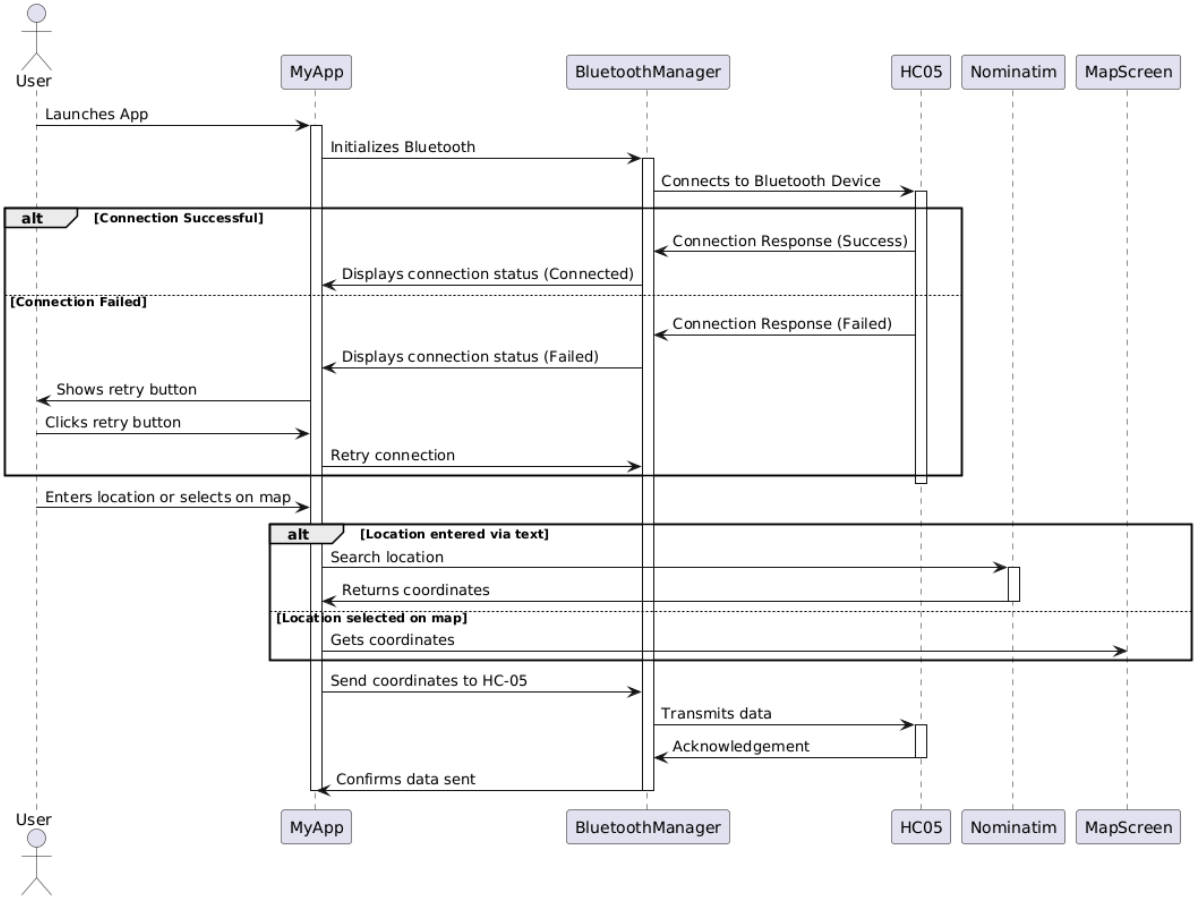
Implementation

Sequence Diagrams

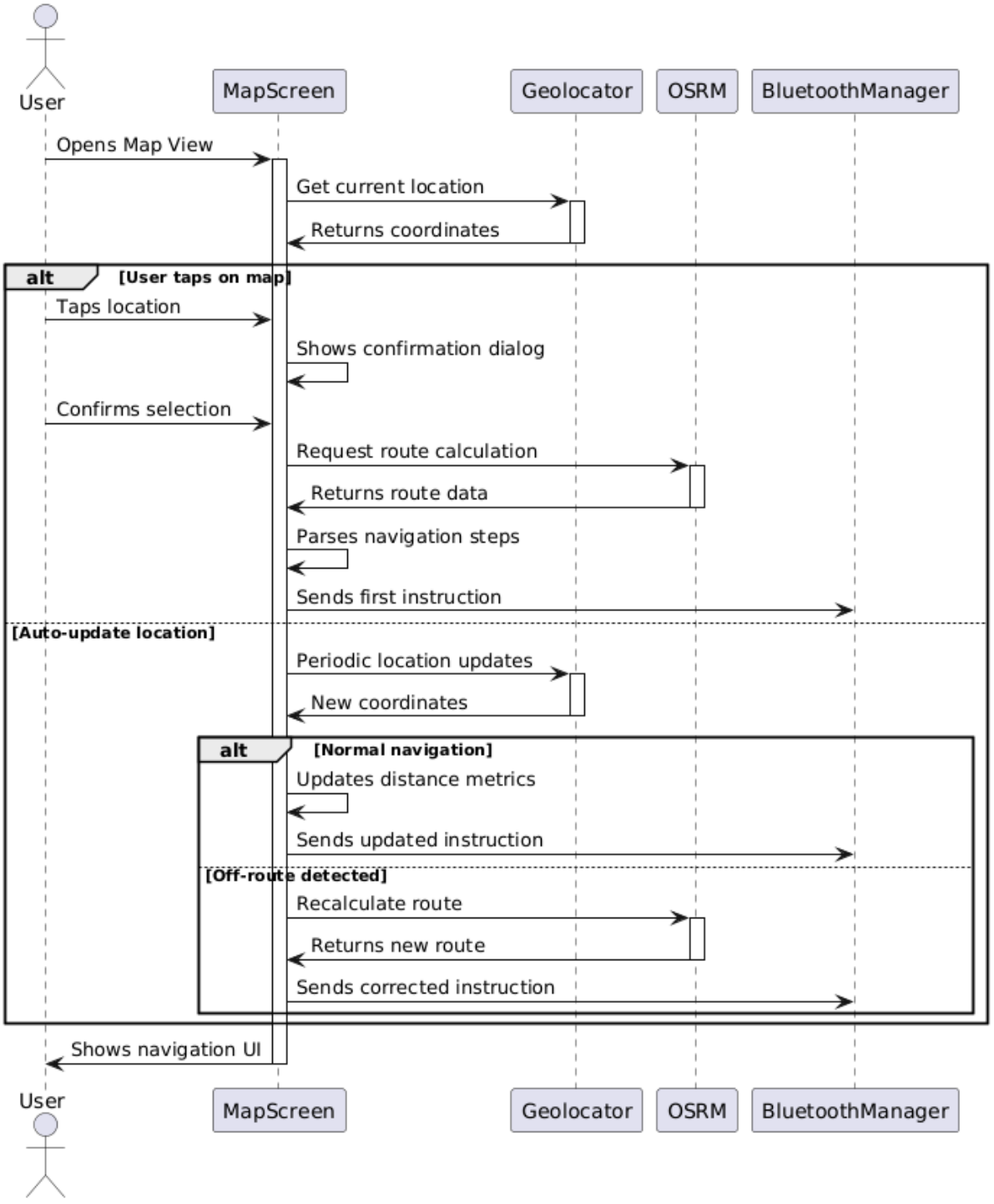
Connecting to HC05 High level



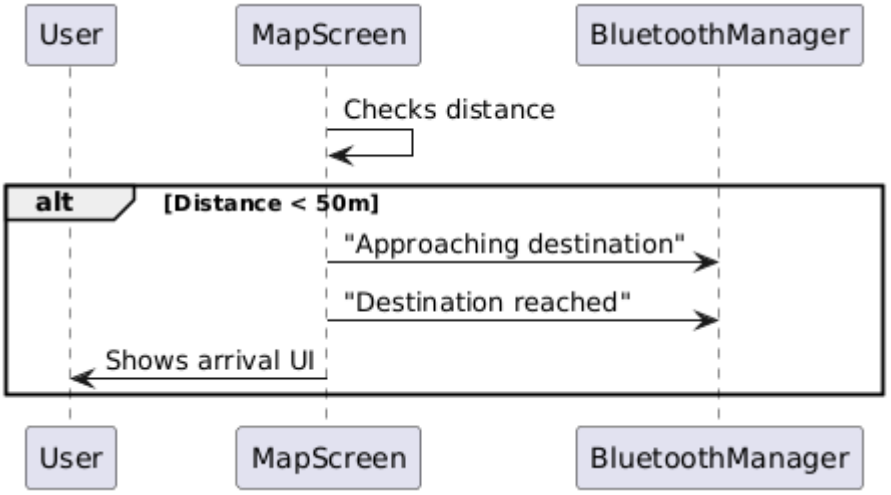
Connecting to HC05 (Depth)



