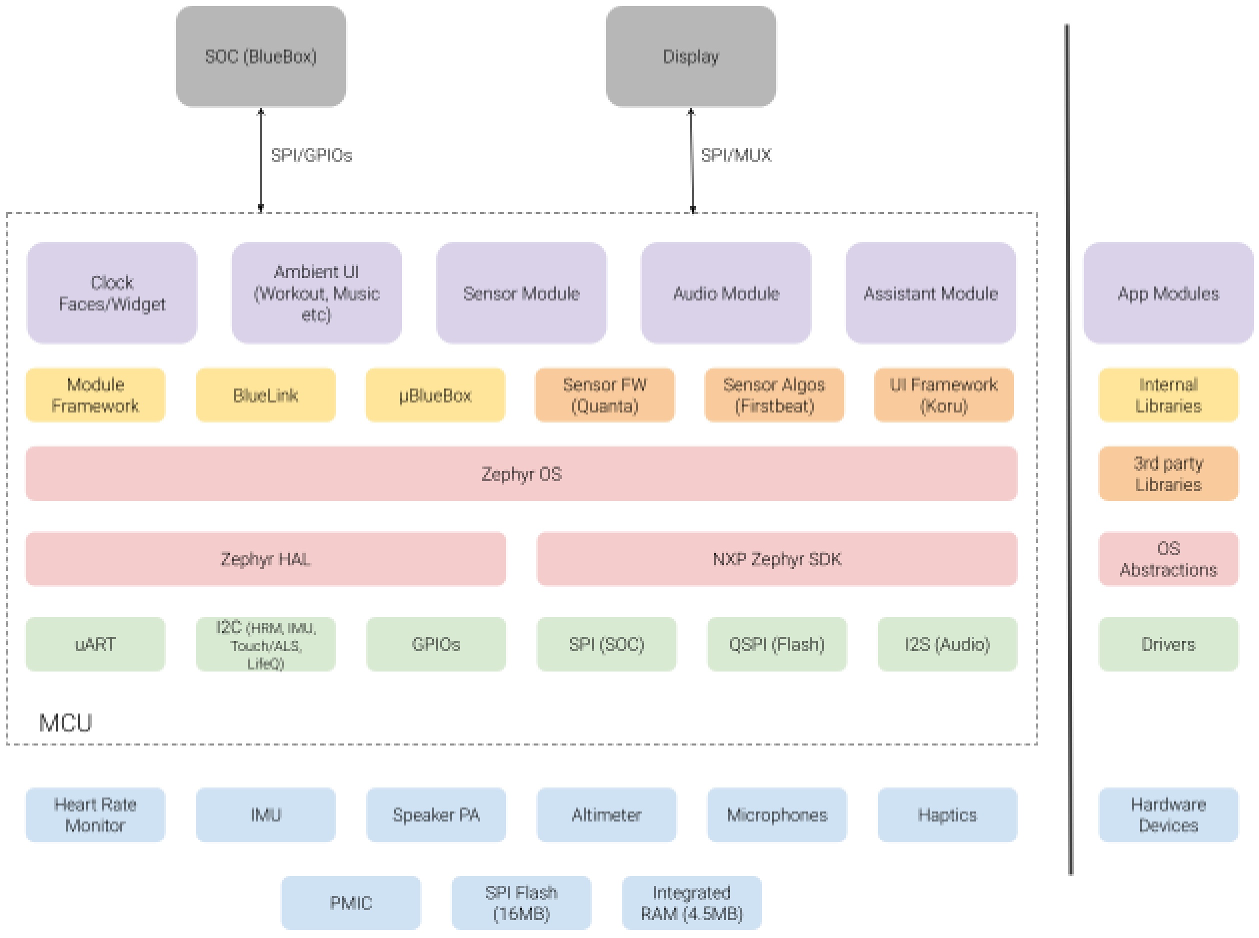
[WIP] Milan MCU Software Architecture

This doc is a high level description of the MCU architecture. We'll link to more detailed documents from here. *All diagrams are created inline with Google Drawings; you can double click to edit or export.*

Table of Contents (Click to jump to a section)

SW Block Diagram



Double-click to edit or download

Why do we have an MCU?

