

A212 Sensor FW

User Guide

A212 Sensor FW

User Guide

Author: Acconeer AB

Version: a2-v1.2.0

Acconeer AB Nov 03, 2026

© 2026 by Acconeer AB - All rights reserved

Contents

1 Introduction ... 6
1.1 Acconeer Sensor FW ... 6
1.2 User Sensor FW ... 6
2 Related documents ... 8
3 Glossary ... 9
4 SDK ... 10
4.1 Folder Structure ... 10
4.2 GCC Toolchain ... 10
4.3 Examples ... 11
4.3.1 Getting Started ... 11
4.3.2 Advanced Control ... 11
4.3.3 System ... 11
4.3.4 Proof of Concept ... 11
5 RSS API Usage ... 12
5.1 API SW Modules ... 12
5.1.1 Config ... 12
5.1.2 Radar ... 12
5.1.3 Processing ... 13
5.2 API Sequence Diagram ... 13
5.2.1 Creating and preparing ... 13
5.2.2 Measure and processing ... 14
5.2.3 Destroying ... 14
5.3 Setting up the Service ... 14
5.3.1 Create Config ... 14
5.3.2 Create Processing ... 14
5.3.3 Allocate Memory ... 15
5.3.4 Powering on Radar ... 15
5.3.5 Calibrate Radar ... 15
5.3.6 Prepare Radar ... 15
5.3.7 Read Radar Temperature ... 16
5.3.8 Get Radar Unique ID ... 16
5.4 Data Retrieval and Processing ... 16
5.4.1 Measure and Read Data from Radar - Blocking ... 16
5.4.2 Measure and Read Data from Radar - Polling ... 16
5.4.3 Sparse IQ Processing ... 17
5.5 Shutdown and Memory De-allocation ... 18
5.5.1 Powering off Radar ... 18
5.5.2 Memory De-allocation ... 18
6 RSS Configuration ... 19
6.1 Configuration Parameter List ... 19
6.2 Configuration Limits ... 19
6.2.1 Limits of Num Points ... 19
6.2.2 Limits of PRF ... 21
6.2.3 Limits of Frame Rate ... 21
6.2.4 Limits of Sweeps Per Frame ... 21
6.2.5 Limits of Sweep Rate ... 21
6.2.6 Limits of Number of Subsweeps ... 22
6.3 HWAAS ... 22
6.3.1 Dynamic HWAAS ... 22
6.3.2 Constant HWAAS ... 22
6.4 Number of Subsweeps ... 22
6.5 Double Buffering ... 22
6.6 High Speed Mode ... 22
6.6.1 Activation Conditions ... 23
7 Radar Frame Data Format ... 24

© 2026 by Acconeer AB - All rights reserved

7.1 Frame data layout ... 24
7.1.1 Nesting order ... 24
7.1.2 Equivalent nested-loop view ... 25
7.2 Using the processing helpers ... 25
7.2.1 Creating the frame info from the configuration ... 25
7.2.2 Computing the index of a sample ... 26
7.2.3 Reading a sample directly ... 26
7.2.4 Example: iterating over all samples of a subsweep ... 26
8 Radar Calibration ... 27
8.1 Radar Re-calibration ... 27
9 RSS Indication Handling ... 28
9.1 Calibration Needed ... 28
9.2 Temperature ... 28
9.3 Calibration Temperature ... 28
9.4 Data Saturated ... 28
9.5 Frame Delayed ... 28
10 RSS Error Handling ... 29
10.1 Error convention ... 29
10.2 Bring-up and in-field errors ... 29
10.2.1 Bring up errors ... 29
10.2.2 In-field errors ... 29
10.3 Error Categories ... 30
10.3.1 Out of Memory Errors ... 30
10.3.2 API Call Order Errors ... 30
10.3.3 API Input Parameter Errors ... 30
10.3.4 Configuration Errors ... 31
10.3.5 Calibration Errors ... 31
10.3.6 Timeout Errors ... 31
10.3.7 System Errors ... 31
10.3.8 Radar Reset Sequence ... 31
11 Radar Data ... 32
11.1 Radar Data Overview ... 32
11.1.1 Multiple Antenna ... 32
11.2 Heatmap ... 32
11.2.1 Beamforming ... 33
11.2.2 Conventional Heatmap ... 33
11.2.3 Capon Heatmap ... 33
11.2.4 Coherent Heatmap ... 33
11.2.5 Heatmap Parameters ... 33
11.3 Heatmap in C ... 34
11.3.1 API Sequence Diagram ... 34
11.3.2 Create heatmap configuration and allocate handle ... 34
11.3.3 Steering vectors ... 35
11.3.4 Noise normalization ... 36
11.3.5 Measure and calculation ... 36
12 System and Drivers ... 38
12.1 RTC ... 39
12.2 APS-PLL ... 39
12.3 Interrupt Handlers ... 40
12.3.1 Avoid Using Floating-Point Operations in Interrupt Contexts ... 40
13 Device Monitoring ... 41
13.1 Diagnostics ... 41
13.1.1 Performed Tests ... 41
13.1.2 Diagnostics Report ... 41
13.2 Self test ... 42
13.2.1 ADC verification ... 42