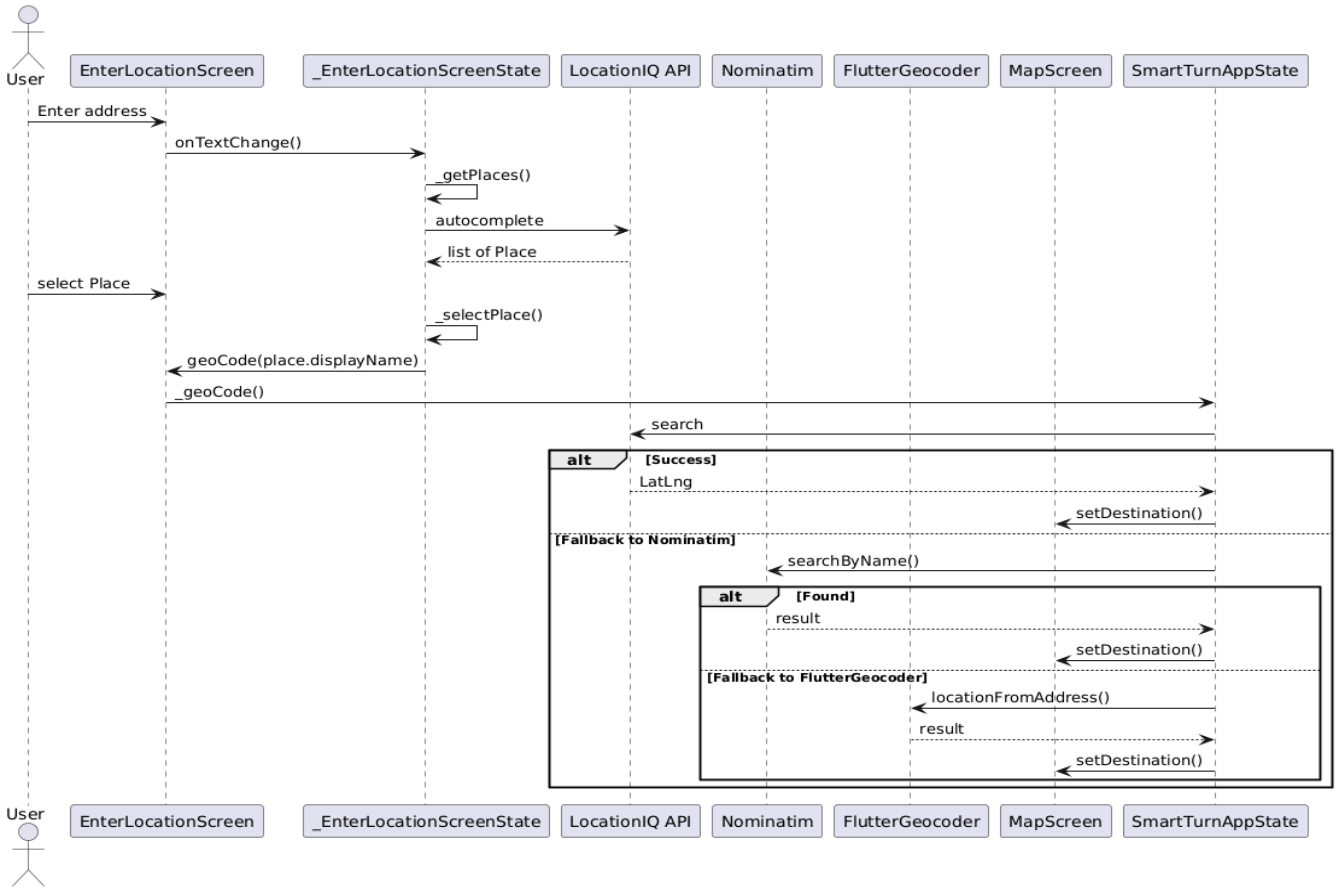
Selecting a location on Map



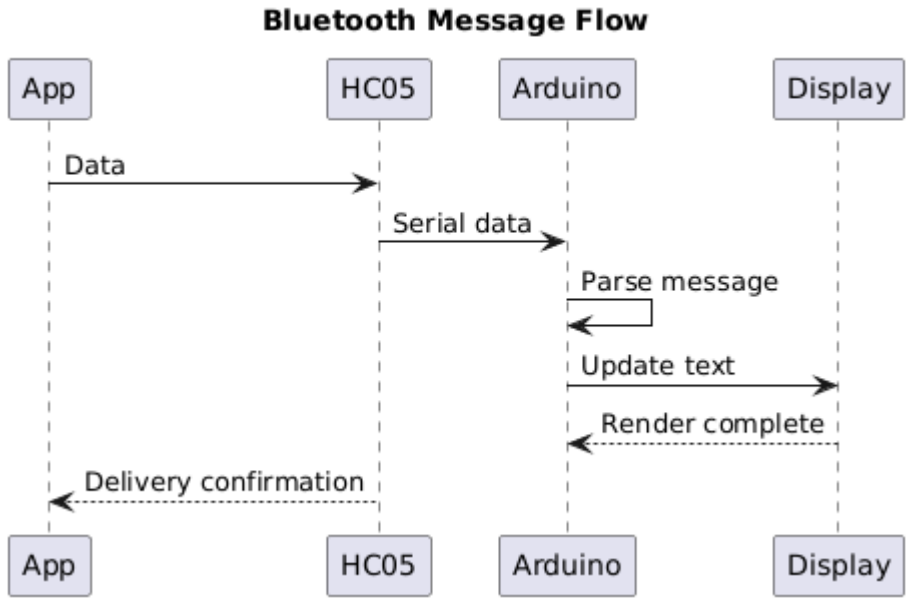
Reaching Destination



Geocoding R&D

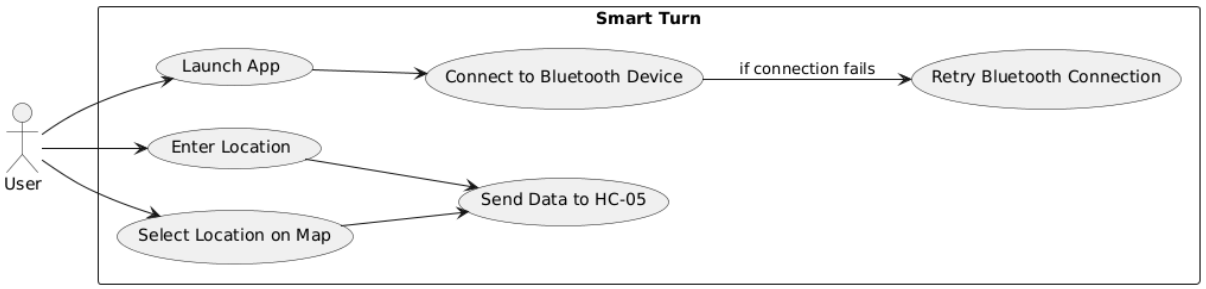


Flutter & Arduino BT Flow

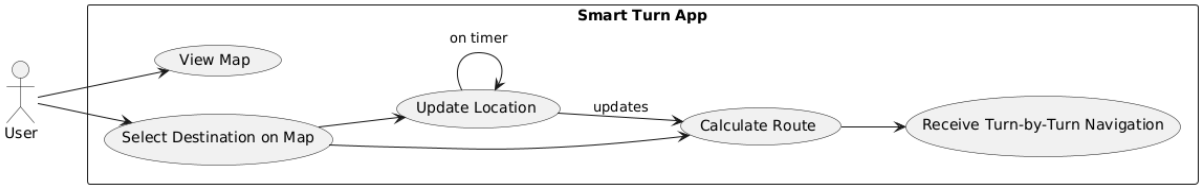