Milan has a “big-little” architecture: we have an SOC (system on chip), which is a powerful chip from Qualcomm that runs Android, and an MCU (microcontroller), which is a low power chip. In our architecture, the MCU is always on, acting as a sensor hub, running detection algorithms, listening for assistant wakewords, and driving an always-on display. The SOC is generally kept in a low power state (often referred to as AIR1), and is brought into full-power mode during active usage (apps, camera etc) or when notifications are received. This big-little architecture lets us conserve power while still having a powerful, capable device.

The MCU and SOC communicate via a SPI and I2S (serial peripheral interface) bus, along with some GPIO pins. The SOC is a QC3300 chip (also referenced as SDM429W). The MCU is an NXP RT600 chip. Our current plan is for the MCU to have 16MB flash and 4.5MB RAM.

Block Diagram Components

Hardware Devices

The MCU acts as a sensor hub; many sensors (heart rate, IMU etc) are connected to it, as are other things like audio and display. See Milan MCU Architecture to see the hardware architecture. *Note: this doc's block diagram is not exhaustive, please read the hardware document linked for an exhaustive list of hardware peripherals and a HW block diagram.*

Drivers

We have a set of device drivers for our peripherals. Notably:

- I2C for IMU, Altimeter, HRM AFE, Touch, ALS
- I2C for PMIC, LRA and the HRM Boost
- I2S for Audio (x2, one for SOC<-> MCU and one for MCU<->Speaker)
- Dual SPI for the MCU Flash
- SPI and GPIOs for connecting to the SOC
- LifeQ, which provides algos+device drivers
- uART for flashing the firmware and key provisioning

Drivers come from three sources:

- Written by us
- Written by our JDM Quanta (and possibly modified by us)
- Written by device manufacturers (and possibly modified by Quanta/us)

OS

We’re using the Zephyr RTOS as our OS. We chose it over FreeRTOS for (among other reasons) its security model and because Orion is also using it. Zephyr comes with a Hardware Abstraction Layer for higher level code to not have to deal with specifics of hardware. NXP also gives us an SDK for Zephyr to work properly on the RT600.

3rd Party Libraries

We’re using:

- A sensor framework from Quanta (see details in the Sensor Framework section)
- Koru, a UI framework that lets us write Javascript and XML instead of C
- Firstbeat, a sensor algorithm provider

Internal Libraries

We're using:

- A module framework, which is an abstraction over Zephyr. It lets us write high level code while ignoring the specific RTOS mechanisms and implementation details.
- uBlueBox and Bluelink, which together provide a pub/sub system and mechanism for us to communicate with the SOC.

App Modules

This is the application level code we write. We have

- Watch faces and widgets: The Always-on display, with watch faces and complications on them. On interaction, we generally handoff to the SOC.
- Ambient apps: Always on display for apps started on the SOC. For example, glance at the screen to see the current status of your workout, or the current song playing.
- Sensor module: the high level code that provides sensor data to MCU apps and communicates sensor data and configurations with the SOC.
- Assistant: Handles "Hey Facebook" or similar.
- Audio: audio playback and music.

MCU Frameworks/Mechanisms

MCU Boot Flow

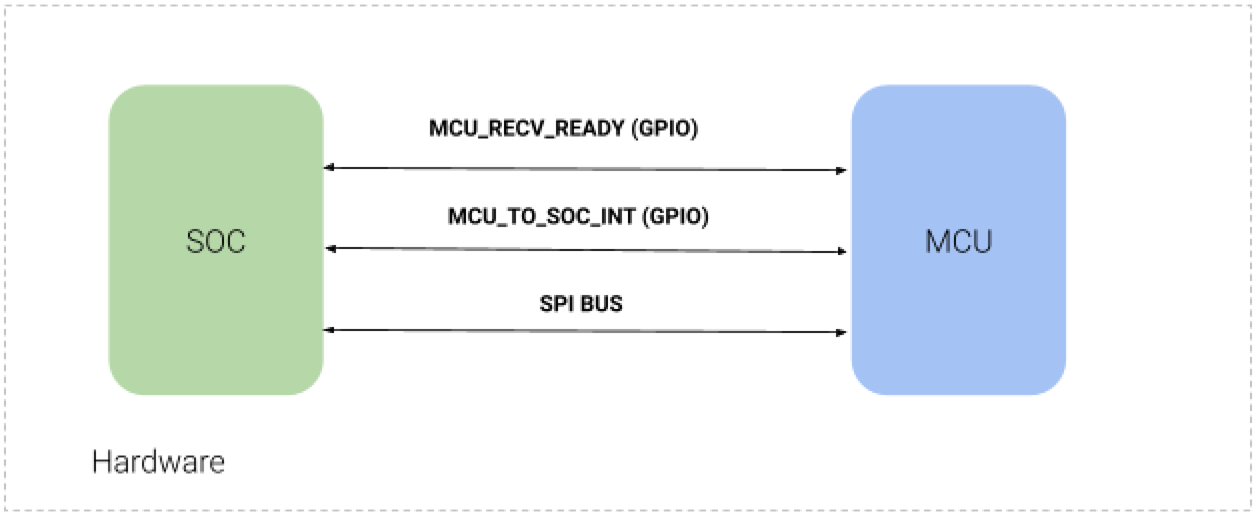
The very first time we boot Milan, the SOC will boot the MCU. We ship a prebuilt binary of MCU firmware with our AOSP codebase; during SOC boot (i.e. Android boot), we run a script that flashes the MCU with this firmware and boots it. The firmware is written to MCU flash. You can view the current code for this by running `godir mcu_flash.sh` in your AOSP repository.