© 2026 by Acconeer AB - All rights reserved

13.2.2 Factory calibration verification 43
14 Memory 44
14.1 Non-volatile Memory 44
14.2 Volatile Memory 44
14.3 ILM Usage 44
14.4 DLM Usage 44
15 Change Log 45
16 Disclaimer 46

© 2026 by Acconeer AB - All rights reserved

Acconeer SDK Documentation Overview

To better understand which SDK document to use, a summary of the documents is shown in the table below. More general information about Pulsed Coherent Radar, PCR, can be found at docs.acconeer.com.

Name	Description	When to use
RSS API documentation (html)		
rss_api	The complete C API documentation.	• RSS application implementation • Understanding RSS API functions
User guides (PDF)		
A212 Sensor FW	Describes how to develop SW on the A212	• Developing SW inside the A212 sensor
A212 SDK VS Code	Describes how to build the A212 SDK and XE212 SDK, and use the XE212 EVK, in a VS Code dev container	• Working with A212 SDK or XE212 SDK in VS Code
XE212 SDK	Describes how to develop SW on the A212 using the XE212 EVK	• Working with XE212 SDK
Reference Manuals (PDF)		
A212 Technical	Complete information on A212 memory and peripherals	• Advanced development of SW inside the A212 sensor • Development of driver SW inside the A212 sensor
Application Notes (PDF)		
DFU Protocol	Defines the protocol used to communicate with the A212, either with UART or SPI in DFU mode	• Development of bootstrap software for external host
NVM Image Format	Defines the format of the image data to stored in external NVM	• Development of A212 software that loads from external NVM
Exploration Server Protocol	Defines the request/response protocol between clients and the Exploration Server	• Development of client software for the exploration server
Readme (txt)		
README	Various target specific information and links	• After SDK download
BUILDINFO	Compiler information	• To understand pre-compiled code

Table 1: SDK document overview.

© 2026 by Acconeer AB - All rights reserved

1 Introduction

This document describes how to develop sensor firmware (FW) for the A212 sensor. There are two ways to develop software for the A212 sensor:

- *Acconeer Sensor FW*: Acconeer provides the software that runs inside the A212.
- *User Sensor FW*: You develop the software that runs inside the A212.

This document focuses on *User Sensor FW*. The following sections give an overview of both ways.

1.1 Acconeer Sensor FW

When developing using *Acconeer Sensor FW*, the software running inside the A212 sensor is provided by Acconeer. The host communicates with it through a documented protocol, which depends on the *Acconeer Sensor FW* that is used. The host can be a PC or a microcontroller, and Acconeer usually provides example host code.

An example is the Exploration Server application (Acconeer Sensor FW), and the *Exploration Tool GUI* and *Exploration Tool Python API* (example host code, for PC).

