User manual

AOS user guide

v1.0

Revisions

Version	Primary Author	Description of Version	Date Completed
1.0	Patrick Beatini	Initial draft	June 13th, 2024
	Hamza SAADI	Add sections 2.4.8; 6.8; 6.9; 6.10	July 8th, 2024

Table of Contents

1 Introduction

This document is a guide providing the internal AOS SDK architecture and implementation.

1.1 Document Purpose

The document covers the architectural aspect of AOS as well as its implementation. The dynamic behavior of the operating system is also deeply discussed.

1.2 Intended Audience and Document Overview

The intended audience for this document is restricted to the engineering team.

1.3 Definitions, Acronyms and Abbreviations

The terms *Software* and *Firmware* are used interchangeably in this document.

Acronym	Description
AOS	Abeeway Operating System
BLE	Bluetooth Low Energy
LoRa	Low Range Radio
AOS-SDK	Abeeway Operating system, Software Development Kit.
CLI	Command Line Interface
GNSS	Global Navigation Satellite System
A-GNSS	Aided GNSS
BOM	Bill Of Material
HAL	Hardware Abstraction Layer
PWM	Pulse Width Modulation
ADC	Analog to Digital Converter
LBM	LoRa Basic Modem. Stack provided by Semtech.

2 Getting started

This section covers the basic knowledge to be aware of before starting a new development.

2.1 Overview

AOS, the Abeeway Operating System, and its associated Software development Kit (AOS-SDK), have been created in support to the Abeeway geolocation module.
The module is targeted to simplify the BOM and to reduce the time to market of hardware manufacturer. Note that the lnext generation of the egacy Abeeway boards will also include it.

We remind that the Abeeway geolocation module embeds the following main components:

- A main micro-controller STM32WB55 supporting Bluetooth Low Energy
- A LR1110 supporting LoRa, WIFI sniffing and A-GNSS.
- A MT3333 providing GNSS and A-GNSS positions.

For some specific configuration, it is recommended to read some specific part of the STM32WB55 user manual. Note that when such a knowledge is required, the document will provide the needed reference to the ST manual.

2.2 Installation

The AOS SDK is provided as a single tarball. The content should extracted at the desired location. Once extracted, the directory **AOS-SDK-public** will be automatically created. This directory contains two sub-directories

- lib/release: Contains the static AOS library. Applications using the SDK should link against this library.
- include: Contains API header files.

Notes

- To use the SDK, the programming language should be C or C++.
- Any GCC based cross-compilation chain can be used. However, It is advised to use the STM32CubeIDE **version 1.15** or above freely provided by ST Micro-electronic.
- The module is built around the STM32WB55 component, which contains an ARM cortex M4. The compilation tools should support this type of processor.

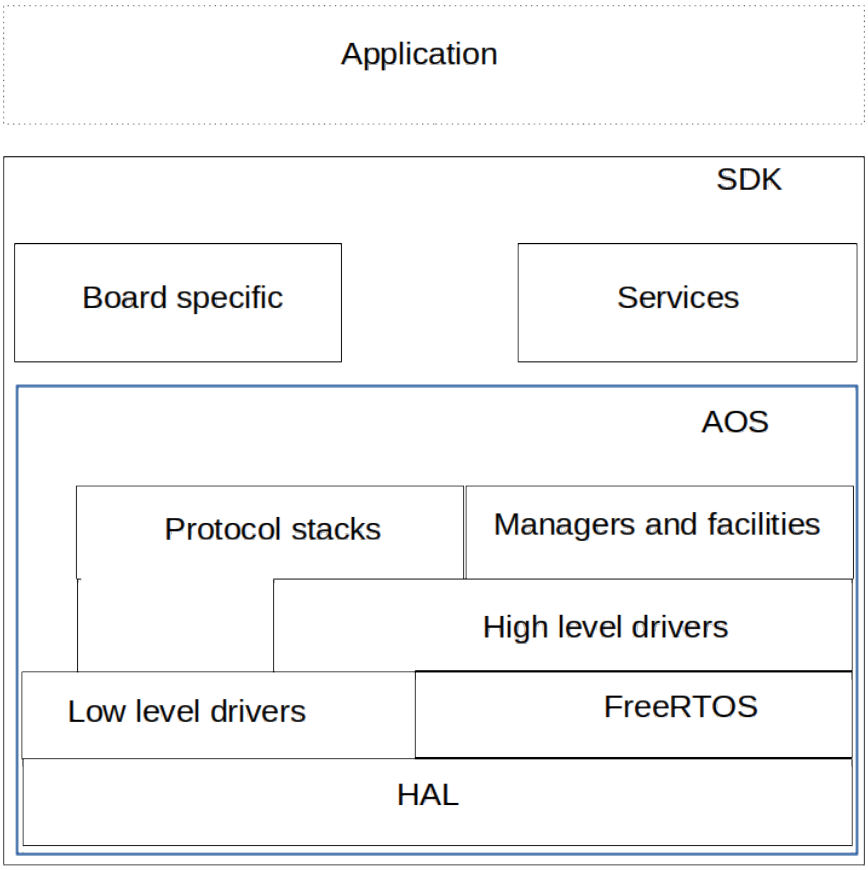
STM32CubeIDE v1.15.1 installation
Download the tool from from https://www.st.com/. Select the version 1.15.1. Previous versions may generate compilation or linker errors.

To create your working environment, follow the steps:
1. Get the official SDK version tarball and extract it. This create a directory **aos-sdk_vx.y-z**,which contains:
    - A manuals directory containing the documentation
    - The **AOS-SDK-public** directory containing the SDK and include files.
2. Create a working directory. Example: **abw-module-user**
3. Under the working directory, copy the **AOS-SDK-public** directory.
4. Under the working directory, clone the **abw-app-demo** or extract the demo tarball.
5. Start the IDE and create a new workspace pointing to **abw-module-user**.
6. Finally, from the IDE import the project **abw-app-demo**.
Now, the environment is ready.

Note that the **manuals/html** contains the API description. To read it it, double click on the file index.html. This will open your favorite WEB browser with the documentation.

2.3 General architecture

The overall architecture can be pictured as follow.



2.4 Architecture parts

2.4.1 HAL

The Hardware Abstraction Layer (HAL), provides basic functionalities to access the different functionalities of the main MCU and other components.
The HAL low layer (HAL-LL)of the main MCU is provided by ST Microelectronic. Note that some parts of the HAL high layer are also provided by the MCU manufacturer, while some others are provided by Abeeway.

The LR1110 HAL is provided By the component manufacturer (Semtech).

Note
As far as it's unusual, the application can access directly the HAL. In this case, care should be taken.

2.4.2 Low level drivers

These drivers rely on the HAL and provide basic functionalities such as the ADC, PWM, SPI, I2C, LP-timer and so on. These drivers do not use FreeRTOS.

2.4.3 FreeRTOS

FreeRTOS is a free Real Time Operating system. It offers a preemptive task scheduler, mutexes, semaphores, scheduled queues and software timers. While the dynamic memory manager is turned on, it is not used by AOS. Only static memory allocations are actually used. This comes from the following facts:

- The memory deallocation method used by FreeRTOS may lead into hole in memory.
- The amount of the actual memory required is not deterministic (not done at the compile time).

Note
It is not advised to use the dynamic memory management.

2.4.4 High level drivers

High level drivers are build on top of the low layer drivers. They usually offer more features than a simple driver. The aos_cli or aos_provisioning are good examples.

2.4.5 Managers and facilities

The managers (ie aos_lr1110_mgr) are used to sequence multiple services using a common resource, while the facilities (aos_error, aos_log, …) simplify the application development.

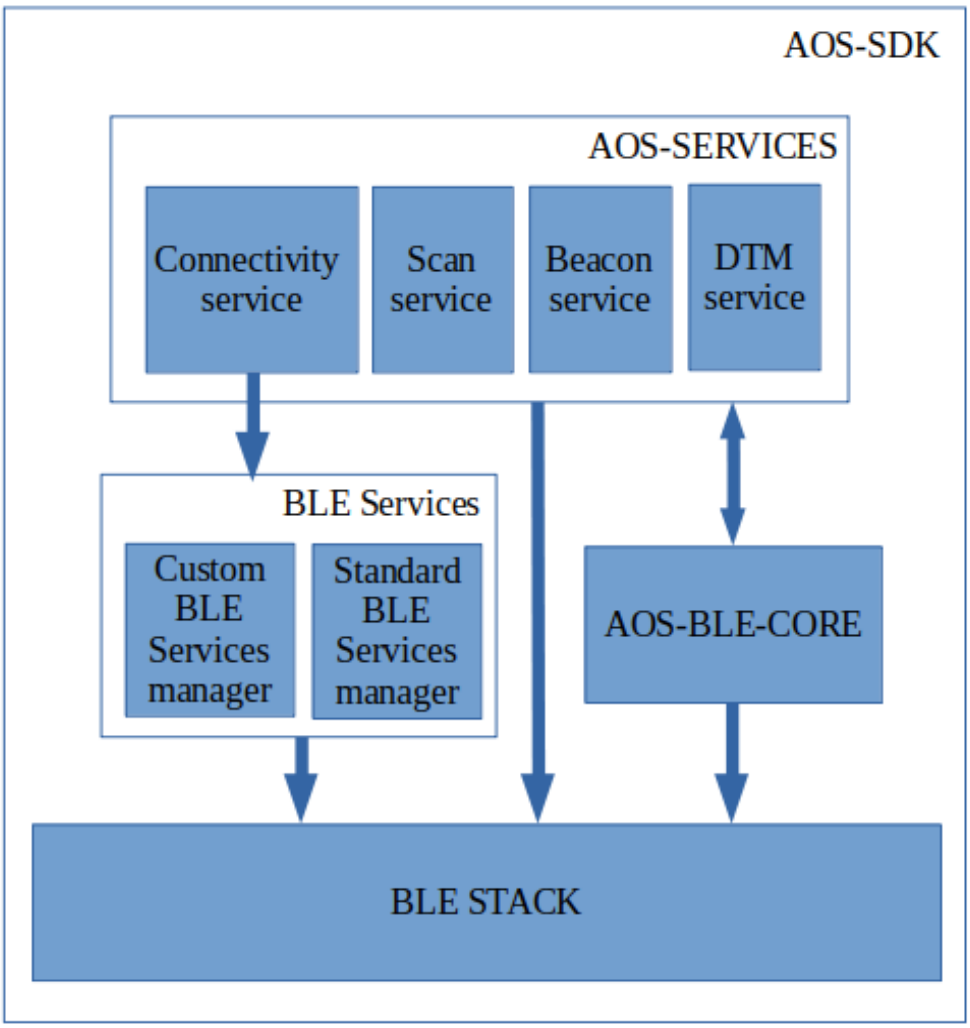
2.4.6 Services

These software entities provide enhanced services to the application.