Use Cases

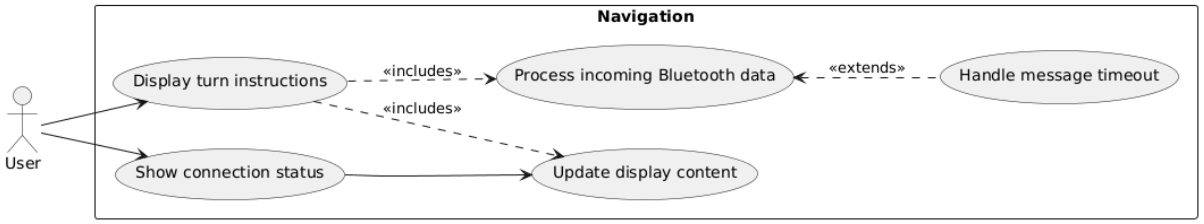
Overview



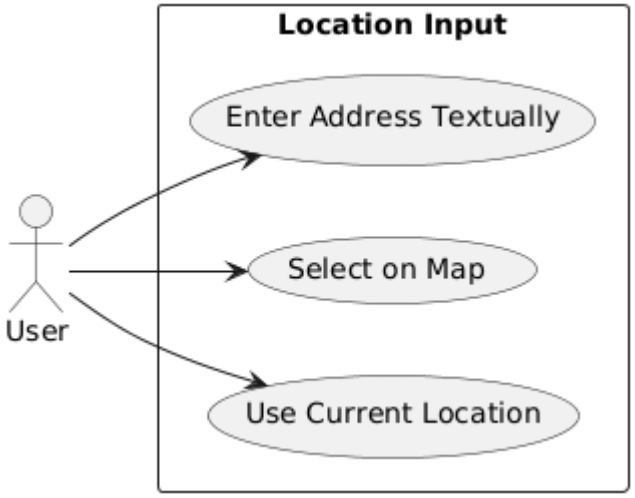
For navigation



Navigation with Display

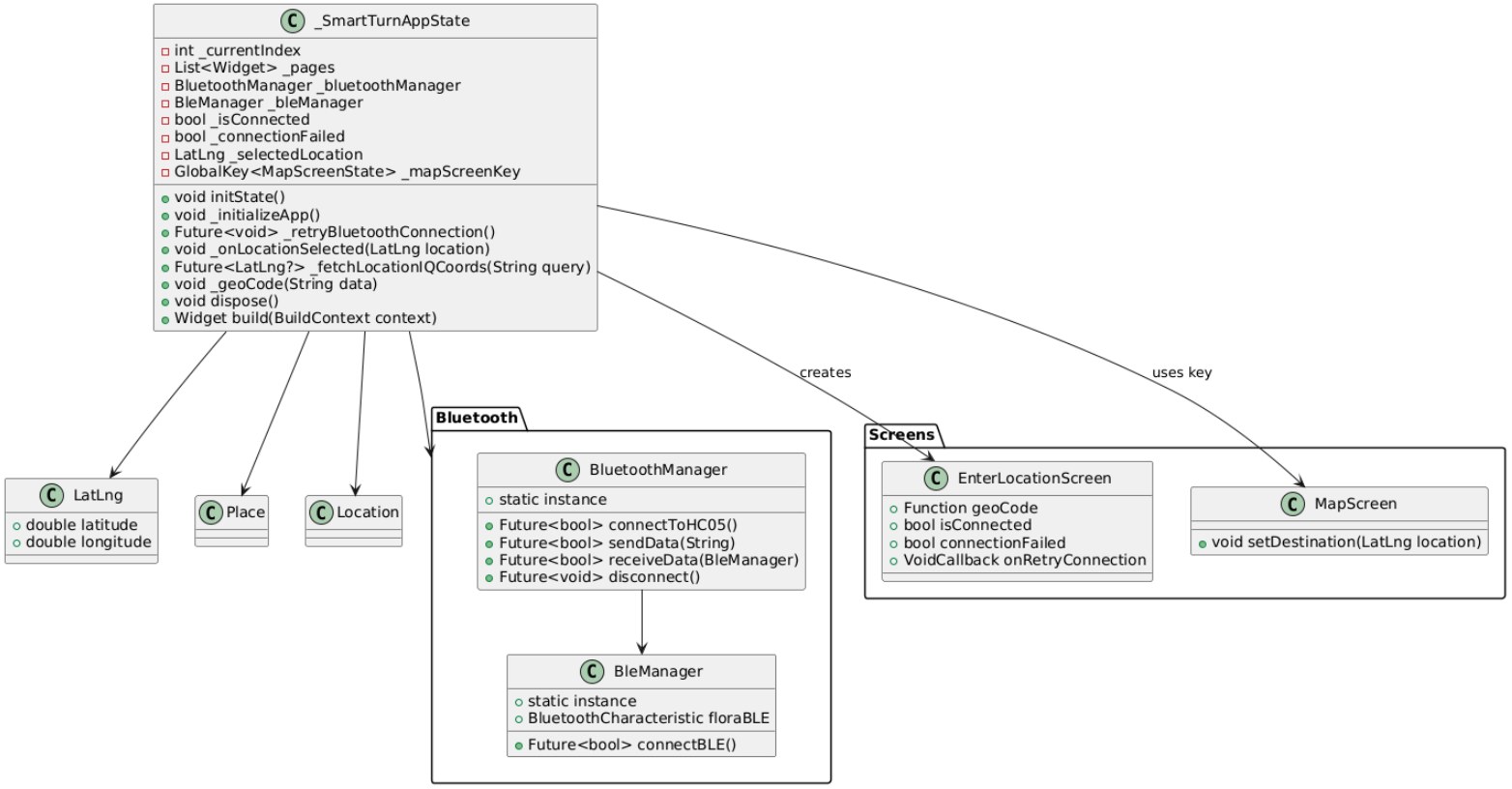


Selecting a Location

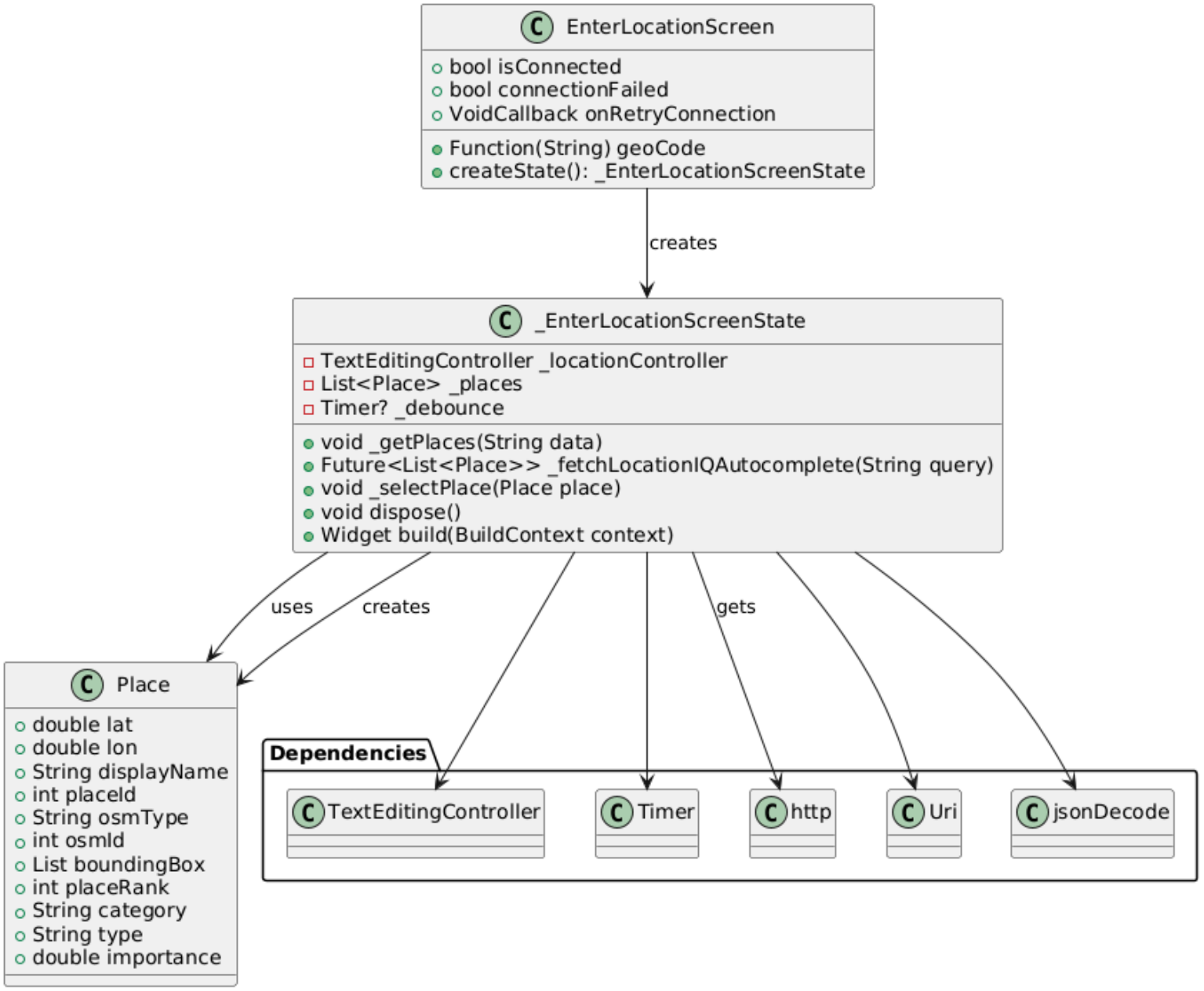