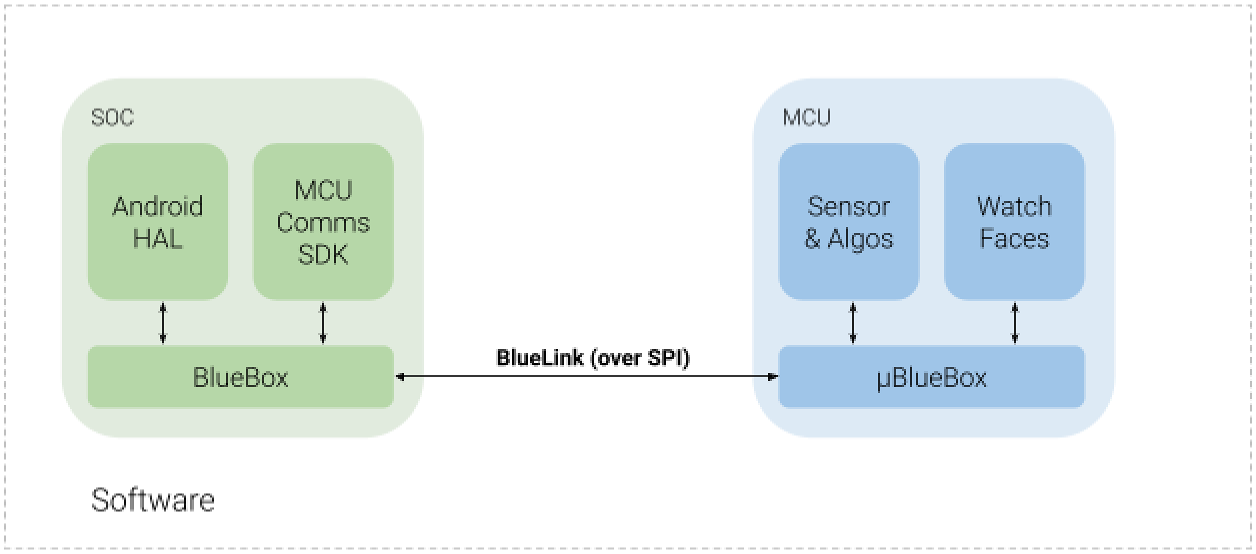
During subsequent boots, the MCU boots by reading the firmware from SPI flash. We can also ship updates to the MCU firmware over-the-air; when the SOC receives these software updates, it can reflash the MCU.

TODO: Add more details + diagrams (Stas).

SOC-MCU communication

We use a SPI bus for communication between the SOC and the MCU. In addition to the SPI, we have 2 GPIO pins to implement flow control. The SOC is the SPI master.





In hardware:

- We have a SPI bus between the MCU and SOC for them to communicate.
- We have 2 GPIO pins (marked MCU_RECV_READY and MCU_TO_SOC_INT in the diagram). Those are respectively for the MCU to signal that it can receive more data (because the MCU has limited memory and CPU, we may need to chunk data) and for signalling interrupts between the two.