2.4.7 Board specific

AOS provide a collection of component drivers that are not directly included in the Abeeway Module. However, such components can be installed on board around the module.

2.4.8 BLE architecture



The BLE in AOS-SDK has four major components:

- **BLE Stack**: BLE HCI/ACI APIs.
- **BLE Services**: APIs to initialize and set standard BLE services and custom BLE services:
  - **Custom BLE Services Manager**: Offers APIs to create new services and characteristics, and initialize characteristic data. The user can create a maximum of 10 custom BLE services and a maximum of 10 characteristics for each service.
  - **Standard BLE Services Manager**: Manages the initialization and characteristic updates of various BLE standard services (TX Power Service, Device Information Service, Immediate Alert Service, Link Loss Service, Battery Service, Environmental Sensing Service). For some characteristics, when a read operation is performed, this component

will use the application callback saved during initialization to request data from the application layer.

- **AOS-BLE-CORE**: Responsible for managing the BLE core initialization and dispatching BLE notifications to the connectivity service, scan service, and the main application.
- **AOS-SERVICES**: This component offers several services:
  - **Connectivity Service**: Manages BLE connectivity and exposes APIs related to BLE connectivity.
  - **Scan Service**: Manages beacon scanning.
  - **Beacon Service**: Manages beacon advertising (different types of beacons managed include Eddystone UUID, AltBeacon, iBeacon, Quuppa, and Exposure).
  - **DTM Service**: Manages the Direct Test Mode service (test modes include tone, BLE TX, and BLE RX).

2.4.9 Conclusion

Generally, a simple application only uses:

- The facilities
- The services
- The board specific drivers (if any).

2.5 Memory layout

The memory (RAM and FLASH) layout is provided by the linker script. So it's up to the application developer to ensure that the layout is correct.

The usual flash memory map is:

```
Base address
08000000          20k   Abeeway bootloader
08005000          4k    Application parameter
08006000          696k  Application flash
080B4000          12k Reserved in case the BLE code size increases
080c7000          148k BLE full stack (fixed address for BLE V1.15.0)
080ec000          80k BLE FUS (fixed address for FUS V1.2.0)
```

Notes

- In absence of the bootloader, the associated memory can be used by the application.
- In absence of parameter, the dedicated section can be used by the application.
- It is discouraged to change the mapping of the BLE stack.

The usual RAM memory layout is:

```
Base address
20000004          192k   RAM application and SDK
20030000          10k    Shared RAM (M0/M4)
```

2.6 Applications

The package offers some demo applications showing the usage of some SDK parts.

2.6.1 Demo applications

There are 3 demo application, implemented as a tutorial:

- The demo1 is the simplest one, which introduces the SDK to a new user.
- The demo2 mainly focuses on the board drivers.
- The demo3 introduces simple and enhanced services.

Refer to the README of each demo application for more details.

2.6.2 Create your own application

To quickly start, it is recommended to use the STM32CubeIDE and the demo applications. Then choose the application that better fit your need and clone it.

3 Bootloader and independent watchdog

3.1 Bootloader

3.1.1 Overview

Along to the SDK, Abeeway provides a simple bootloader, which allows firmware update using the USB port.

Note that the use of the bootloader is not mandatory. The initialization part of the SDK relies on the address of the __isr_vector__ variable provided by the linker script.If the bootloader is expected, then the linker script should map the .text section of your application at the offset 0x6000.

Please refer to the section related to the linker script.

3.1.2 Behavior

The bootloader is always called at the boot time. It determines whether it should stay in bootloader or it should jump to the main application.

The criteria to stay in bootloader mode are:

- The content at the address 0x6000 is not a valid RAM address (stack pointer)
- The content at the address 0x6004 is not a valid FLASH address (boot vector)
- The RTC backup register #2 (starting from #0) is not null.

Note
When the main application wishes to jump to the bootloader, it should write the RTC backup register to a non null value then reset.

Example

```
// Set the bootloader mode
aos_rtc_backup_write(aos_rtc_backup_register_bootloader,
      aos_rtc_bootloader_rtc_cmd_enter);
NVIC_SystemReset();
```

Note
The bootloader mode is limited to 1 minute without any input. Once this delay exceeded, the bootloader clears the RTC backup register and reset the device. Note that once the xModem transfer starts this delay is no more taken into account and the timer is used to monitor the xModem transfer.

3.1.3 Commands

The bootloader includes the following commands (character cases must be respected):

- ABWu: Upload a new binary using xModem transfers. The binary will be written at the address 0x08006000.
- ABWe: Erase the application parameter located at the address 0x08005000 (single FLASH page of 4kB).
- v: Display the bootloader version
- r: Clear the RTC backup register #2 and reset the device. This will launch the new binary.
- ?: Display the help

3.1.4 Versions

It exists 3 versions of the bootloader:

- v1: Does not initialize and start the hardware watchdog
- v2: Initialize and start the hardware watchdog. It configure the independent watchdog to the max delay (around 30 seconds).
- v3: Pay attention to the Flash option byte and reconfigure the watchdog period to 30 seconds.

3.2 Independent watchdog

AOS supports the independent watchdog referred as IWDG . It can be either used or not and can be either hardware of software.

3.2.1 Hardware versus software

The configuration of the independent watchdog can be either hardware of software.
When configured as hardware, the watchdog starts with a hardware configuration once the chip boots and does not need the help from the firmware to start. This usage is particularly convenient when the board is powered by worn out battery and super capacitors: during the charge of capacitor if a component needs a high peak current, the voltage may strongly decrease and generates a lock of the main CPU.

When configured as software, the independent watchdog starts only if the firmware actually starts it.

This option should be configured via the flash memory option register of the chip and should be done during the manufacturing process of your board. Note that this option byte is accessible via the ST programmer tool.
This register contains several flags. The relevant ones are:

- **IWDG_STDBY**: Must be reset (independent watchdog frozen in standby mode).
- **IWDG_STOP**: Must be reset (independent watchdog frozen in standby mode).
- **IWDG_SW**: Should be set if the the software mode is chosen and reset for the hardware mode.

Note
The default values of the module are IWDG_STDBY and IWDG_STOP reset and IWDG_SW set.

3.2.2 Usage

Depending on the watchdog mode choice (hardware versus software), the appropriate bootloader version (if used) should be chosen. The safest one is the version 3 since it automatically detect whether the watchdog is started.

If you don't want to use the watchdog, then let the flash memory option register to its default configuration and calls the function **aos_system_init** with false. This will not start the watchdog.

4 AOS facilities

AOS provides several useful facilities and drivers.

4.1 Tracing and logs

4.1.1 Overview

The tracing and logs facility is accessible via aos_log.h. It should be initialized by the application before being used.
SDK modules are already registered against the tracing facility and one entry is reserved for the application.
To select the verbosity of the trace, a level is available per module:

- Disabled: No messages are traced.
- Warning: Only the warning messages are traced
- Status: Status messages and warnings are traced.
- Debug: All messages are traced.

At the initialization stage, a tracing output function should be provided. This function is application specific and can redirect the trace output where you want. If you wish to redirect the traces to the CLI, then you should use pass the **cli_log** function as the redirecting trace function.

4.1.2 Usage

The tracing facility should be initialized early in the **main** function, usually just after the system initialization.

A trace message consists of an optional timestamp followed by the abbreviated module name and finally the trace content.

Several API functions are available and described in the API documentation. Relevant functions are:

- **aos_log_msg**: Generic function to log a message (module and level are provided). It also accept a parameter indicating whether a timestamp should be appended to the trace. If the trace is complete the trace should ends with the line feed character ('\n').
- **aos_log_msg_feed**: Allow to complete a previous message started with **aos_log_msg**. In that case the timestamp and the module name are not set.
- **aos_log_status**: Wrapper to log a status message.
- **aos_log_warning**: Wrapper to log a warning message.
- **`aos_log_dump_hex`**: Output an hexadecimal buffer.

4.2 Low power manager

4.2.1 Overview

AOS is a low power operating system. The low power part is fully managed by the SDK. However, under certain condition it is possible that the application requires a certain level of stop mode of the main processor.

The STM32WB55 processor accepts the multiple power saving modes. However, AOS supports only the following ones:

- run: The processor runs at the full speed (64 MHz) and does not sleep. Consumption: 10.2 mA
- sleep: Sleep mode. Consumption: 5.1 mA
- stop1: Medium low power mode. Consumption: 9.2 uA
- stop2: Lowest power mode supported by AOS. Consumption: 5.0 uA

Note that some peripherals do not support deep sleep modes.

As you can see in the API description, a lot of low power entities requires an access to the low power manager (Refer to `aos_lpm_requester_t` enumerated). Most of these entities are under the AOS responsibility. However the `aos_lpm_requester_application` entry is reserved for the application.

4.2.2 Usage

AOS initializes itself the Low power manager. In the case where the application requires a control to the low power mode of the CPU (application driver for example), the application should call the function `aos_lpm_set_mode` with the requester being `aos_lpm_requester_application`.