Class Diagrams

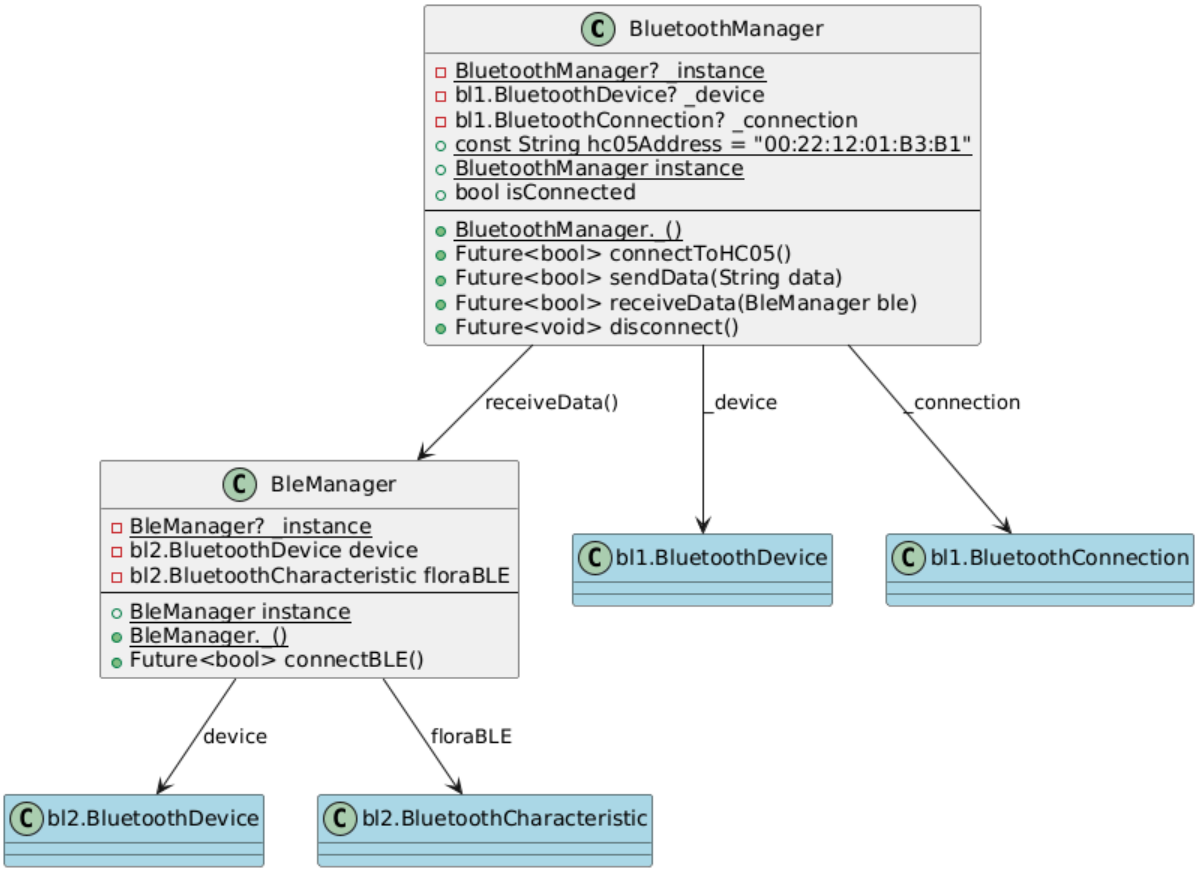
Main.dart



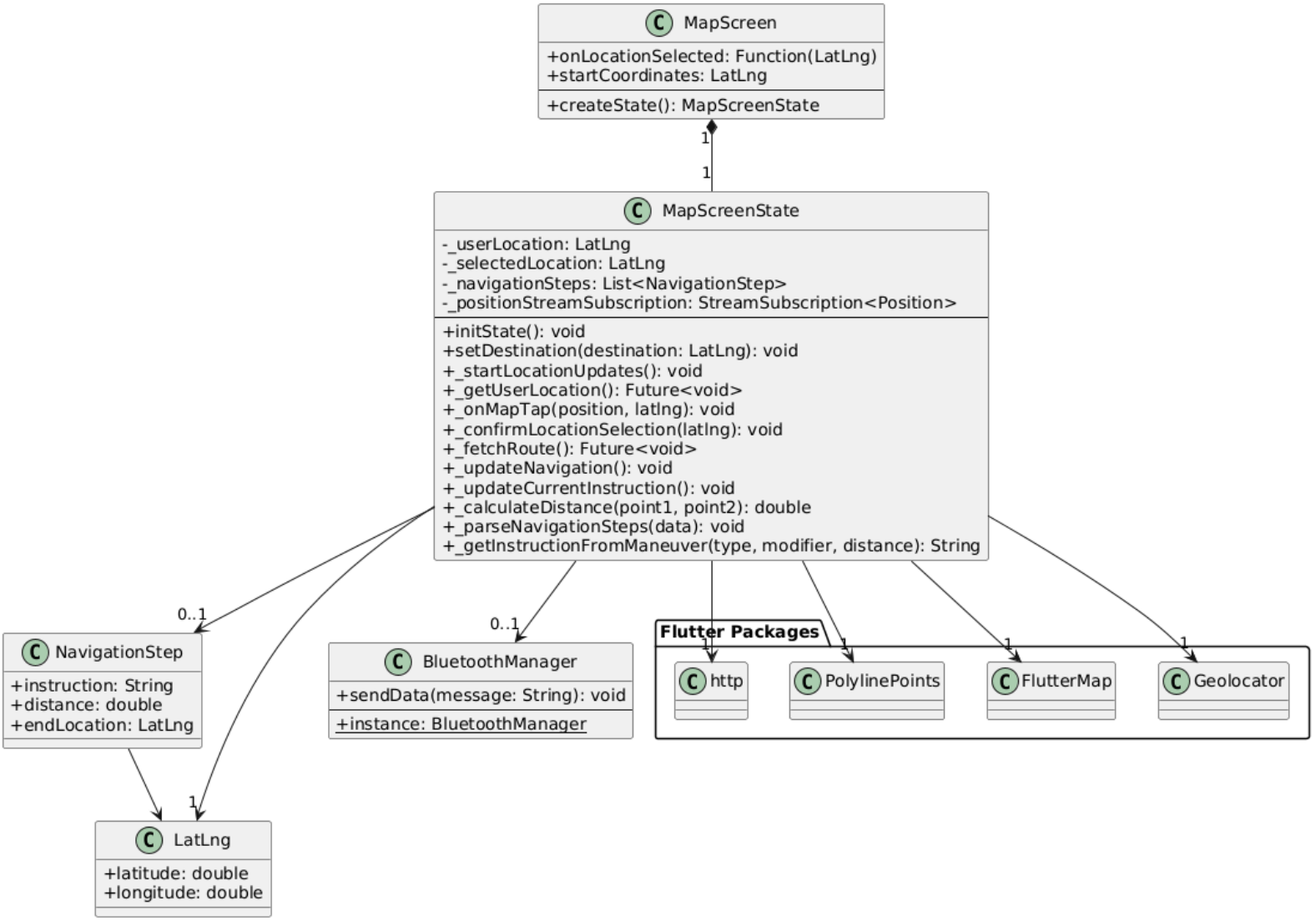
Location.dart



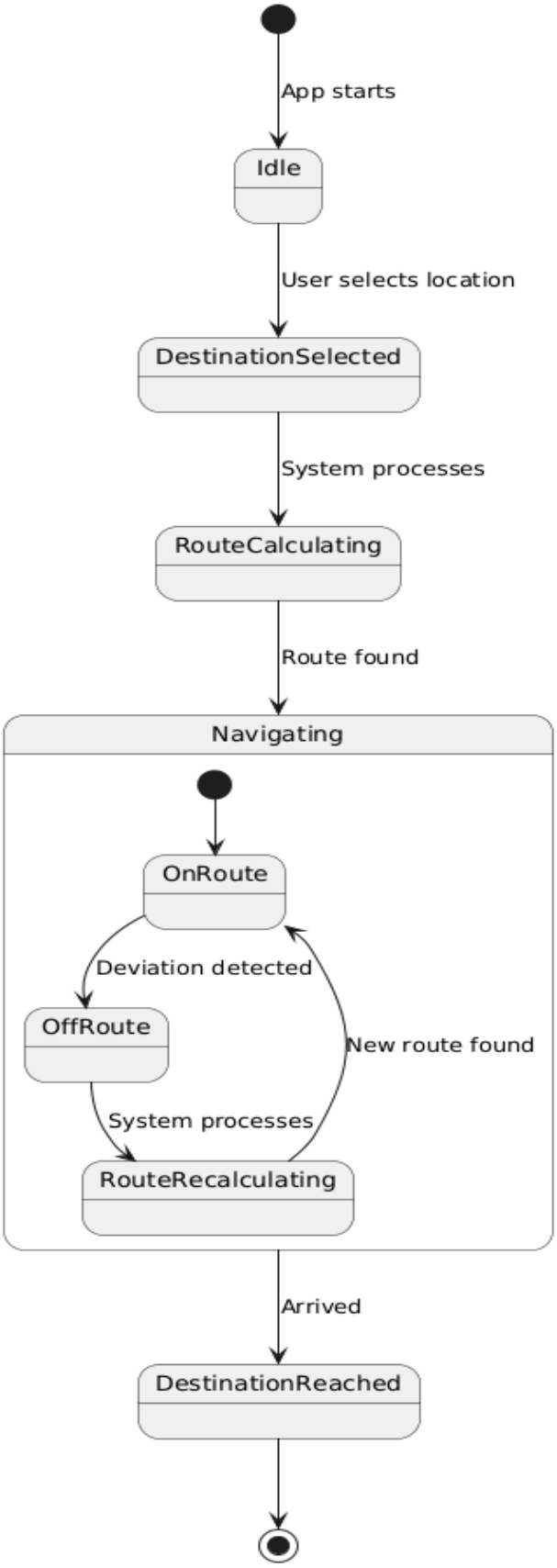
Bluetooth.dart



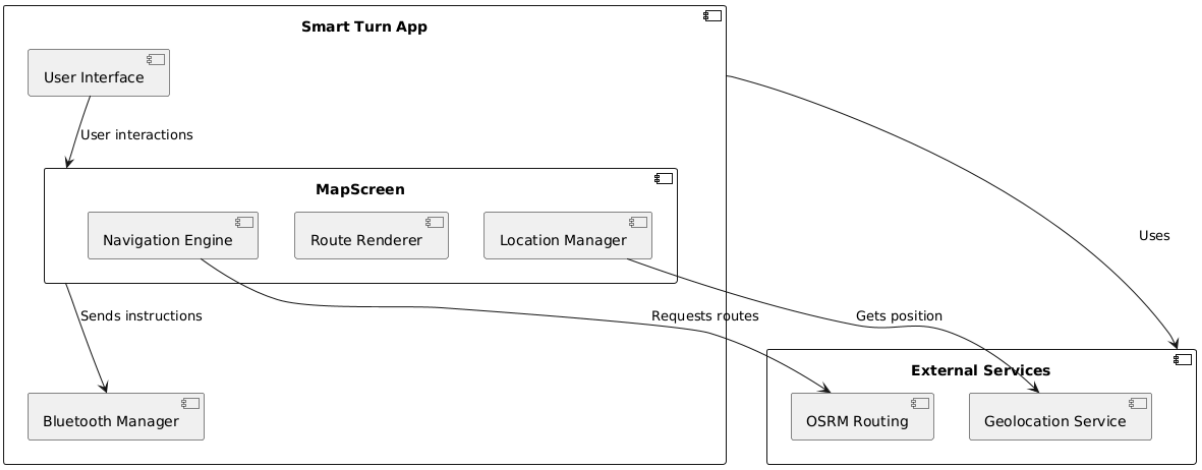
Navigation.dart