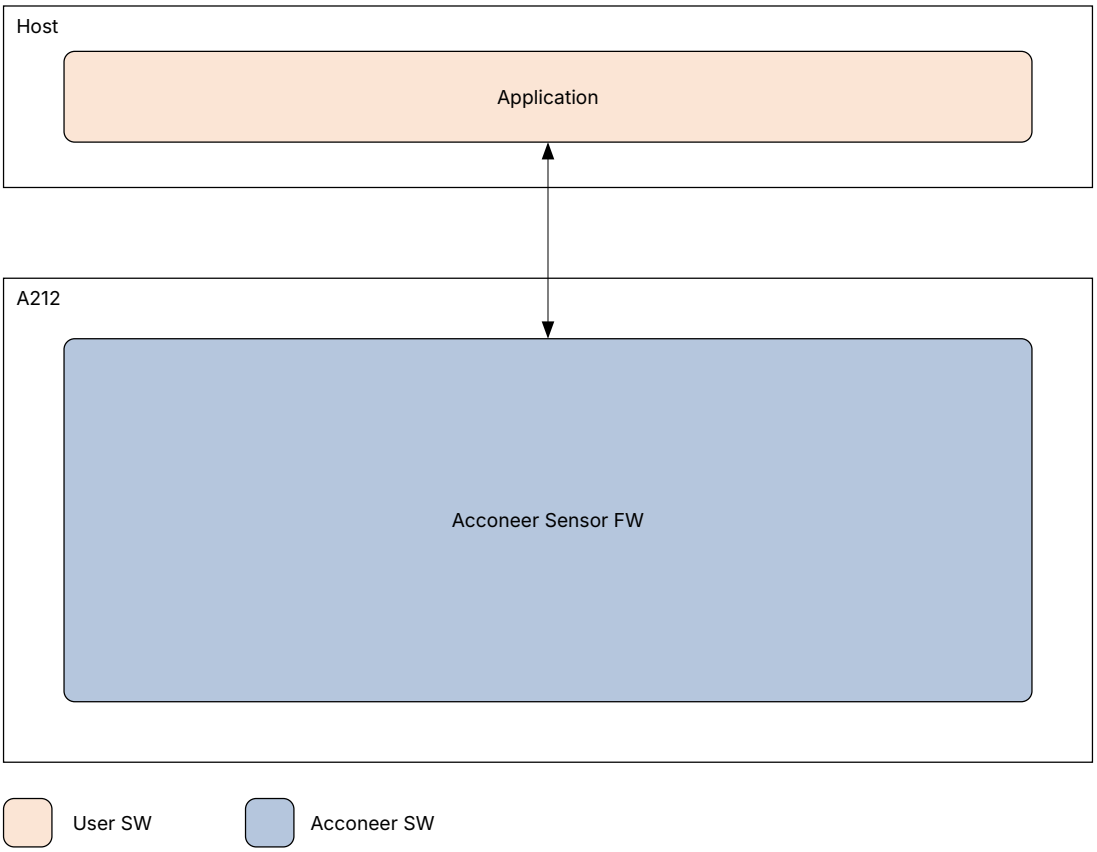


Figure 1: Acconeer Sensor FW Overview.

As shown in Figure 1, all SW running in the A212 is provided by Acconeer.

1.2 User Sensor FW

When developing using *User Sensor FW*, the software running inside the A212 sensor is developed by you with the help of the RSS library, drivers, and examples provided by Acconeer.