When calling this function, the parameter `mode` reflects the maximum low power mode supported. For example, the mode can be set to `aos_lpm_mode_no_sleep` or `aos_lpm_mode_stop1` if at a time the application cannot support the stop2 mode and set back to `aos_lpm_mode_stop2` once the stop2 mode can be restored.

Note that a specific callback function may be provided along to the `aos_lpm_set_mode` call. This callback function is used to deeper control the low power mode. It will be called each time the low power mode is entered or left. The callback should return **false** whenever it expect the CPU to not sleep and true otherwise.

4.3 Provisioning

4.3.1 Overview

The provisioning is a permanent storage where invariable parameters can be stored. This storage is hosted by the LR1110 component and is usually used to place information related to the manufacturing.
This permanent storage should not be written to much time to avoid damaging the component (usual limitation is around 10,000 writes.

Note that the LoRa information such as the device unique identifier, key and so on are stored there. These values are setup by the module manufacturer.

For a module user, it can be however convenient to place in this storage area additional parameters belonging to the hardware of the board.

4.3.2 Usage

This facility does not need to be initialized. As stated above, the storage area contains critical information of the device. So, it is strongly discouraged to erase the area (will lost all critical information) or to erase the keys.
Note that the provisioning information is also cached in the RAM memory.

For a module user, the API function that could be used are:

- `aos_provisioning_read`: To read the whole provisioning.
- `aos_provisioning_save`: To save the whole provisioning in the LR1110 flash
- All `aos_provisioning_get_xxx` functions are safe to use.
- `aos_provisioning_set_parameter`: Set a user parameter.

Note that if you want to store your own parameters, the is the function to use. Once called, you should call the function `aos_provisioning_save` to permanently store your values.

4.4 Error handling

4.4.1 Overview

AOS includes an error handling, which catches unexpected critical hardware or software errors. Once such errors have been processed, the system automatically reset.

The error handling stores the last error in a `.noinit` section, which is preserved across reset. Refer to the linker script section to know how to place this section.

4.4.2 Usage

This facility does not need to be initialized. Hardware (bus error, illegal instruction, …) errors as well as unexpected exceptions are already registered against the error handler.

Based on the error code, either the MCU internal register values or file/code line are provided.

All errors prefixed by `AOS_ERROR_HW` have the MCU internal register information, while the ones prefixed with `AOS_ERROR_SW` have the file name and code line information. There are two exceptions at this rule:

- The hardware watchdog `AOS_ERROR_HW_WDOG`. All MCU internal registers are set to a null value. This is due to the nature of the independent watchdog (no related exception watchdog).
- The Brown-out `AOS_ERROR_HW_BOR`. All MCU internal registers are set to a null value.

Once a reset occurs, it is convenient to retrieve the cause. This is done via the function `aos_error_get`, which will provide the whole information about the reset.

The error storage area can be cleared by using the function `aos_error_clear`.

Finally, the module user can trigger its own error (requiring a reset), by using the function `aos_error_trigger`.

4.5 Fixed point mathematics

4.5.1 Overview

The SDK embeds a fixed point library, called fix16. This library, located under the aos-boards directory provides 16Q16 fixed points arithmetic, which is a good alternative to the floating numbers. Using this technique for decimal value increase the speed of the calculation and is less memory consuming.

4.5.2 Usage

The API library is accessible via the fix16.h file.

5 Low level drivers

This section describes only the low level drivers that can be used by boards integrating the Abeeway geolocation module.

5.1 GPIO

5.1.1 Overview

The general purpose input/output driver provides the access functions to the input/output pins. As its name indicates, it configures the I/O, provide accesses (read/write) and catches associated interruptions (if any). A general purpose I/O can be configured as:

- Digital input with or without interruptions
- Digital output
- Analog input. The ADC driver will be also used in this case.
- Analog output. Not supported by the driver. Refer to the PWM for such a mode.

The driver API is accessible via the file aos_gpio.h.

Note that the driver is also able to setup a voltage bridge used to monitor the battery level.

5.1.2 Usage

The GPIO are identified by the enumerated `aos_gpio_id_t`. It should be noted that some dedicated GPIOs can be used as general purpose if their alternate function (function for which they have been defined) is not used. They are:

- `aos_gpio_id_vbat_sense`
- `aos_gpio_id_lpuart_rx`
- `aos_gpio_id_lpuart_tx`
- `aos_gpio_id_lpuart_rts`
- `aos_gpio_id_lpuart_cts`
- `aos_gpio_id_spi_cs`
- `aos_gpio_id_user_adc1`
- `aos_gpio_id_i2c_int1`
- `aos_gpio_id_i2c_int2`
- `aos_gpio_id_pwm_ctrl`

Note that all pins supports the analog input mode..

Before being used, the GPIO driver must be initialized by the function `aos_gpio_init`.

A given GPIO must be opened before being used and close if it is no more used. The manager also offers the possibility of reserving a GPIO. In this case, it means that the GPIO cannot be for another purpose and will be configured later. For example, when opening the LPUART, the RX/TX will be reserved as

alternate functions (RTS/CTS may also be reserved if the flow control is enabled). Once the LPUART is closed, the reserved pins are released and becomes available.

Opening/Closing
There is two API functions to open a GPIO:

- `aos_gpio_open`: Simple function to open a GPIO. Only the mode is provided. In such a case, a digital input cannot trigger an exception. A digital output opened with this function will have the pull mode set to push-pull and the output type set to `aos_gpio_pull_type_none`.
- `aos_gpio_open_ext`: Extended function to open a GPIO. All parameters are available.
- `aos_gpio_close`: Close a GPIO.

Note that you cannot configure a GPIO as alternate function using these functions. GPIOs used as alternate functions should be reserved.

Digital input exception servicing
The driver offers two different modes to service the exception:

- `aos_gpio_irq_service_type_int`: The user callback function is called under the IRQ exception. It is not advised to use this mode unless you have a strong real time constraint. In this case, the callback function must have the type `aos_gpio_isr_callback_t`.
- `aos_gpio_irq_service_type_thread`: The user callback function is called under the system thread (no more in interrupt context). In this case, the callback function must have the type `aos_system_user_callback_t`.

Analog inputs
The driver uses the ADC for the GPIOs configured as analog input. As a consequence the `aos_gpio_read` function will return an unsigned integer instead of a boolean value.

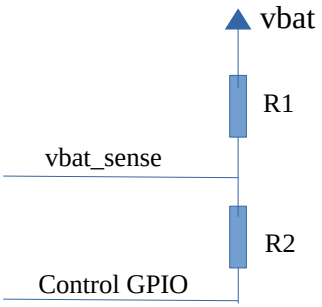
The GPIOs that can be used as analog inputs are:

- `aos_gpio_id_user_adc1`
- `aos_gpio_id_user_vbat_sense`
- `aos_gpio_id_i2c_int1`

Reserving/releasing GPIO
When a GPIO is used as alternate function, it should be reserved via the `aos_gpio_reserve` and released by the function `aos_gpio_unreserve`. These functions are particularly useful, if the LP-UART is used by a user implemented driver.

5.1.3 Battery level measurement

The GPIO driver provides a specific function to measure vbat through the `aos_gpio_id_vbat_sense` pin. It assumes that a voltage bridge is used and controlled by another GPIO.
The schematic looks like



$$vbat_{sense} = vbat * R1/(R1+R2)$$

If such an implementation is done, the `aos_gpio_bat_setup` and `aos_gpio_read_battery_voltage` functions can be used.
The setup function should be called at the application initialization stage and provides the following parameters:

- `gpio_vbat_ctrl`: GPIO used to switch on/off the voltage bridge. Digital output.
- `gpio_vbat_analog`: Analog GPIO which will perform the ADC measurement.
- `vbat_ctrl_ratio`: Voltage ratio of the bridge. R1/(R1+ R2)
- `vbat_ctrl_setup_delay`: Time requires for the voltage to stabilize before doing the measurement.
- `vbat_offset`: Voltage offset introduce by the bridge due to the current leak in the loop ground→ R2 → analog input.

The reading function will provide directly the actual vbat value in mV.

Note that to be able to measure vbat on the Abeeway module EVK, you should connect a battery (or any power supply) to the **BAT** pin.
In the case of an external supply and to avoid electrical conflicts, the jumper **J1** should connect the system power source to USB.

5.2 ADC

This driver should not be used as-is. If a GPIO needs to be configured as analog input, the GPIO driver should be used.

Note that the accesses to the ADC should be serialized. The ADC ensures this aspect by using a FreeRTOS semaphore. The calling thread will be locked until the measurement has been done.

The ADC is used to read the MCU temperature and may be used to measure the vbat voltage. Note that the MCU temperature measure uses a specific ADC internal channel.

The ADC calibrates and uses the voltage reference (VREF = 3.3v) for measurements. The calibration takes place each time an analog pin is read.

The value provided by the ADC API is converted as follow: value = (measure * VREF)/4095, where measure is the actual ADC value and can range in [0..4095].

5.3 PWM

The Pulse Width Modulation driver is usually used to drive a buzzer or to control the speed of a motor.
It uses the timer16 hardware resource configured as PWM.

The driver does not need to be initialized. To start the PWM, the function `aos_pwm_start` should be called with the expected wave frequency and duty-cycle. To stop it, just call the function `aos_pwm_stop`.

Notes

- The driver access is not protected by a semaphore.
- The driver is not blocking. This means that the start function returns immediately after starting the PWM.
- While running, the PWM driver enforces the MCU running state (MCU sleep/stop modes deactivated.
- It is possible to call multiple times the start function without calling the stop function. This allows changing the frequency or the duty-cycle without interruptions.

5.4 LPUART & USART

5.4.1 LPUART overview

The LPUART driver supports only the asynchronous receiver/transmitter mode.
It supports:

- MCU stop2 mode if the selected baudrate is less than or equal to 57600.
- MCU stop1 mode if the selected baudrate is greater than 57600.
- Max baudrate: 115200.
- Hardware flow control (configurable).
- All parity modes (odd, even, none)
- All stop bit selections (0.5, 1, 1.5, 2)

- All data sizes (7,8,9)
- Transmission (RX/TX) done under interruptions

Note
On the Abeeway module EVK, the LPUART is connected to the debug part of the board, which is accessible via the USB port.