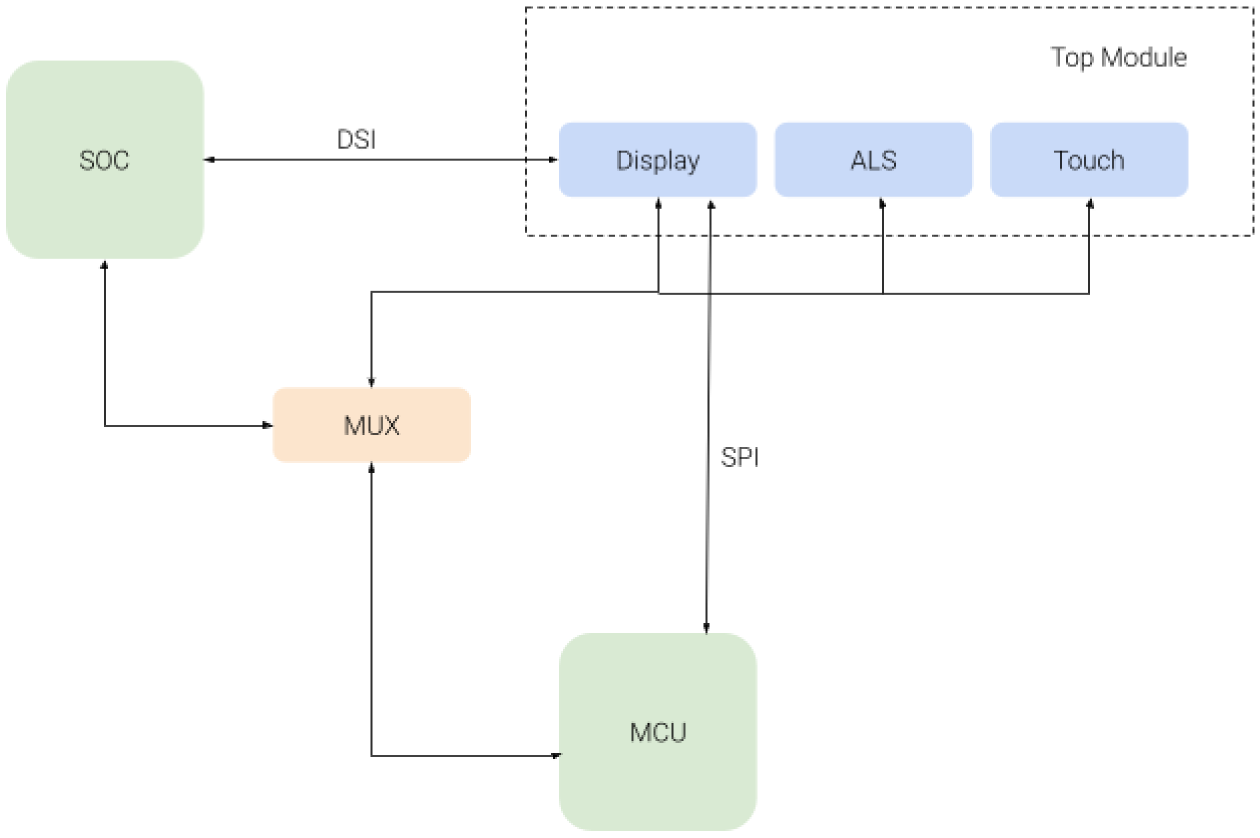
On the SOC:

- The BlueBox module handles all communication between SOC and MCU. It does this via a pub/sub system where there are messages and topics. See this wiki for more.
- There are components that want to talk to the MCU. For example:
  - Sensors: the implementation of the Sensor HAL talks to the BlueBox server to get sensor data
  - MCU Comms SDK: If apps or widgets need to directly talk to the MCU, they go through Bluebox. For example, the always-on display powered by the MCU may need data from SOC. Similarly, if an app (e.g. a workout app) wanted to display a glanceable MCU powered screen, it would provide data using this SDK.

On the MCU:

- MicroBlueBox is the MCU implementation of BlueBox.
- There are components that want to talk to the SOC. For example:
  - The Sensor module will receive configuration and other info from the SOC (e.g. required latency or accuracy for sensors, activating/deactivating sensors etc) and communicates sensor data back
  - Watch faces may need to get data from the SOC. Since all networking radios (Wifi, BT, LTE) are on the SOC, the MCU will need to wake up and ask the SOC to fetch data,
  - Other use cases e.g. audio or assistant also exist
- Algorithm modules will get sensor data from from sensor controllers via MicroBlueBox

Display MUX



The display is connected to both SOC and MCU. The SOC is connected to the display via DSI, and can drive the display at 45fps. The MCU is connected to the display via SPI, and can drive the display at max 5fps (practically, 1fps). In addition to the display, the top module also contains an ambient light sensor (ALS) and a touch module to detect touches.