© 2026 by Acconeer AB - All rights reserved

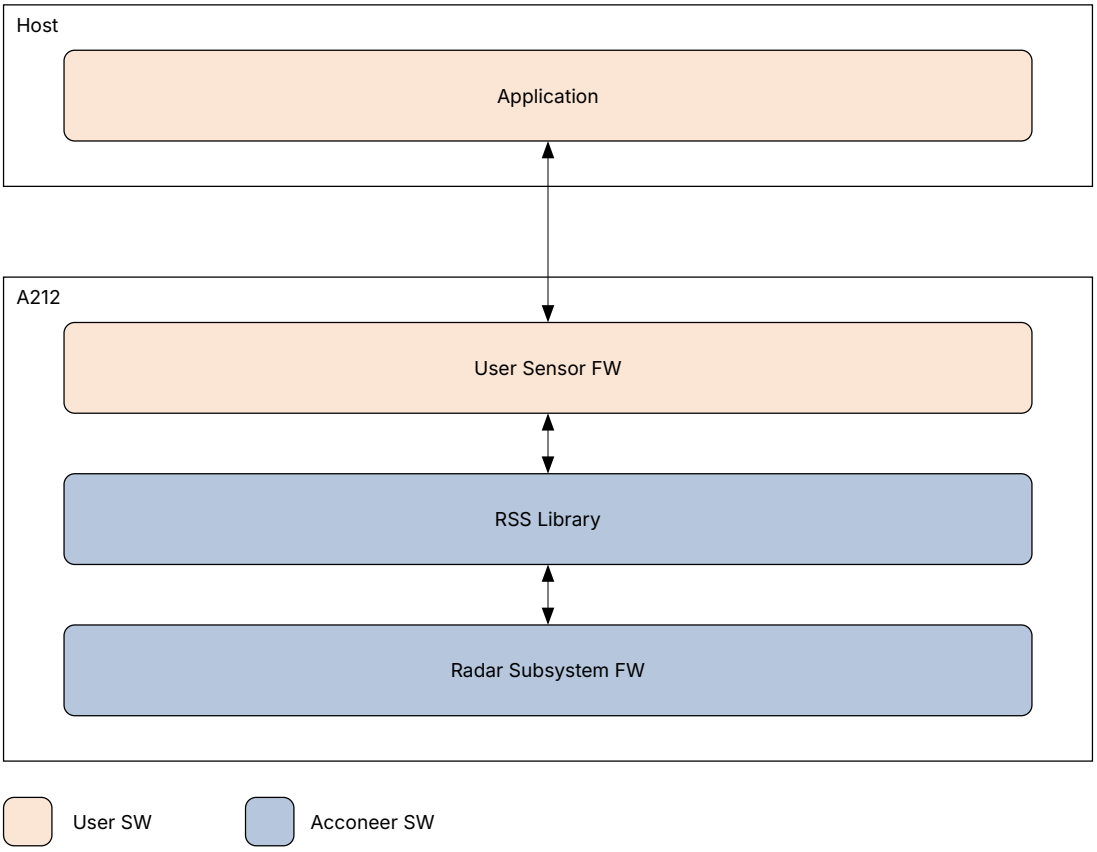


Figure 2: User Sensor FW Overview.

As shown in Figure 2, the SW running in the A212 is both provided by Acconeer and implemented by the customer. The customer can use drivers provided by Acconeer for system setup and communication. To communicate with the Radar Subsystem, which is the part of the sensor that controls the radar, the RSS library must be used. See Section 5 for more information.

© 2026 by Acconeer AB - All rights reserved

2 Related documents

- A212 Technical Reference Manual - Describes the A212 memories and peripherals in detail.
- A212 SDK VS Code User Guide - Describes how to build the SDK in a VS Code dev container.
- A212 DFU Protocol Application Note - Describes how to load sensor FW to the A212 from an external MCU.
- A212 NVM Image Format Application Note - Describes how to load sensor FW to the A212 from an external NVM.
- A212 Exploration Server Protocol Application Note - Describes the protocol of the Exploration Server application.
- rss_api.html - The full RSS API specification, provided in the SDK package.
- example_service_iq.c - Example of a *User Sensor FW* that uses the Sparse IQ Service, provided in the SDK package.

© 2026 by Acconeer AB - All rights reserved

3 Glossary

***ADC* – Analog-to-Digital Converter**: A circuit that converts an analog voltage to a digital value.

***APS* – Application Subsystem**: A digital subsystem in the A212 sensor.

***CRC* – Cyclic Redundancy Check**: An error detecting code used to check the integrity of data.

***DFU* – Device Firmware Update**: A boot mode of the A212 sensor.

***DLM* – Data Local Memory**: The data memory that the ACPU accesses through its local memory interface.

***EVK* – Evaluation Kit**: A board with an Acconeer sensor, used to evaluate the sensor.

***FPU* – Floating Point Unit**: A processor component that performs arithmetic operations on floating-point numbers in hardware.

GCC: The GNU Compiler Collection (GCC).