5.4.2 USART overview

The USART driver supports only the asynchronous receiver/transmitter mode. Unlike the LPUART, It support only the MCU stop1 mode.

The configuration is the same than the LPUART except the baudrate which can be greater.

Note
Usually, the USART is used to access the internal GNSS (MT3333) component.

5.4.3 Usage

The LPUART and USART drivers are exported via the variable `aos_uart_driver`. The associated structure contains:

- An open function needing a configuration parameter. This parameter handles the UART configuration.
- A close function.
- A read function to get bytes received.
- A write function to transmit bytes
- An IO control function to perform action. The UART does not need to be opened to send IO controls.

UART selection
The UART driver exports a single access structure. Each access function has an UART identifier selecting the appropriate UART.

Configuration
The configuration is done during the opening stage and cannot be changed on the fly. It is passed via the structure `aos_uart_config_t` containing the following fields:

- `speed`: UART baudrate. The associated enumerated should be used.
- `stop`: Number of stop bits. The associated enumerated should be used.
- `parity`: Parity selection. The associated enumerated should be used.
- `data_format`: Format of the data. The associated enumerated should be used.
- `hard_flow_control`: Hardware flow control selection. The associated enumerated should be

used. Note that the USART does not support the hardware flow control.
- `tx_buffer_size`: Size of the transmit buffer;
- `rx_buffer_size`: Size of the receive buffer
- `tx_buffer`: Transmit buffer. Byte array with a size of t`x_buffer_size`.
- `rx_buffer`: Receive buffer. Byte array with a size of `rx_buffer_size`.
- `user_rx_cb`: User callback called upon characters reception
- `user_arg`: User argument passed along the callback. Opaque for the driver.

The UART buffers are provided by the user. This let free the user to decide the amount of memory dedicated to the UART. The buffers must be not shared between UARTs.
The buffers are managed by the driver as independent FIFO. The user does not need to pay attention to the FIFO management.

The user callback function is called whenever the reception FIFO is not empty. Note that the driver uses the AOS system thread to call the user callback. This means that it is called under thread context and not directly under interruption. However, processing time under the user callback should remain limited since the system thread is also used for other purpose.

In the case where the processing time cannot be short, it is advised to create another thread and to defer the processing to it.

5.5 I2C

5.5.1 Overview

The I2C driver manages the two I2C serial buses. One is internal (I2C3) and its usage is restricted to AOS. The second (I2C1) is said external and is free for use.

The driver acts as the I2C master on both buses and transfers are done under interruptions (no DMA).
Since multiple devices can reside on the same I2C bus, the access is protected by a mutex. Each buses have their own mutex.

During data transfers, the I2C is fully locked. Once the transfer ends up, the I2C driver is unlocked. This means that the calling thread is locked until the transfer completes.

5.5.2 Usage

The driver is exported as a structure containing the following access functions:
- `open`: Open the driver.
- `close`: Close the driver
- `read`: Read I2C device registers
- `write`: Write I2C device registers
- `ioctl`: IO control

The I2C configuration is static and cannot be modified:

- Addressing mode: 7 bits or 16 bits (configurable via IO control)
- Dual addressing mode: disabled
- Transfer timeout: 500ms (configurable via IO control)
- Fast mode (400 kHz)

5.6 SPI

TO BE DONE

6 AOS services

Most of time the services relies on underlying drivers. They provides enhanced features which simplifies the user application code.

6.1 Configuration service

6.1.1 Overview

This service offers a configuration management, which resides in a permanent storage. A dedicated FLASH page (4Kb) is reserved for this service.

The configuration consists of several user parameters that can be dynamically changed. The service supports the following parameter format:

- int: Integer 32 bits (signed)
- float: Floating point simple precision
- string: ASCII string null terminated (max 32 characters including the NULL).
- byte-array: Array of bytes (max 32 bytes)

Note that a special type called **deprecated** is also available. It indicates that the parameter is no more used but remains in the configuration for backward compatibility.

The maximum number of configuration parameters is 300. The storage are for byte-array and string has a size of 1664. Such variables are stored on 32 bytes regardless of their actual size. This leads into a maximum of 52 string/byte-array variables.

The storage area contains a header handling:

- A magic number, indicating whether the parameter page is valid.
- A CRC 32 bits done over all variables
- The number of variables
- A version number, which can be used for backward compatibility.

The parameters are accessible via its unique identifier (integer value on 16 bits).

In order to optimize the flash accesses (particularly the write action), the service includes a cache located in RAM. The cache is written in flash only on user demand.

6.1.2 Usage

6.1.2.1 Initialization and formatting

Before being used, the server must be initialized using the function `srv_config_init`. Note that this function requires a parameter indicating the actual address in flash of the page that will be used by the service. Usually this parameter can be extracted from the linker script.

Once initialized, it is recommended to call the function `srv_config_get_info` to know whether the configuration page is properly formatted. If it is not the case, it is advised to call the function `srv_config_format_and_init`. This will format the flash and write the parameters with their default values. You may also check the parameter version contained in the flash against what your code expect and in case of incompatibility take the appropriate decision. The formatting forces both the RAM cache and the flash to be synchronized with the initial values. So, there is no need to save the configuration.

While formatting, you should provide the list of parameter descriptors. These descriptors are typed `srv_config_param_descriptor_t` and contains the following fields:

- `descriptor.identifier`: Unique identifier on 16 bits
- `descriptor.type`: Type of the variable (refer to `srv_config_param_type_t`)
- `descriptor.length`; Length of the byte-array value.
- `value`: Parameter value (refer to `srv_config_param_value_t`)

The parameter descriptor list consists of an array of `srv_config_param_descriptor_t` structure.

Note that the value is simply a 32 bits, which is typed as a pointer for string and byte-array. For such variables, the value resides in another location for which the pointer just references it. **Care must be taken when reading or writing such a value since the pointer is not an array of bytes**.

Note that the type and the identifier of an existing parameter cannot be modified. If such a modification is expected then you have to reinitialize the manager (which will force a flash page erasing and a rewriting).

6.1.2.2 Creating/deleting a parameter

The service allows dynamic parameter creation and deletion. This means that after formatting, it is possible to add or remove parameters.
Removing a parameter is done via the function `srv_config_param_new`, which requires the parameter identifier. The service will not actually erase the parameter. Instead, it will set its type to **deprecated**.

Adding a new parameter is done via the function `srv_config_param_new`. It requires the filling of a descriptor structure, for which the identifier must not be already used by another parameter.

6.1.2.3 Writing a parameter value

A configuration parameter can be written using the `srv_config_param_set`. Note that the parameter descriptor should be provided.

Note that writing a parameter does not update the flash. The configuration save should be called for this update.
To preserve the flash (limited number of erase/write operation), It is recommended to write as much as possible the parameters then save the configuration.

6.1.2.4 Reading a parameter value

A configuration parameter can be read using the `srv_config_param_get`. Note that a handle on the parameter descriptor should be provided. It will be filled with the the parameter descriptor that resides in the configuration RAM cache. For this reason it is discouraged to modify the descriptor or the value.

6.1.2.5 Saving the configuration

After a parameter modification, it is usual to save the configuration in the flash. It is done via the function `srv_config_save`. Through this function, you may change the configuration version.
Note the the `keep_deprecated` calling parameter indicates whether the deprecated parameters should be kept. If not, while saving the parameters the deprecated ones are omitted.

6.1.2.6 Linker script

To use this service the linker script should declare a specific FLASH page (size: 4 kB and aligned on 4 kB). At the top of the reserved page, the exported variable `__user_param_flash_base_addr` should be defined. An example of a such declaration could be:

```
MEMORY
{
…
PARAM (rx) : ORIGIN = xxxx, LENGTH = 4k // Where xxx is the physical location
…
}

SECTIONS
{
…
 .user_param :
  {
      PROVIDE(__user_param_flash_base_addr = .);
      PROVIDE(__user_param_flash_size = LENGTH(PARAM));
  } >PARAM
…

}
```

6.2 MT3333 GNSS service

6.2.1 Overview

This service manages the actual GNSS device (MT3333). This device can work in either assisted or non-assisted mode. We remind that the assisted mode requires a position solver residing in a dedicated machine located on the network side. The device in non-assisted mode performs itself the position resolutions.

The API is accessible via the srv-gnss-mt3333.h, while most of GNSS definitions are located in the aos-core/aos-gnss-common.h file.

Notes

- The service also offers accesses to the almanac of the GPS and BEIDOU constellations.
- The service runs it its own thread.
- The interface between the service and the application is done through callbacks. There are two different callbacks:
  - Main: This user callback is used to pass most of information between the service and the application.
  - Request: This user callback is used whenever an application request is done.

  Note that the request callback is not required. In that case the user request results are serviced via the main callback.

6.2.2 Usage

6.2.2.1 Initialization and start

Before being used, the service must be initialized via the function `srv_gnss_mt3333_init`.
Once initialized, the service can be instructed to start the position acquisition via the function `srv_gnss_mt3333_start`. Along to this function, the application must provide the main callback function as well as the settings that the GNSS must use.