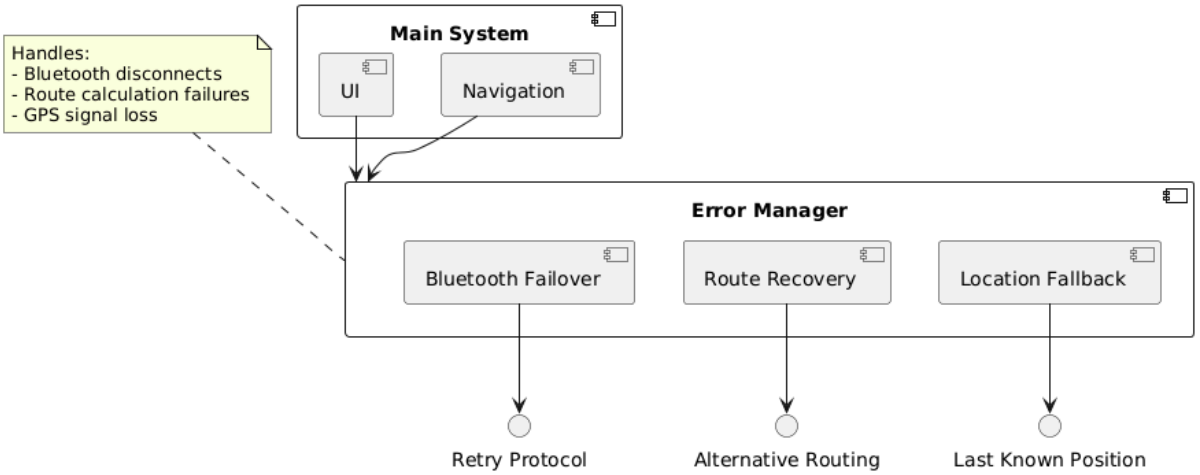
State Diagram



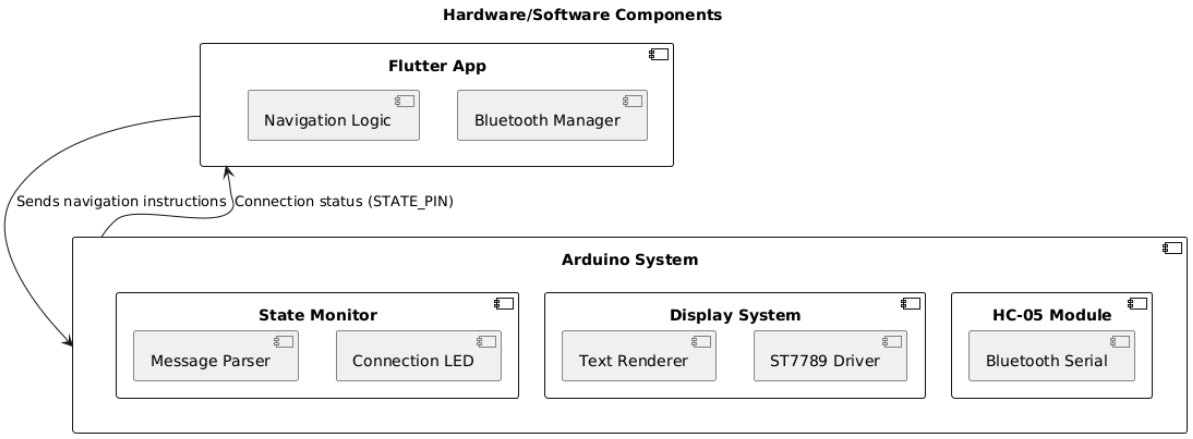
Component diagram



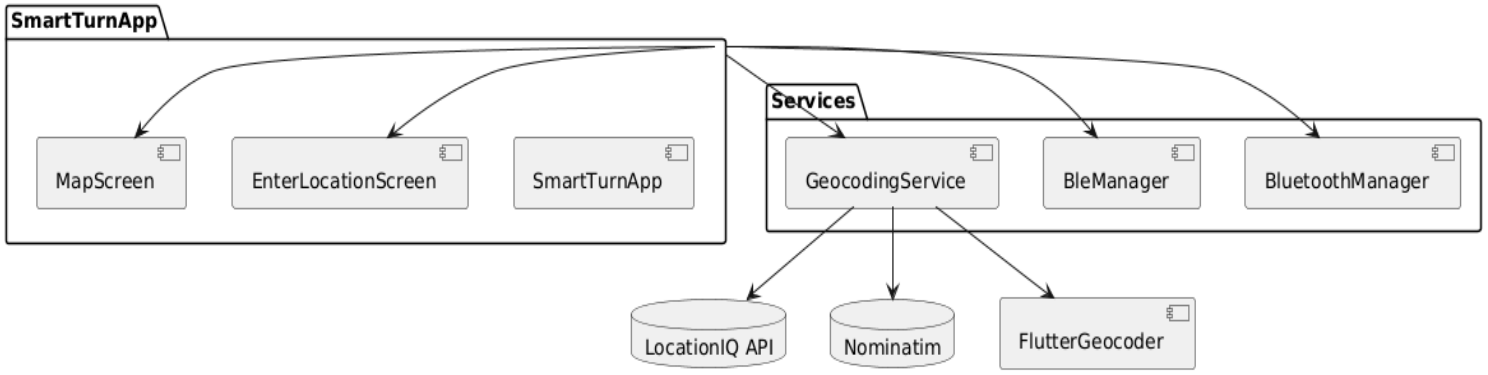
Failure possibilities



Flutter & Arduino



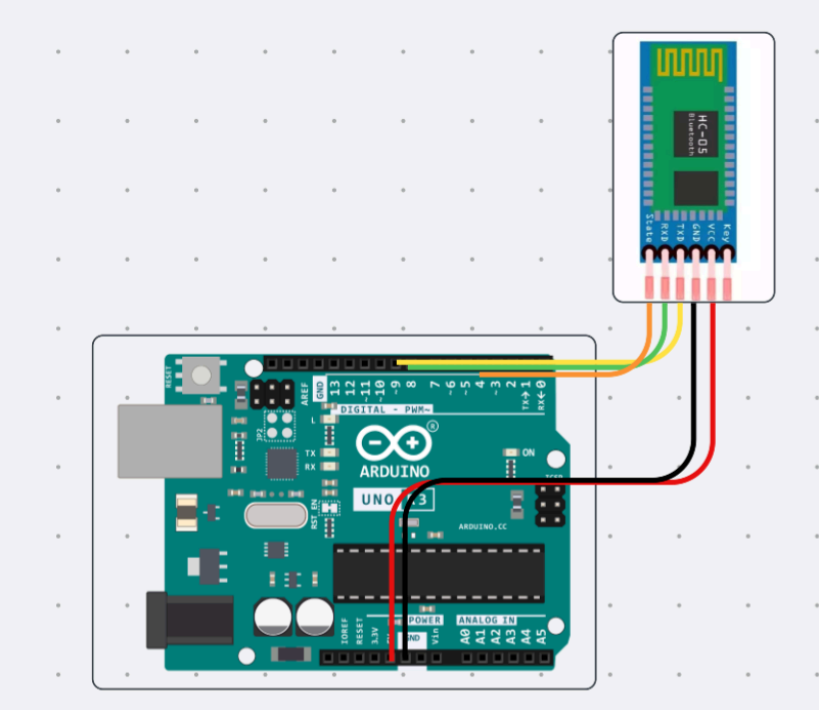
