Architecture

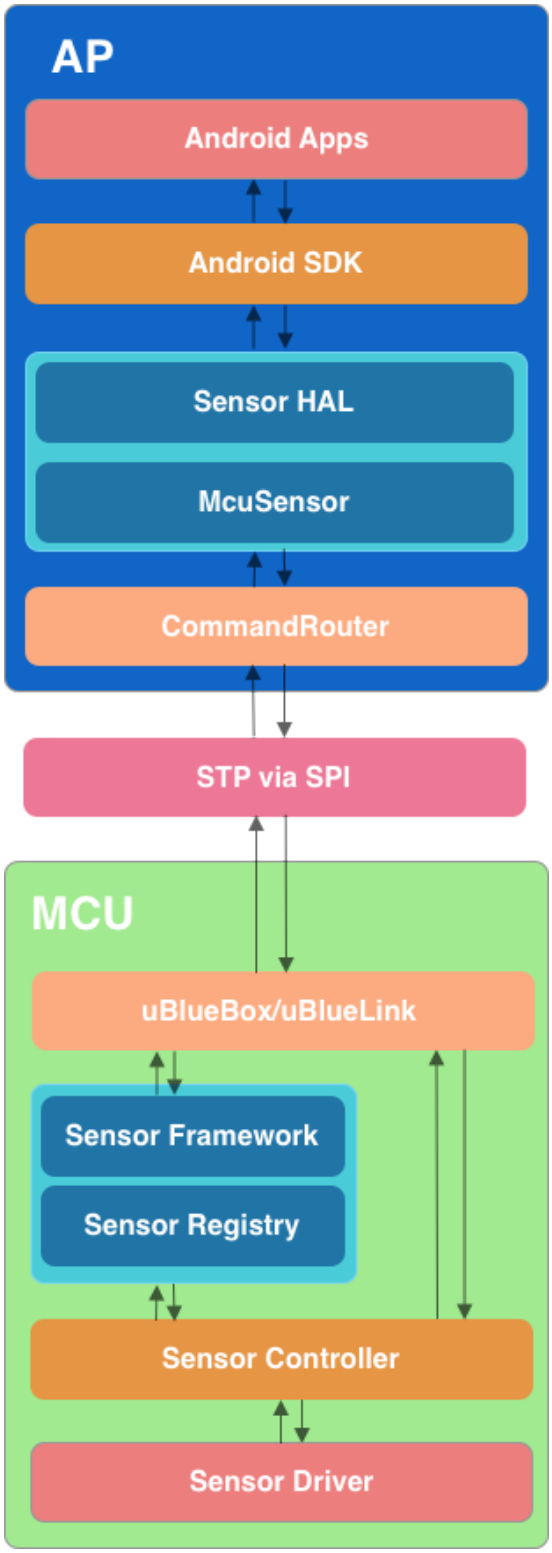
Overview

To help reduce power consumption, Merlot is designed to have most of its sensors connected to an MCU which enables us to put the AP to sleep more often. Depending on the scenario, the MCU will collect sensor data and provide it to the current consumers (which may be on the MCU, the AP, or both).

This page will go over the high level architecture of how we connect to sensors and share their data with consumers and introduce key terminology. There are other sub0-pages which will go into more detail on how specific pieces work (like adding a new sensor type).

Sensor Architecture

The diagram below shows the main components that enable sensor usage on Merlot. At a high level you have the Application Processor (AP) which handles requesting and receiving sensor data and providing it to the Android layer. In the middle you have the transport layer between the MCU and AP which allows the data to flow, and at the lowest level is the Microcontroller (MCU) which is where most of Merlot sensors are located.



MCU

SENSOR DRIVER

The sensor driver contains all the code for communicating with the actual sensor hardware. The driver exposes utilities to allow for configuration of the device (setting sampling rate, orientation, power modes, etc) and the ability to read data from the sensor. For Merlot, the main interface to these drivers is via the Zephyr Sensor APIs.

SENSOR CONTROLLER

The sensor controllers (IMU controller for example) are responsible for invoking the capabilities exposed in the Sensor Driver to configure the sensors correctly for Merlot product use cases. The controller will receive requests from the Sensor Framework to do things like enabled/disable the sensor, or change the sampling rate. The controller is also responsible for publishing sensor data so that consumers can receive the raw sensor data. Data is published to uBluebox where it can then be subscribed to from other services on the MCU and/or the SensorHal on the Android side.

SENSOR FRAMEWORK

The Sensor Framework handles requests and response for all available sensors. All available sensors are registered with the sensor framework via the sensor registry. Requests to the sensor framework contain a sensor id which the sensor framework uses to route the request to the correct sensor controller.

UBLUEBOX/UBLUELINK

uBluebox is a generic pub/sub framework we use to transport sensor data. uBluebox on its own is only able to communicate locally on the MCU so to publish messages to the AP we also use uBlueLink. We have 2 main types of messages we send via uBluebox for sensor use cases: **command** messages are used to configure the sensor (enable/disable, set sampling rate), and **data** messages are used to send and receive data from the sensor. A simple example of this would be how the Android HAL (which we will cover later) would handle step counting. When a user requests to start receiving the step count, the Android HAL will send a uBluebox message like *enable sensor id <pedometer id>*. At the same time the Android HAL will subscribe to the pedometer data messages using the message topic *"msgs:sensors:Ped*".

STP via SPI

While uBlueLink is used to allow uBlueBox messages to cross device boundaries, it is not a full transport layer on its own. To handle passing data between the AP and MCU we use the *Synchronized Transport Protocol (STP)* which operates on the SPI bus. Understanding STP is not critical for sensor development so we won't go into much detail here.

AP

COMMAND ROUTER

Command router is (somewhat) the equivalent to uBluebox/uBluelink on the AP side. It allows for sending and receiving of uBluebox messages. We could have used BlueBox or ported uBlueBox to Android to do the same, but CommandRouter was already created and working for the Stella project so we decided to use it as well.

MCUSENSOR

McuSensor.cpp is the base class for all MCU sensors which we expose to Android. McuSensor is part of the SensorHal and handles functionality which is common across all MCU sensors. This includes sending requests via Command router to start/stop the sensors when needed, and registering the subscribers to receive data from the sensors.

Each sensor type on the MCU will have an implementation in the Sensor HAL which extends McuSensor. The implementations are responsible for deserializing the received sensor data and exposing it to the Android SDK. An example of this can be seen in PedometerSensor.cpp

SENSOR HAL

The Android Sensor Hardware Abstraction Layer (HAL) is where we register sensors to make them available to the Android SDK. As the name suggests, the HAL abstracts all the low level hardware code (and in this case the AP to MCU communications as well) and provides a simple interface to the Android SDK for communicating with sensors. Android supports a number of sensors by default. For sensors that are already defined in Android we just need to add an implementation for that sensor and add the sensor type to the list of currently available sensors for our device. If a sensor does not exist, we will also need to define the new sensor type, the data format it will expose, and create an ID from the sensor which applications will reference. See the AOSP sensor documentation for more details.

ANDROID SDK/APPS

These blocks are included in the diagram to visualize the whole stack, but in reality there is nothing unique we do at these levels to enable sensor support. Because we conform to the Android sensor HAL, any generic Android applications are able to communicate with our sensors.

« Sensors

Android Sensor Listener Overview »

)}