The configuration is defined by the structure `srv_gnss_mt3333_configuration_t`, where:

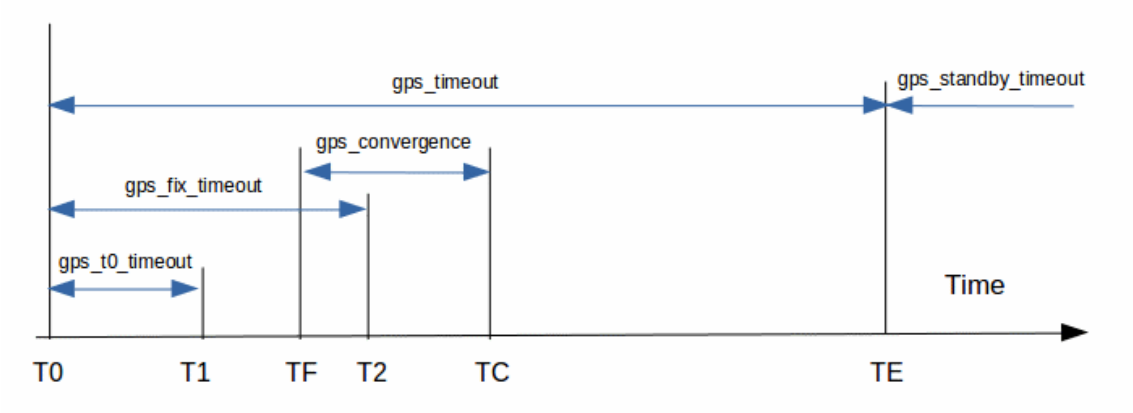
- `mode`. Acquisition mode (assisted vs non-assisted).
- constellations: GNSS constellations that the device can use to acquire a position. Note that less constellations imply less energy consumption and less accuracy in the position.
- `fix_acq_timeout`: Max time in seconds to make a GNSS fix. Once this delay expires, the result is provided to the user via the callback.
- `agnss_acq_timeout`: Max time in seconds to make a GNSS fix or a valid acquisition in the assisted mode. Once this delay expires, the result is provided to the user via the callback.
- `t0_timeout`: Max time in seconds to see at least one satellite. If this timeout is enabled (non null value), the service checks that after the timeout one satellite at least is in view. If this is the case, the service will let the device opens until a valid fix is done or the xxx_acq_timeout expires.
  This timer is usually used to limit the GNSS power-on time in poor conditions (indoor for example) where a fix cannot be done.
- `t1_timeout`: Max time in seconds to acquire a first fix. A null value extend this duration to

`fix_acq_timeout`

- `ehpe`: Estimated horizontal position error in meters that is acceptable for the fix.
- `convergence_timeout`: Max time in seconds from the first fix obtained (during this acquisition) to converge to the given EHPE.
- `standby_timeout`: Duration in seconds for which the GNSS device stays in standby mode after an acquisition. Once elapsed, the GNSS device is powered off. A null value disables the standby mode. Note that while in standby mode, the device (in low power mode) keeps in memory all information to make a next fix faster. In the case where fixes should be done at a frequency below than one each 2 hours, It is recommended to use a non null value to avoid powering off the device.
- `prn_cfg`: Specific configuration for the assisted mode.
- `update_systime_on_fix`: When this variable is set to TRUE, the AOS system time is automatically updated when a fix is complete. We remind that AOS uses the UTC time.
- `hold_gnss_on`: When this variable is set to TRUE, the service closes its driver but keep the GNSS power to ON. This mode is consuming but allows a a very short Time To First Fix and a good position accuracy.
- `local_info`: Local information related to the position. The main processor may provide its position if it knows it or the previous position If it is static for example. When provided, the device will reduce its TTFF since it can use its own almanacs.

6.2.2.2 Timings

The configurable timings can be pictured as follow



Where:

- T0: Start the GNSS geolocation.
- T1: No satellite seen in the sky.
- T2: No GNSS fix.
- TF: GNSS first fix acquired.
- TC: Convergence time to refine the GNSS position.
- TE: Termination of the active GNSS state. The GNSS transitions back to standby mode.

6.2.2.3 Queries

The service supports user queries. A query can be sent at any time: If a position acquisition is in progress, the query is serviced in parallel of the acquisition. Otherwise, the driver opens the GNSS device and closes once the query (or the query set) has been done.
A specific user callback should be provided along to the request type. The callback function should return TRUE if another request will be performed and FALSE otherwise. Note that upon the return value of the callback, the driver will close or not the device. Along to the callback, the request status as well as the related information is provided.

The supported queries are:

- get_version: Retrieve the GNSS firmware version.
- get_almanac_gps: Retrieve a GPS almanac entry
- set_almanac_gps: Write a a GPS almanac entry. Note that the entries are locally flashed in the device only in certain circumstance.
- get_almanac_beidou: Retrieve a BEIDOU almanac entry.

Notes

- The actual answer data is located in the driver. It is accessible until another query is done.
- Each query type has its own API function.
- It is not possible to send multiple query at a time. The callback should be triggered before performing another request. Under the callback it is possible to request another query.

6.2.2.4 Retrieving data

The GNSS driver keeps data until it is started for another position. The data can be:

- The fix information
- The tracking information: Satellites tracked as well as their C/N.
- The pseudo-range in case of assisted positions.

Note
An API function is available for each type of information

6.2.2.5 Stopping vs aborting

At the end of the acquisition, the GNSS stops by itself if the convergence criteria is met. However, in some cases, the user may want to abort the current position. In this case the `srv_gnss_mt3333_abort_acquisition` function should be called. Unlike the stop function, which shutdowns the device, the abort function will put the device in standby mode if configured for and in power off mode if the the standby_duration parameter is null.

As said above, the `srv_gnss_mt3333_stop` function puts the device in power off mode regardless the configuration.

6.2.2.6 Helpers

The service offers some helper functions to get and clear statistics, get and clear GNSS usage duration and to retrieve the power consumption.

The power consumption is based on the usage duration. So, to clear the power consumption, you have to clear the usage duration.

6.3 WIFI scan service

6.3.1 Overview

This service uses the LR1110 device with its WIFI sniffing capability. It scans WIFI devices around and reports the list of BSSID found.

It relies on the `aos_lr1110_mgr` and the LoRa Basic Modem stack to sequence the LR1110 requests. As a consequence a WIFI scan can be done at any time for a user point of view.

6.3.2 Usage

The AOS LR1110 manager must be initialized (`aos_lr1110_init`) first then the service via the function `srv_wifi_scan_init`.

The function `srv_wifi_scan_start` should be used to actually start the scan. The function requires a callback that will be triggered once the scan is done. The result is provided to the scan callback. Note that the service also maintains a copy of the scan, which can be retrieve later.
The configuration should be provided via the `settings` parameter. It is a structure containing the following fields:

- `channels_mask`: mask of the channels to be scanned. It is a bitfield representing the combination of channels that should be scanned.
  Refer to `aos_wifi_scan_channel_mask_type_t`.
- `types`: WIFI signal types to be scanned.
- `max_results`: Maximum number of BSSID you expect.
- `timeout_per_channel`: Max time in ms to spend scanning one channel. Range [1 .. 65535].
- `timeout_per_scan`: Max time in ms to spend in preamble detection for each single scan. Range [1.. 65535].

The configuration is as flexible as possible to deal between scan quality and power consumption. Be aware that the scan ends up once the max_result is reached. This means that you may not have the closest BSSID (highest RSSI). However, the service sorts by RSSI order the result.

To be sure to have the closest BSSID, it is advised to:

- Request the maximum number of BSSID (32)
- Use the channel-all for the channel_mask.
- Scan all types: B, G, N.
- timeout_per_channel should be set to 500 ms
- timeout_per_scan should be set to 130 ms.

To abort a WIFI scan in progress, use the function `srv_wifi_scan_stop`.

Refer to the LR1110 user manual provided by Semtech for more detail.

6.4 BLE scan service

6.4.1 Overview

This service uses the BLE stack provided by STMicroElectronic. The stack runs partly on the ARM M0+ embedded in the STM32WB55 and on the main core ARM cortex M4.

Note that the BLE stack requires a general initialization (for all services). This done via the function `aos_ble_core_app_init`.

Basically the service performs a beacon scan based on filters. Once the scan is done, a user callback is triggered with the result and a report.
Both the result and the reports contain the list of observed beacons matching the filtering criteria. The difference comes from the provided data:

- The result handles the information contained in the beacon advertisement part according to the beacon type.
- The report contains only information requested by the user. For example, the user can be interested by the beacon identifiers (carried inside the advertised data) or MAC addresses.

6.4.2 Usage

This service does not need to be initialized. However, before starting a scan, the service should be configured. This is achieved by the function `srv_ble_scan_get_params`, which returns the pointer to the configuration parameters. At this stage, the user should fill the parameters using the returned value then start the scan using the function `srv_ble_scan_start`.
Note that the start function accepts a callback function, which will be called once the scan is complete.

During the scan, the user should not manipulate the scan parameters.

The scan operation can be aborted at any time by calling the function `srv_ble_scan_stop`.

6.4.2.1 Filtering

The service uses an enhanced filtering scheme depending on the beacons type to filter. The beacon type, listed by the enumerated `srv_ble_scan_beacon_type_t`, can be:

- `srv_ble_scan_beacon_type_all`: All beacon types are accepted. The filters cannot be applied and the report can contain only MAC addresses.
- `srv_ble_scan_beacon_type_eddy_uid`: Only Eddystone UUID beacons are accepted. The filters can be applied and the report can contain any type of information (beacon identifiers or MAC addresses)
- `srv_ble_scan_beacon_type_eddy_url`: Only Eddystone URL beacons are accepted. The filters can be applied and the report can contain any type of information (beacon identifiers or MAC addresses)
- `srv_ble_scan_beacon_type_eddy_all`: All Eddystone beacons (UUID/URL/TLM/EID) are accepted. The filters cannot be applied and the report can contain only MAC addresses.
- `srv_ble_scan_beacon_type_ibeacon`: Only iBeacons are accepted. The filters can be

applied and the report can contain any type of information (beacon identifiers or MAC addresses)
- `srv_ble_scan_beacon_type_altbeacon`: Only altBeacons are accepted. The filters can be applied and the report can contain any type of information (beacon identifier or MAC addresses)
- `srv_ble_scan_beacon_type_custom`: The filters must be configured. Beacons matching the filter are accepted. The report can contain any type of information (beacon identifiers or MAC addresses)
- `srv_ble_scan_beacon_type_exposure`: Only exposure advertisements are accepted. The filters cannot be applied and the report can handle only MAC addresses.