Since both are connected to the MCU, there is a hardware multiplexer (MUX) that controls which chip is driving the display. Since the MCU is always on, it's the master of the display MUX.

SOC → MCU Transition:

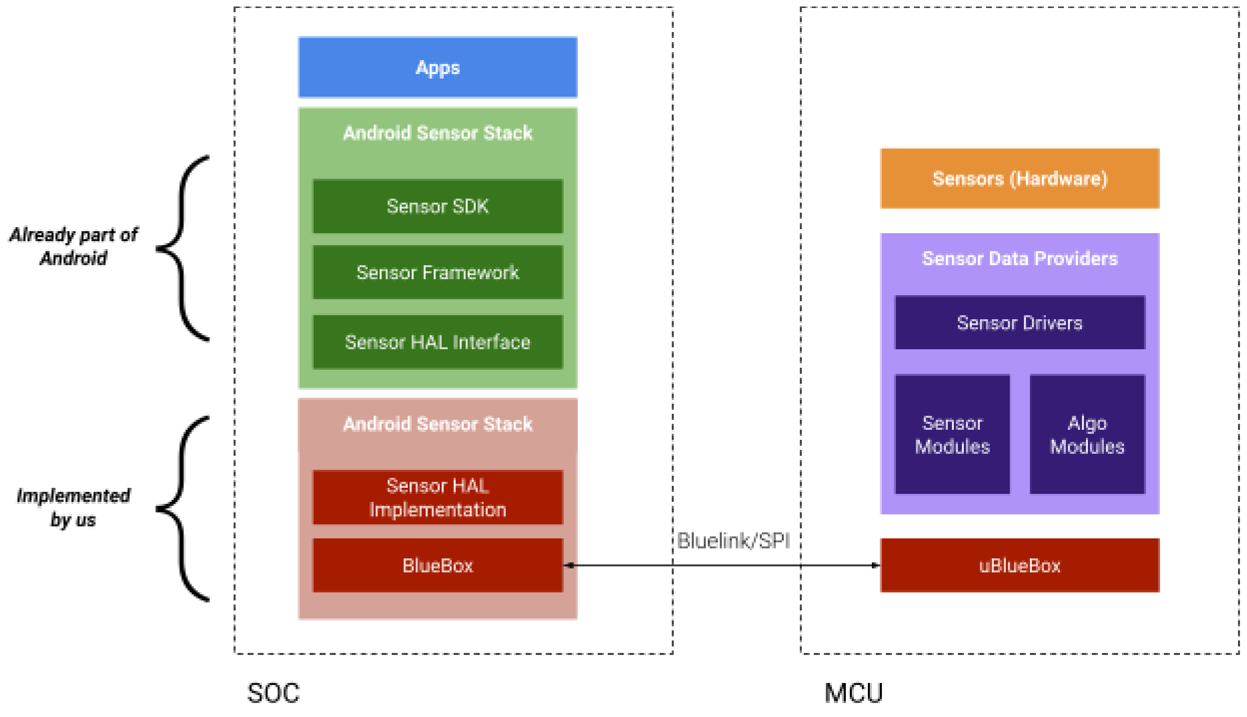
- The SOC transfers the display content to the MCU over SPI (SPI interrupt)
- The MCU sets up its display pipeline
- SOC requests that we switch from DSI to SPI via DSI (*To be confirmed, cc David Liu*)
- MCU updates DSI as well as switching to touch/ALS control

MCU → SOC Transition:

- MCU alerts SOC of touch via a GPIO interrupt
- SOC confirms this and starts driving via DSI
- MCU hands off touch/ALS control

MCU Subsystems and Applications

Sensor Framework



At a high level:

- On the SOC:
  - Android comes with a sensor stack that includes developer facing APIs (SensorManager) as well as frameworks that handle multiplexing with the HAL (the HAL is single-client)
  - Android also defines the interface for the Sensors HAL
  - We implement the Sensors HAL. We use BlueBox to communicate both ways - we send configuration to the MCU (e.g. activate/deactivate sensors, configure latency etc) as well as receive sensor data.
- On the MCU:
  - Most of our sensors are connected to the MCU
  - We (i.e. us + Quanta) implement sensor drivers on the MCU
  - We also integrate Quanta's sensor framework and algorithms/drivers from LifeQ and FirstBeat on the MCU

    - Finally, we send any sensor data to the SOC over Bluebox

For more detail, see **Sensor Framework High Level Design**

Always-on Display

The MCU is connected to the display through a SPI bus. When the user is actively using Milan (taking pictures, using apps etc), the display will be driven by the SOC. When the device is inactive, we will put the SOC into a low power state and drive the display from the MCU.