Interaction of components



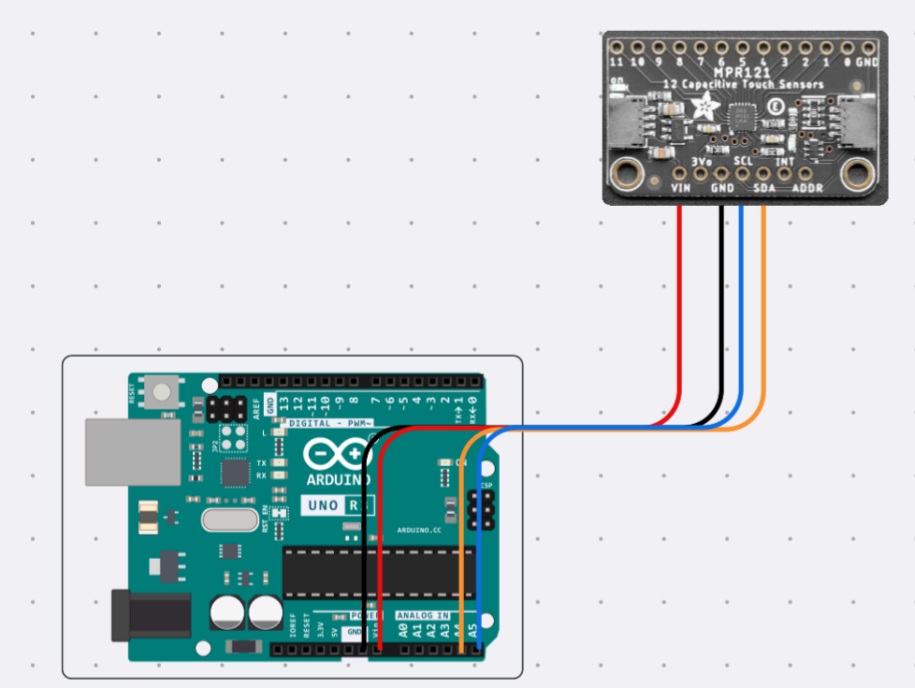
Hardware Wiring

Front Assembly

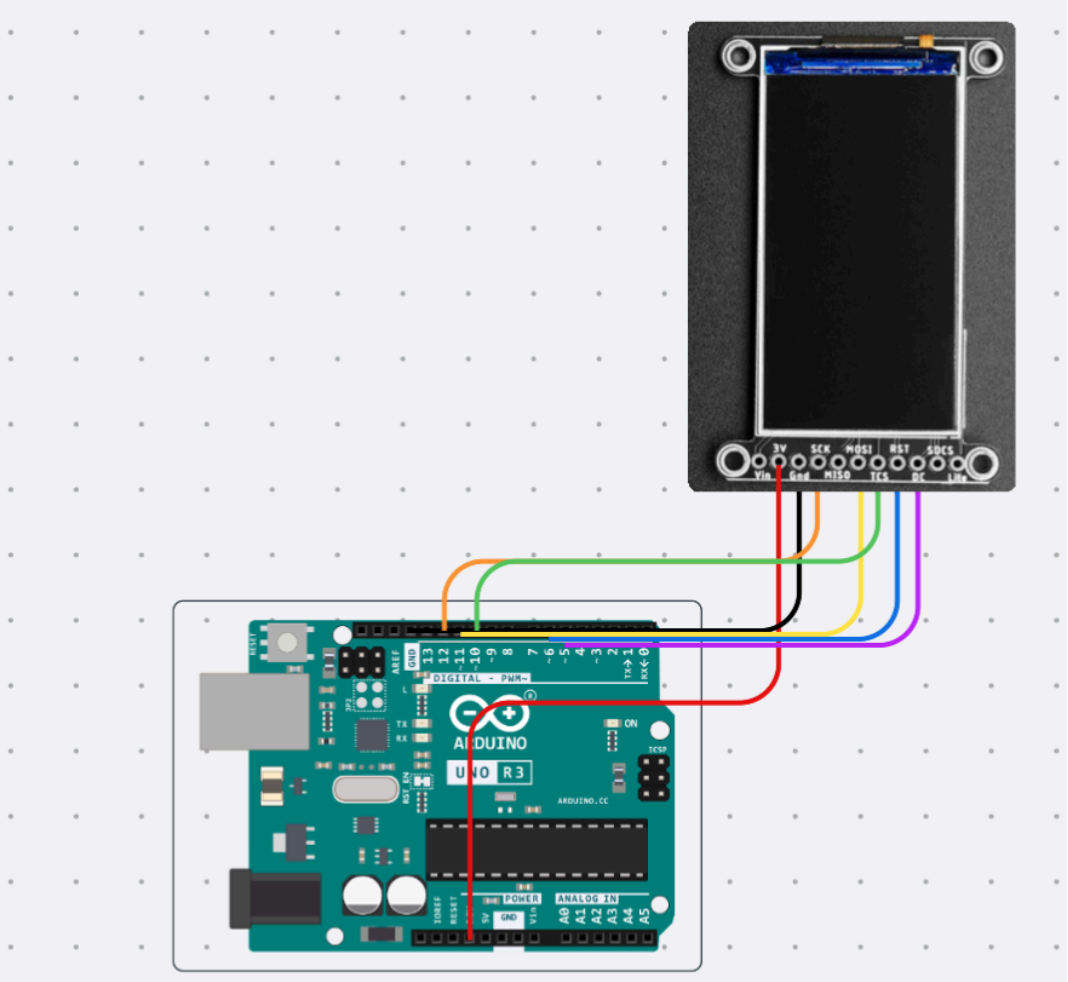
Uno and HC-05



Uno and Adafruit MPR 121



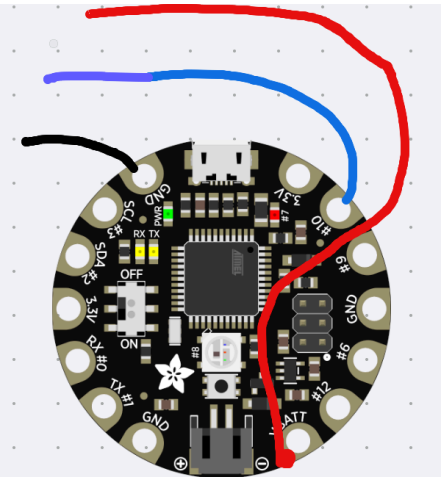
Uno and Adafruit Display - ST7789