The scan configuration contains two filters. Each filter contains a 10 bytes mask and value. It also defines a start offset from which the filter should applied.
The filtering principle is straight forward: The filter mask is applied (logical AND) on the advertised data and compared to the value.
If the filter matches (comparison success), the beacon is kept otherwise it is discarded.
The start offset is the offset starting from the beginning of the advertisement data part.

6.4.2.2 Result vs report

Once the scan is done, two structures are provided to the user callback. Both contain beacons entries stored after filtering:

- `srv_ble_scan_result_t`: Each entry contains the full beacon information. The entry format is based on the beacon type. The structure content is fixed and cannot be tuned.
- `srv_ble_scan_report_t`: Each entry contains the relevant part of the beacon information. It is configurable via the `srv_ble_scan_param_report_t` structure. You can decide about the number of entry you want and the type of identifier:
  - `srv_ble_scan_report_type_mac_address`: The entries contain the MAC address in the identifier field. The short form is used in this case.
  - `srv_ble_scan_report_type_short_id`: The identifier field of an entry contains 6 bytes starting from the configured start_id_offset.
  - `srv_ble_scan_report_type_long_id`: The identifier field of an entry contains 16 bytes starting from the configured start_id_offset. Such a type is usually configured to do BLE finger printing.

Note that the `start_id_offset` does not start from the beginning of the advertisement frame. Instead it is related to the type of beacons:

- Eddystone beacons: `start_id_offset` = 0 locates the data part of the advertisement frame.
- iBeacons: `start_id_offset` = 0 locates the manufacturing UUID field of the advertisement frame.
- altBeacons: `start_id_offset` = 0 locates the beacon ID field of the advertisement frame.
- exposure: `start_id_offset` = 0 locates the RPI field of the advertisement frame.
- custom: `start_id_offset` = 0 locates the beginning of the advertisement frame.

6.4.2.3 Scan configuration

The scan is configured using the structure `srv_ble_scan_param_t`. This structure should be retrieved and manipulated before starting a scan.

The overall scan duration is configured via the `scan_duration` parameter. At the end of this duration,

the user callback will be triggered.
The `scan_window` and `scan_interval` reflects the standard BLE parameters:

- The window defines the actual scan duration for a given channel.
- The interval defines the actual scan duration for a given channel and the delay to switch to the next channel. It is always greater than window.

The `repeat_delay` parameter defines the period of scans:

- If the user expects a single scan, this parameter should be set to 0.
- If the user expects periodic scans, this parameter should contains the period (delay between each scans).

The configuration structure contains the filters, the report parameters and the type of beacons that should be considered.

The field `rssi_threshold` provides an extra filtering based on the RSSI level. Beacons with a RSSI below this threshold are discarded.

Finally, the configuration allows a fine tuning of the advertised channels via the parameter `adv_compensation`. It may be used in the case where the BLE antenna has not the same gain on all advertisement channel frequencies.

6.5 LR1110 GNSS service

To be done

6.6 LoRa service

6.6.1 Overview

This service supports the LoRaWAN v1.0.4 class A, with its corresponding Regional parameters rp002-1-0-4.
Only the Over The Air Activation is supported (OTAA). The LoRa parameters (DevEUI, Network and Application keys) must be provisioned. Note that the initial provisioning is done by the manufacturer and the LoRa region is set to EU868 (Europe). The provisioning can be modified by the `aos_provisioning` facility.

The service relies on the `aos_lr1110_mgr` and the LoRa Basic Modem (LBM) stack to sequence the LR1110 requests.

6.6.2 Usage

Before being used, the service must be initialized using the function `srv_lora_init`. This function requires a user callback parameter, which will be triggered for all LoRaWAN events.

Joining the network
Once initialized, the LoRa network should be joined. This is achieved via the `srv_lora_join` function. This function requires an array of 16 bytes, which provides the list of the data rates allowed to be used for the join process. The way the array is filled also provides the distribution of the data rate.
When sending a join, the LBM randomly picks up a data rate from the list and uses it.
Examples:

- Use of a single dara rate (DR0): Fill the array with DR0 only.
- Use two data rates DR0 and DR1 with a distribution of 50%: fill the array by alternating DR0 and DR1.
- Use two data rates DR0 at 80% and DR1 at 20%: fill the array by setting 12 times DR0 and 4 times DR1.
- Use all data-rate at equal distribution: fill the array by setting 2 times each DR0 plus 3 DR of your choice.

Once joined, the user callback is triggered with the appropriate event (join failure or success).

Leaving the network
It is possible to leave the network by using the function `srv_lora_leave`. Once left, the LoRa communication are stopped and you have to join again the network to recover the connectivity.

Sending an Uplink
You have to remind that the LoRaWAN standard limits the packet size regrading the data rate used (IRefer to the Regional parameters rp002-1-0-4).
Uplinks are sent by the `srv_lora_tx` function. Along to it, you have to provide:

- `dr`: The expected data-rate at which the uplink should be sent.
- `flags`: Bit-field handling the control information:
  - `AOS_LR1110_LORA_FLAG_CONFIRMED` (bit 0): If set the uplink will be send in confirmed mode.

    - AOS_LR1110_LORA_FLAG_ALLOW_UP_DR (bit 1): This flag is used when the packet size is not supported by the given data rate. If set, the manager will automatically adapt the provided data rate to an acceptable one able to pass the packet size. If reset and the packet size is too large for the provided data rate, the function return an error and the packet is not sent.
- port: Uplink port number.
- data_len: Length of the data.
- data: Pointer to the data to be sent.

Once the transmission is complete, the user callback is triggered with the TX success/failure/request failure event.
Note that the stack supports only one transmission at a time. The application should wait for the TX xxx event before sending another uplink. Also remind that due to the network regulation, certain LoRa region are submitted to a duty-cycle, which limits the transmission. So, an uplink may stay up to 1 hour in the LBM before being actually sent.

Receiving a downlink

Based on LoRaWAN standards, class A reception can occurs just after an uplink. When a downlink messages is received, the user callback is called with the event:

- srv_lora_user_event_rx: The downlink data and information are pointed by the rx_data variable. The network has no more downlinks to send.
- srv_lora_user_event_rx_pending: he downlink data and information are pointed by the rx_data variable. The network has more downlinks to send (Frame pending bit set).

When a reception occurs Frame with a frame pending bit set, the application has to send another uplink to trigger the reception. In the case where the application has nothing to transmit but wants to get the pending downlink, the function srv_lora_tx_empty may be used. This function sends an empty uplink (no port and no data).

LoRaWAN DevNonce

The DevNonce (device nonce), is a 16 bits value sent along to the join request. The LoRaWAN 1.0.4 requires that this value increases each time a new join is issued regardless if the device rebooted.
AOS stores this value in the LR1110 flash memory each time the join is successful. Usually this value starts to 0 when on-boarding a device on a LoRaWAN network. It can be convenient to update this value particularly if you change the LoRaWAN network provider. This action can be done by using the srv_lora_set_devnonce function and retrieved via the srv_lora_get_info function.

Network utilities

Along to the basic functions, AOS provides :

- srv_lora_request_time: Request the GPS time and update the system time (UTC).
- srv_lora_link_check: Send a link check request. Usually a link check response is received from the network (if reachable).

6.7 CLI service

6.7.1 Overview

This service manages the Command Line Interface (CLI). It can be connected on the LPUART, on the USB CDC driver or a custom driver.

Notes

- To run at the lowest consumption possible, it is advised to use the LPUART with a baud-rate lower than or equal to 57600 bauds.
- When using the USB CDC connectivity, the main processor cannot use the STOP2 mode. The STOP1 mode is used when the USB cable is disconnected and the main processor runs at the full speed when the cable is connected (no low power mode).
- The UDB CDC driver is seen as an UART for the CLI service point of view.
- The commands can be nested.
- The service supports user authentication.
- The service supports two different access privileges (**user** for normal accesses and **super** with the highest privileges).

6.7.2 Usage

Before using the service is should be initialized via the `srv_cli_init` function.

Once initialized, the service can be opened by the function `srv_cli_open`. The first calling parameter indicates the UART (LPUART or USB CDC) that the service should use.

In this section we call **main command** a command that is directly accessible from the prompt and **sub-command** a command that is nested to a main one.

From an user point of view, the access to a sub-command is done as follow:

main_command sub_command1 sub_command2 etc...

Each word is separated by a space.

6.7.2.1 Command callback

All command callback functions should have the prototype:
`cli_parser_status_t _cli_cmd_t(void *arg, int argc, char *argv[]).`

Where:

- `arg`: Internal argument containing the service context. Do not use.
- `argc`: Number of command arguments found (each argument is separated by a space char). Note that the first argument is always the command (main or sub-command) name.
- `argv`: List of arguments. Each argument is in string format.

The command should return:

- `cli_parser_status_ok`: This will display the string OK
- `cli_parser_status_error`: This will display ERROR
- `cli_parser_status_void`: No display

6.7.2.2 Simple main command creation

A simple main command (not having sub-commands) should be defined by using the macro `CLI_COMMAND_FUNC_REGISTER(name, help, callback, access)`

Where arguments are:

- `name`: Command name as displayed and managed by the serviced. Double quotes should be omitted.
- `help`: Command help. It should be enclosed by double quotes.
- `callback`: Function that will be called once the service matches the name. The function should have the prototype `cli_parser_status_t _cli_cmd_t(void *arg, int argc, char *argv[])`.
- `access`: access privileges granted to the command

6.7.2.3 Main command with sub-commands

A main command with sub-commands should be defined by using the macro `CLI_COMMAND_TAB_REGISTER(name, help, array, access);`

Where arguments are:

- `name`: Command name as displayed and managed by the serviced. Double quotes should be omitted.
- `help`: Command help. It should be enclosed by double quotes.
- `array`: Array of sub-commands accessible at the first level. It should have the prototype `cli_parser_cmd_t`.
- `access`: access privileges granted to the command and sub-commands

The array of sub-commands should be filled with entries using the macros:

- `PARSER_CMD_FUNC`, if the entry is a final sub-command
- `PARSER_CMD_TAB`, if the entry has sub-commands (nested sub-commands)
- `PARSER_CMD_END`, if this is the last entry of the array. I must be present.