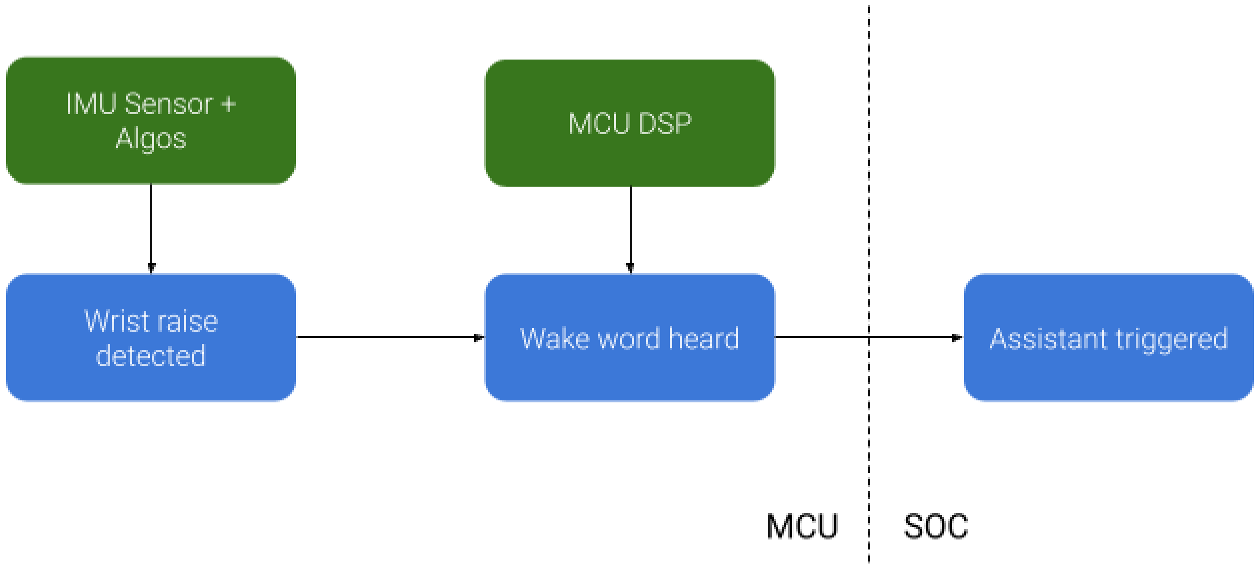
Notable details:

- The SPI bus to the display is a 25Mhz bus. This theoretically lets us update the display at <5fps (474x474 pixels, times 3 colors, times 9 bit, times 5fps ≈ 30Mhz). In practice, we are assuming this to be 1fps since we can't perfectly utilize the bus bandwidth, and because anything below 30fps makes scrolling/interactivity bad.
- There are 2 modes for the always on display:
    - AOD Clock Faces: A single screen with various widgets e.g. time, weather, calendar, heart rate etc.
    - Ambient apps: An AOD screen for apps started on the MCU. For example, you start a workout app on the SOC; when you start working out, you have a glanceable screen with time elapsed, heart rate, calories burned etc.
- On touch, we immediately transfer control to the SOC, since we can't update the display at a high frame rate from the MCU.

For more details, contact Dalton Flanagan or read these docs:

- Initial AOD Research
- Levers to raise SPI frame rate

Assistant



At a high level, the assistant works in 3 steps:

1. First, we need to detect a wrist raise. This is based on measurements from our IMU sensor
2. Next, we detect the wake word (e.g. “Hello Facebook”). This is done by processing audio data through the MCU DSP
3. Finally, if we detect the wake word successfully, we pass the data over to the SOC where the real assistant runs and responds to the user

There are open questions around how timeouts, false positive rate etc. For more details, contact Zhong Zhang and read the docs below:

- Milan Wakeword PRD
- Stella Wakeword Design
- Audio SW arch

Audio

Milan | Audio Architecture

Audio TODO: Marc Salem, Michael Asfaw.