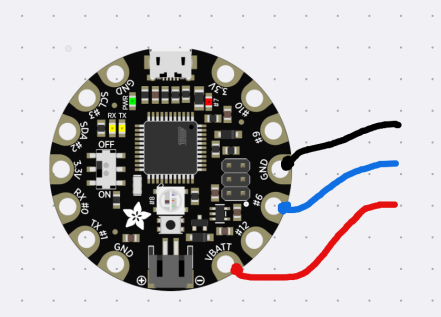
Rear Assembly

Flora Turn Signals

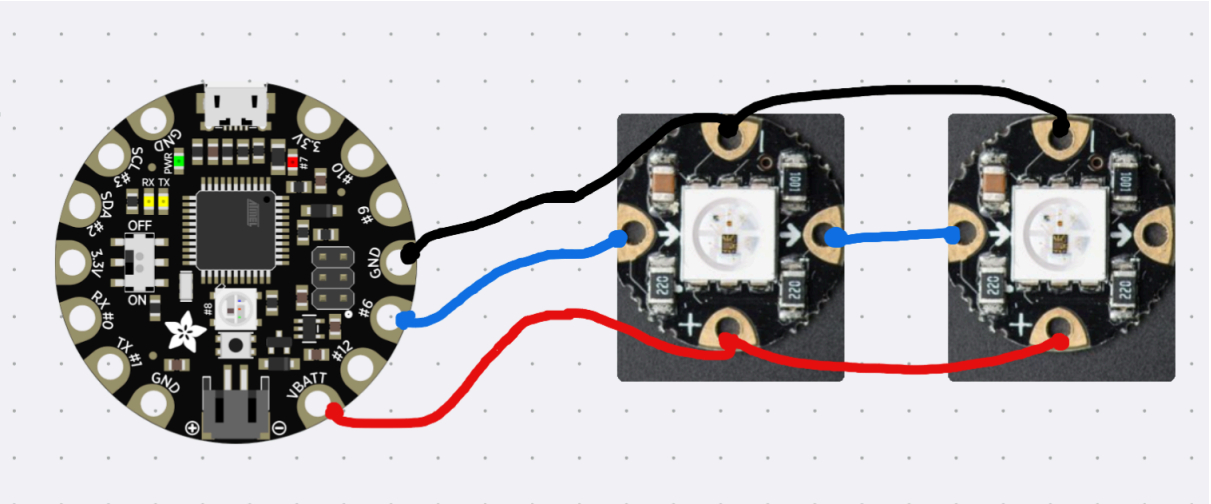
Start of Left Signal



Start of Right Signal

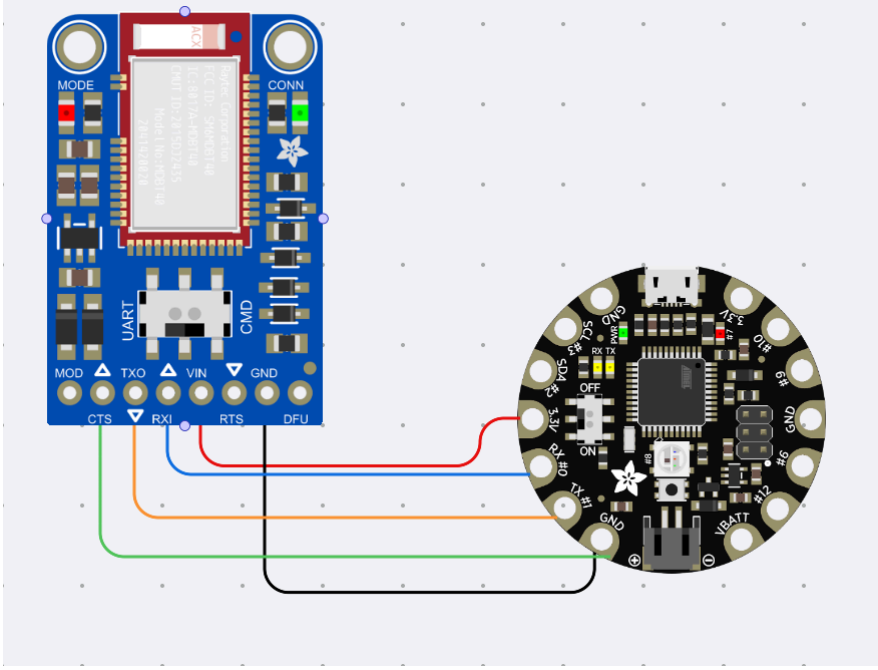


Each LED is connected in the following way:



Red is power, Black is ground, and Blue is signal the LEDs are chained together like into arrow shapes

Flora and Adafruit Bluefruit LE UART Friend



Problems Solved

Hardware - Arduino UNO

When initially setting up connections with the Arduino Uno and the HC05 module, as it was the first time using the hardware it initially made sense to connect the RX and TX pins directly to the Arduino RX (pin 0) and TX (pin 1). This led to serial communication port issues, however, as it seemed to be a problem everyone ran into the community was a great helping resolving the issue of just defining the RX and TX pins elsewhere.

Flutter Packages

While using Flutter it originally made sense to utilise the packages provided by the Flutter community such as its geocoder library which is currently still implemented in the system but only as a backup if the first two fail. Many changes had to be made to external API's for better performance, accuracy and overall reliability.

Permissions

While initially using the Flutter app it was important to ask the user for permission to use the device's Bluetooth and location services. This was originally implemented using Flutter's permission handler which at the start worked. However, as the system evolved and the package also evolved we came across conflicting version issues leading to the abandonment of the package which was resolved using the in-use package functionalities for permissions such as Geolocator & flutter_blue_plus.

Excessive API calls

For searching for locations using text input the original idea was to update the suggestions upon every character change, however, this led to excessive API calls and performance issues. To resolve this I used the same approach for the message passing, implementing a timer debounce to only call it every 300 milliseconds.

Conflicting bluetooth types

The two Bluetooth components that we ordered, the HC05 and Flora were both of different types of Bluetooth, classic and BLE. We overcame this by constructing a hardware bridge to handle the communication between both in the flutter app with the mobile phone acting as the hardware bridge and the logic handled in the background of the flutter application.

BLE Module characteristics not existing

Even Though the BLE module was connected to the mobile app and the messages were sent to it, they all failed with the error message of characteristic not found. The characteristics have a set ID in the BLE module but they need to be discovered on each connection for the flutter ble plus library to recognise them. This was resolved by scanning for characteristics each time before connecting allowing for smooth communication.

Background Execution

The app would not send location updates to the Uno when the app was running in the background. At first, we thought this was because of Bluetooth disconnecting but the connection persisted, after debugging we realised that the geolocation calls we were using stopped when the app was in the background. To fix this, we monitored the App’s state and when it was “not focused” we called a timer which executes every 3 seconds, where we update the user's location using a BackgroundGeolocation package and recalculate the route, once the route is updated we send the next instruction to the screen.

Testing

Unit, Integration, System and User testing was performed for this project, please see the Smart Turn Testing document which is available in our repo in ***docs/documentation.***

Key points from user feedback:

The screen is too dim when outside.
There was no indication of which turn signal was on the screen.
Sometimes the screen would not clear the previous information fully.

We incorporated the feedback given and tested it. This can be seen in the System testing section.
Sadly we could not change the brightness of the screen as it was already at its peak brightness.

Division of work

Jakub

- Indication vest
- Touch gesture
- BT Hardware bridge via App
- Displaying Arrows for indicators on the screen
- Testing, Documentation
- Research of Hardware
- Background Execution of the application.

Josh

- Flutter Application
- HC05 message passing
- Displaying navigation information
- Gitlab CI/CD
- Testing
- Documentation
- Research of Hardware

Future Work

Our project ended up with all the functionality we planned but if we had more time to work on this project we would do the following:

- Make it waterproof so it can be used in any condition.
- Make it fully IOS compatible.
- Add some basic encryption to the Bluetooth messages.
- Create a better mounting system for it & smaller hardware.
- Add a brake light to indicate when the user is braking.

References

1. Geocoding. [online] *pub.dev*. [Accessed 25/02/2025], from https://pub.dev/packages/geocoding
2. Geolocator. [online] *pub.dev*. [Accessed 02/03/2025], from https://pub.dev/packages/geolocator
3. Latlong2. [online] *pub.dev*. [Accessed 09/03/2025], from https://pub.dev/packages/latlong2
4. HTTP. [online] *pub.dev*. [Accessed 12/03/2025], from https://pub.dev/packages/http
5. OSM Nominatim. [online] *pub.dev*. [Accessed 30/03/2025], from https://pub.dev/packages/osm_nominatim
6. Flutter Blue Plus. [online] *pub.dev*. [Accessed 12/03/2025], from https://pub.dev/packages/flutter_blue_plus
7. Flutter Map. [online] *pub.dev*. [Accessed 15/03/2025], from https://pub.dev/packages/flutter_map
8. Flutter Polyline Points. [online] *pub.dev*. [Accessed 15/03/2025], from https://pub.dev/packages/flutter_polyline_points
9. Flutter Background Geolocation. [online] *pub.dev*. [Accessed 30/04/2025], from https://pub.dev/packages/flutter_background_geolocation
10. Adafruit GFX Library. [online] *Arduino Docs*. [Accessed 17/03/2025], from https://docs.arduino.cc/libraries/adafruit-gfx-library/
11. Adafruit MPR121. [online] *Arduino Docs*. [Accessed 20/03/2025], from https://docs.arduino.cc/libraries/adafruit-mpr121/
12. Adafruit NeoPixel. [online] *Arduino Docs*. [Accessed 28/03/2025], from https://docs.arduino.cc/libraries/adafruit-neopixel/
13. Adafruit ST7735 and ST7789 Library. [online] *Arduino Docs*. [Accessed 03/04/2025], from https://docs.arduino.cc/libraries/adafruit-st7735-and-st7789-library/
14. Arduino.h - ArduinoCore-avr. [online] *GitHub*. [Accessed 10/04/2025], from https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/Arduino.h

15. Installing the BLE Library. [online] *Adafruit Learning System*. [Accessed 14/04/2025], from https://learn.adafruit.com/adafruit-feather-m0-bluefruit-le/installing-ble-library
16. Introducing the Adafruit Bluefruit LE UART Friend. [online] *Adafruit Learning System*. [Accessed 18/04/2025], 2025, from https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-uart-friend/introduction
17. SoftwareSerial. [online] *Arduino Docs*. [Accessed 20/04/2025], from https://docs.arduino.cc/learn/built-in-libraries/software-serial/
18. SPI. [online] *Arduino Docs*. [Accessed 22/04/2025], from https://docs.arduino.cc/language-reference/en/functions/communication/SPI/
19. Wire. [online] *Arduino Docs*. [Accessed 30/04/2025], from https://docs.arduino.cc/language-reference/en/functions/communication/wire/
20. LocationIQ [online] Accessed 22/04/2025, from https://locationiq.com/
21. OSRM [online] Accessed 20/04/2025, from https://project-osrm.org/docs/v5.24.0/api/#
22. PlatformIO [online] Accessed 23/04/2025, from https://docs.platformio.org/en/latest/advanced/unit-testing/index.html
23. Mockito [online] Accessed 23/04/2024 from https://pub.dev/packages/mockito
24. Pump And Settle [online] Accessed 23/04/2025 from, https://api.flutter.dev/flutter/flutter_test/WidgetTester/pumpAndSettle.html