The macro `PARSER_CMD_FUNC` is defined as follow:
`PARSER_CMD_FUNC(name, help, callback, access)`
Where:

- `name`: Sub-command name as displayed and managed by the serviced. The format is a C string enclosed by double quotes.
- `help`: Command help. C-String enclosed by double quotes.
- `callback`: Function that will be called once the service matches the name. The function should

have the prototype `cli_parser_status_t _cli_cmd_t(void *arg, int argc, char *argv[])`.
- `access`: access privileges granted to the command

The macro `PARSER_CMD_TAB` is defined as follow:
`PARSER_CMD_TAB(name, help, array, access);`
Where arguments are:
- `name`: Sub-command name as displayed and managed by the serviced. The format is a C string enclosed by double quotes.
- `help`: Command help. It should be enclosed by double quotes.
- `array`: Array of sub-commands accessiblel. It should have the prototype `cli_parser_cmd_t`.
- `access`: access privileges granted to the command and sub-commands

The macro `PARSER_CMD_END` has no parameter.

6.7.2.4 Example 1. Simple command

In this example, we create a main command test, which has no sub-commands.

```
// Callback function
static  cli_parser_status_t _cli_cmd_test(void *arg, int argc, char *argv[])
{
      cli_printf("Test command\n");
      return cli_parser_status_success;
}

// Register main commands against the CLI parser
CLI_COMMAND_FUNC_REGISTER(test1, "My test command", _cli_cmd_help,
CLI_ACCESS_ALL_LEVELS);
```

From the CLI prompt, just enter **test1** followed by a carriage return.

6.7.2.5 Example 2. Nested commands

```
// Callback functions
static  cli_parser_status_t _sub_cmd1(void *arg, int argc, char *argv[])
{
      cli_printf("My Sub command 1\n");
      return cli_parser_status_success;
}

static  cli_parser_status_t _sub_cmd2(void *arg, int argc, char *argv[])
{
      cli_printf("My Sub command 2\n");
      return cli_parser_status_success;
}
```

```
static  cli_parser_status_t _sub_sub_cmd1(void *arg, int argc, char *argv[])
{
      cli_printf("My Sub sub command 1\n");
      return cli_parser_status_success;
}

static  cli_parser_status_t _sub_sub_cmd2(void *arg, int argc, char *argv[])
{
      cli_printf("My Sub sub command 2\n");
      return cli_parser_status_success;
}


// Sub-commands definition for the nested sub-commands
static const cli_parser_cmd_t _nested_cmd_table[] = {
            PARSER_CMD_FUNC("ssc1", "sub-sub-command 1",
_sub_sub_cmd1_ACCESS_ALL_LEVELS),
            PARSER_CMD_FUNC("ssc2", "sub-sub-command 2", _sub_sub_cmd2,
CLI_ACCESS_ALL_LEVELS),
            PARSER_CMD_END
};

// Main sub-commands definition
static const cli_parser_cmd_t _test2_cmd_table[] = {
            PARSER_CMD_FUNC("scmd1", "Sub command 1" _cli_sub_cmd1,
CLI_ACCESS_ALL_LEVELS),
            PARSER_CMD_TAB("nested", "Nested sub-commands", _nested_cmd_table,
CLI_ACCESS_ALL_LEVELS),
            PARSER_CMD_FUNC(""scmd2", "Sub command 2" _cli_sub_cmd2,
CLI_ACCESS_ALL_LEVELS),
            PARSER_CMD_END
};

// Register main commands against the CLI parser
CLI_COMMAND_TAB_REGISTER(test2, "Test 2 commands", _test2_cmd_table,
CLI_ACCESS_ALL_LEVELS );
```

The use will be:

- Enter `test2`, the output will be the help of the command.
- Enter `test2 scmd1`, the output will be: `My sub command 1`.
- Enter `test2 nested`, the output will be the help of the sub-command.
- Enter t`est2 nested ssc2`, the output will be `My sub sub command 2`.
- Try by yourself the other combinations.

6.7.2.6 Display the commands help

The service offers two functions, which displays the help:

- `srv_cli_show_help`: Display the help in the short form: Only the mains command names and their help are displayed.
- `srv_cli_show_help_ext`: Display the help in the long form: All command names and their sub-commands are displayed. The help is also displayed.

Example
The following example uses the two functions. The short form is displayed when the `help` command is entered. The long form is displayed when the `?` character is entered.

```
// Function callbacks
static  cli_parser_status_t _cli_cmd_help(void *arg, int argc, char *argv[])
{
      srv_cli_show_help_ext(argc, argv);
      return cli_parser_status_void;
}

static  cli_parser_status_t _cli_cmd_help_long(void *arg, int argc, char *argv[])
{
      srv_cli_show_help();
      return cli_parser_status_void;
}

// Registration

/*
 * The "?" command needs manual declaration
 */
__attribute__((section(".commands.qmark"),used))
static const cli_parser_cmd_t __cmdfun_qmark = {
            .command = "?",
            .help = "Display all helps",
            { .func = _cli_cmd_help_long },
            .action = cli_parser_action_execute,
            .access = CLI_ACCESS_ALL_LEVELS };

CLI_COMMAND_FUNC_REGISTER(help, "<cmd> Display help information", _cli_cmd_help,
CLI_ACCESS_ALL_LEVELS);
```

6.7.3 Linker script

To use the CLI service the linker script should declare a specific sections where main commands will be located. This section as well as the related definitions are known by the service and underlying drivers. The section should be defined as follow:

```
.commands : {
    . = ALIGN(4);
    __cli_command_table = .;
    KEEP(*(SORT_BY_NAME(.commands.*)));
    LONG (0)     /* end of table */
  } > FLASH
```

Usually this section is placed after the well known `.text` one.

6.7.4 Facilities

The service provides facilities to simplify the commands implementation. Usually the most used are:

- `cli_printf`: Display a formatted string. This is the usual well known printf. Note that it does not support the display of long long integer (64 bits).
- `cli_print_hex`: Display a byte array in hexadecimal. Each value is separated by a space char.
- `cli_print_hex_with_separator`. Same as above with the provided value separator.
- `cli_strcase_ncmp`: Compare two strings regardless the case.
- `cli_print_missing_argument`: Display the "Missing argument" string
- `cli_print_invalid_param_id`: Display the "Invalid parameter ID" string
- `cli_str_on_off`: Display the string "on" or "off"
- `cli_str_yes_no`: Display the “yes” or “no” string.
- `cli_str_success_failure`: Display the string "success" or "failure"
- `cli_print_aos_result`: Display the result in string format.
- `cli_parse_int`: Parse an integer value.
- `cli_parse_float`: Parse a floating point value.
- `cli_xdump`: Display a hex dump of a buffer
- `cli_print_systime`: Display the system date/time

6.7.5 Custom driver

The service accept a custom driver, which can be connected to any physical interface. It’s up to the application developer to implement it.
It should respect the format of the other serial drivers defined by the `aos_uart.h` file. The driver is a structure typed `aos_uart_driver_t` containing the following function pointers:

- `open`: Called when the CLI service is being opened.
- `close`: Called when the CLI service is being closed.
- `read`: Called when the CLI service has been instructed to get received bytes.
- `write`:Called when the CLI service needs to send bytes
- `ioctl`: Called to control the driver

Note:
When the custom driver receives data, it should inform the CLI service. To do this the custom driver should call the user callback function. This callback function is inside the `aos_uart_config_t` provided at the custom driver stage (in this case this is the callback of the CLI service).
Please keep in mind that the user callback function should not be called under the interrupt context.

6.8 BLE beacon service

6.8.1 Overview

This service uses the BLE stack provided by STMicroElectronics. The stack runs partly on the ARM M0+ embedded in the STM32WB55 and on the main core ARM cortex M4.

Note that the BLE stack requires a general initialization. This is done via the function `aos_ble_core_app_init`.

The service performs beacon advertising. The user has a set of predefined beacon types:

- Eddystone UID
- iBeacon
- AltBeacon
- Quuppa
- Exposure

6.8.2 Usage

This service does not need to be initialized. However, when starting this service, the user must set and configure the beacon type.

Beaconing can be started in parallel with scanning and when the device is connected, but it cannot be started when the device is advertising for connectivity. When starting beaconing, the user should fill the config struct `srv_ble_beaconing_param_t` and send it with the desired beacon type using the function `srv_ble_beaconing_start` .

The beaconing operation can be aborted at any time by calling the function `srv_ble_beaconing_stop`.

6.8.2.1 Beacon configuration

Beaconing is configured using the structure `srv_ble_beaconing_param_t`. This structure should be set up before starting a beaconing.

The advertising interval is set via the `adv_interval` parameter. The unit of the advertising interval is milliseconds.

The user can set the calibrated TX power at 0m or 1m via the `calibrated_tx_power` parameter. This value depends on the beacon type as defined in the beacon type standard.

The user can specify the desired TX power via the `tx_level` parameter, representing the TX level as defined by ST microelectronics for the STM32WB55, the correspondence between TX power level and TX power dBm is as follows:

Level	dBm	Level	dBm	Level	dBm	Level	dBm	Level	dBm	Level	dBm

0x00	-40	0x06	-15.25	0x0B	-9.9	0x10	-4.95	0x15	-1.3	0x1A	+1
0x01	-20.85	0x07	-14.1	0x0C	-8.85	0x11	-4	0x16	-0.85	0x1B	+2
0x02	-19.75	0x08	-13.15	0x0D	-7.8	0x12	-3.15	0x17	-0.5	0x1C	+3
0x03	-18.85	0x09	-12.05	0x0E	-6.9	0x13	-2.45	0x18	-0.15	0x1D	+4
0x04	-17.6	0x0A	-10.9	0x0F	-5.9	0x14	-1.8	0x19	0	0x1E	+5
0x05	-16.5									0X1F	+6

The last field to configure is the data to advertise via, the data depends on the beacon type to advertise:

- Eddystone UID: 10 bytes of namespace followed by 6 bytes of instance.
- Ibeacon: 16 bytes for company UUID followed by 2 bytes for major number and 2 bytes for minor number.
- Altbeacon: 4 bytes for manufacturer ID followed by 4 bytes for beacon ID.
- QUUPPA: 1 byte for compensated Tx power followed by 6 bytes for identifier.
- Exposure: 16 bytes for random public identifier followed by 4 bytes for meta data.

6.9 BLE connectivity service

6.9.1 Overview

This service uses the BLE stack provided by STMicroElectronics. The stack runs partly on the ARM M0+ embedded in the STM32WB55 and on the main core ARM cortex M4.

Note that the BLE stack requires a general initialization. This is done via the function `aos_ble_core_app_init`.

The service manages the connectivity between the tracker and a central device.

6.9.2 Usage

This service does not need to be initialized. However, when starting this service the user must set the connectivity configuration.

Connectivity can be started in parallel with scan service only.
Note that connectivity cannot be started when the device is beaconing (2 types of advertising could not be run at the same time).

When starting the connectivity the user should fill the config struct `srv_ble_conn_adv_config_t` and send it using the function `srv_ble_connectivity_start` .

The connectivity operation can be aborted at any time by calling the function `srv_ble_connectivity_stop`.

This service provide a complete set that allow a device to:

- Start/stop advertising
- Perform connection and pairing
- Manage the standard operations (BLE services initialization, connection parameters update, ...)

The user must manage bonding in the application layer as this part can be managed differently by different users.

Any specific processing should be managed in the application layer. To do this, the user must set the application notification callback by calling `aos_ble_core_set_app_callback`. By doing this, all BLE events can be managed in the callback function.

6.9.2.1 Connectivity configuration

The connectivity is configured using the structure `srv_ble_conn_adv_config_t`. This structure should be set up before starting the connectivity.

The minimum and maximum advertising intervals are configured via the `min_interval` and `max_interval` parameters.

The user can set the advertised address type via the `own_address_type` parameter:

- PUBLIC_ADDR: 0X00
- RANDOM_ADDR: 0X01
- STATIC_RANDOM_ADDR: 0X01
- RESOLVABLE_PRIVATE_ADDR: 0X02
- NON_RESOLVABLE_PRIVATE_ADDR: 0X03

The white list is set via the `filter_policy` parameter:

- NO_WHITE_LIST_USE: 0X00
- WHITE_LIST_FOR_ONLY_CONN: 0X02
- WHITE_LIST_FOR_ALL: 0X03

The advertised device name can be configured via the `local_name` parameter. This parameter size cannot be greater than 27 bytes, and its first byte must be 0x08 for Shortened Local Name or 0x09 for Complete Local Name. No NULL character should be at the end.

The `local_name_size` parameter specifies the length of the local name.

The user can add a scan response data via the `scan_resp_data` parameter, this parameter can hold 31 bytes of data formatted as defined in [Vol 3] Part C, Section 11 of the BLE core spec. If the first byte of this parameter is 0, the scan response is disabled.

The user can advertise the list of UUIDs (as defined in Volume 3, Section 11 of GAP Specification) via the `advt_serv_uuid` parameter. The first byte is the AD Type.

The length of UUID list is set via the `advt_serv_uuid_len` parameter.

Note that the slave connection interval min and max are set to 0 when connectivity advertising is started.

6.10 BLE Device Test Mode service

6.10.1 Overview

This service uses the BLE stack provided by STMicroElectronics. The stack runs partly on the ARM M0+ embedded in the STM32WB55 and on the main core ARM cortex M4.

Note that the BLE stack requires a general initialization. This is done via the function `aos_ble_core_app_init`.

The service starts different test modes useful for certification. Three test mode are implemented:

- TX test mode: Sends continuous BLE packets
- RX test mode: Receives continuous BLE packets
- Tone test mode: Sends z tone in a specific BLE channel

6.10.2 Usage

This service should be used when the BLE is in idle state (no scan nor beaconing neither connectivity). Before starting any test the user must initialize this service by calling `srv_ble_dtm_init`, which initializes the context to its default value.

Before starting any test mode the user must to stop any ongoing test mode.

The user can change the default configuration by calling `srv_ble_dtm_set_params`.

After initialization and configuration the user can start one of the three test modes using the appropriate function:

- `srv_ble_dtm_tone_start`: to start tone test mode.
- `srv_ble_dtm_tx_start`: to start TX test mode.
- `srv_ble_dtm_rx_start`: to start RX test mode.

All test modes can be aborted at any time by calling the function `srv_ble_dtm_test_stop`.

6.10.2.1 Test configuration

The user can change the default configuration by calling `srv_ble_dtm_set_params` with new configuration as defined in the structure `srv_ble_dtm_param_t`:

- channel_idx: channel index for tone test, N = (F - 2402) / 2
- data_length: Length in bytes of payload data in each packet for TX test mode
- packet_payload: Type of packet payload for TX test mode, this parameter could be:
  - `srv_ble_dtm_payload_type_pseudo_random_bit_seq_9`: Pseudo-Random bit

sequence 9
    - `srv_ble_dtm_payload_type_alternating_bits_11110000`: Pattern of alternating bits '11110000'
    - `srv_ble_dtm_payload_type_alternating_bits_10101010`: Pattern of alternating bits '10101010'
    - `srv_ble_dtm_payload_type_pseudo_random_bit_seq_15`: Pseudo-Random bit sequence 15
    - `srv_ble_dtm_payload_type_all_ones`: Pattern of All '1' bits
    - `srv_ble_dtm_payload_type_all_zeros`: Pattern of All '0' bits
    - `srv_ble_dtm_payload_type_alternating_bits_00001111`: Pattern of alternating bits '00001111'
    - `srv_ble_dtm_payload_type_alternating_bits_0101`: Pattern of alternating bits '0101'
- phy: PHY to use for test packet in TX and RX test mode:
    - `srv_ble_dtm_phy_le_1m`: Transmitter set to use the LE 1M PHY
    - `srv_ble_dtm_phy_le_2m`: Transmitter set to use the LE 2M PHY
    - `srv_ble_dtm_phy_coded_s8`: Transmitter set to use the LE Coded PHY with S=8 data coding
    - `srv_ble_dtm_phy_coded_s2`: Transmitter set to use the LE Coded PHY with S=2 data coding
- modulation_idx: Modulation index capability of the transmitter in RX test mode:
    - `srv_ble_dtm_modulation_index_standard`: Assume transmitter will have a standard modulation index
    - `srv_ble_dtm_modulation_index_stable`: Assume transmitter will have a stable modulation index

7 Geolocation basic engine

7.1.1 Overview

This service (also called GBE) schedules different types of geolocation technology and provides the results once ended.

It relies on the following underlying services:
- srv_gnss_mt3333: GNSS standalone or aided positions.
- srv_gnss_lr1110: GNSS scan (aided only positions)
- srv_wifi_scan: WIFI scan (BSSID scans)
- srv_ble_scan: BLE beacon scans

The service schedules one technology at a time. Note that a given technology may be either skipped (if the previous one is able to provide a position) or forced (always done regardless the result of the previous technology).

7.1.2 Usage

Before being used, the geolocation services used (GNSS, WIFI, BLE) must be individually initialized before initializing the GBE via the function `srv_geoloc_basic_init`.
The service API defines only 3 others functions:
- `srv_geoloc_basic_start`: Start the geolocation technology scheduler.
- `srv_geoloc_basic_abort`: Abort the geolocation technology scheduler. If a technology is in used it is also aborted.
- `srv_geoloc_basic_get_results`: Returns the results.

Note
The results are kept until another run of the technology scheduling is complete.

7.1.3 Configuration

For each service start, the GBE configuration should be provided. It consists of the list of geolocation technologies to be done.
This configuration is defined by the structure `srv_geoloc_basic_configuration_t`, having the following fields:
- `nb_techno`: Number of technology to schedule.
- `scheduling`: Array containing the technology to use and its configuration. Each entry is typed by `srv_geoloc_basic_cfg_per_techno_t`.

The technology configuration structure contains the following fields:
- `type`: Type of technology.
- `action`: What to do with this technology:
  - `srv_geoloc_basic_action_none`: No action. Technology unused
  - `srv_geoloc_basic_action_always_acquire`: Always do the position acquisition

regardless the previous success.
    - `srv_geoloc_basic_action_skip_if_success`: If there is a previous success do not schedule this technology.
- `cfg`: Technology configuration (depends on the type).

To have a solvable position, the following conditions apply:
- BLE scan: A minimum of 3 beacons should be scanned. Note that 2 different scan (with different parameters are supported).
- WIFI scan: A minimum of 3 BSSID should be scanned.
- Standalone GNSS: A GNSS fix should be done.
- Aided GNSS: Specific criteria applies.

Example
In this example, we want to schedule a standalone GNSS position, in case of failure we want a WIFI position and we want systematically a BLE scan.
The configuration will be:

```
srv_geoloc_basic_configuration_t my_config = {
	.nb_techno = 3,
	.scheduling = {
		{
			// GNSS technology
			.type = srv_geolocation_type_gnss,
			.action = srv_geoloc_basic_action_always_acquire,
			.cfg = {},
		},
		{
			// WIFI technology
			.type = srv_geolocation_type_wifi,
			.action = srv_geoloc_basic_action_skip_if_success,
			.cfg = {},
		},
		{
			// BLE technology
			.type = srv_geolocation_type_ble,
			.action = srv_geoloc_basic_action_always_acquire,
			.cfg = {},
		}

	}
}
```