***GPIO* – General-Purpose Input and Output**: General-purpose digital interface pins.

***HSM* – High Speed Mode**: A radar mode with less overhead between the sweeps in a frame.

***HWAAS* – Hardware Accelerated Average Samples**: The number of samples that the radar hardware averages in each distance point.

***ILM* – Instruction Local Memory**: The instruction memory that the ACPU accesses through its local memory interface.

***ISR* – Interrupt Service Routine**: A function that is executed when an interrupt occurs.

***LDO* – Low-Dropout Regulator**: A linear voltage regulator that supplies a power domain.

***MCU* – Microcontroller Unit**: IC including one or more processor cores, memory and input/output peripherals.

***MIMO* – Multiple-Input Multiple-Output**: A radar with multiple transmit and receive antennas.

***MMD* – Maximum Measurable Distance**: The furthest distance that can be measured with a given PRF.

***MUR* – Maximum Unambiguous Range**: The furthest distance at which a reflection is guaranteed to come from the most recent pulse.

***NVM* – Non-Volatile Memory**: External memory that can be connected to the A212 sensor through the SPI0 interface.

***OTP* – One-Time Programmable**: A memory that can only be programmed once, for example with factory calibration data.

***PIT* – Programmable Interval Timer**: The hardware block that provides programmable interval timers.

***PRF* – Pulse Repetition Frequency**: The frequency at which the radar transmits pulses.

***QSPI* – Quad Serial Peripheral Interface**: An SPI variant able to send 4 data bits per clock cycle.

***RAM* – Random Access Memory**: Internal, volatile memory in A212.

***RS* – Radar Subsystem**: A digital subsystem in the A212 sensor that controls the radar.

***RSS* – Radar System Software**: A pre-compiled library with radar control software provided with the A212 SDK.

***RTC* – Real-Time Clock**: The always-on timer used for wakeup from sleep.

***SNR* – Signal-to-Noise Ratio**: The ratio between the signal power and the noise power.

***SPI* – Serial Peripheral Interface**: A standard interface for serial communications.

***TWI* – Two-Wire Interface**: A synchronous, multi-master/multi-slave, single-ended, serial communication bus compatible with the I2C protocol.

***UART* – Universal Asynchronous Receiver Transmitter**: A peripheral device for asynchronous serial communication in which the data format and the transmission speeds are configurable.

***WDT* – Watchdog Timer**: The hardware block that detects when software stops clearing it in time.

***XTAL* – Crystal Oscillator**: The external 26 MHz crystal oscillator.

© 2026 by Acconeer AB - All rights reserved

4 SDK

This section describes the SDK package, the toolchain that builds it, and the examples it contains.

4.1 Folder Structure

This section describes the SDK package folder structure. This is a general overview, so there might be deviations compared to the exact SDK you are using.

sdk/
- **.devcontainer/**, **.vscode/**
  Includes the dev container and VS Code configuration. See *A212 SDK VS Code User Guide*.
- **a2_dfu/**
  Includes example source files for DFU bootstrapping.
- **doc/**
  Includes the A212 user guides, reference manual, application notes, and the full API documentation.
- **exploration_server/**
  Includes the Exploration Server application which is used when running Exploration Tool.
- **LICENSES/**
  Includes the license files.
- **out/**
  Includes all built files.
- **rss/**
  Includes the RSS library and its header files, see Section 5 for more information.
- **rule/**
  Includes makefiles for building examples etc.
- **script/**
  Includes the `a2_nvm_image.py` tool. See *A212 NVM Image Format Application Note*.
- **source/**
  - **algorithms/**
    Includes algorithm source files, such as heatmap. See Section 11 for more information.
  - **examples/**
    Includes the example applications, in one folder per category. See Section 4.3 for more information.
  - **nvm_boot/**
    Includes an example second-stage bootloader. See *A212 NVM Image Format Application Note*.
  - **system/**
    Includes drivers, such as SPI, clock, and registers. See Section 12 for more information.
    - **rss/**
      Includes drivers used by the RSS library. See Section 12 for more information.

The **sdk/** folder also includes the makefile that builds all examples, and the linker scripts.

4.2 GCC Toolchain

A GCC toolchain is needed to build applications for the A212 Application Subsystem (APS). The recommended way to build the SDK is in the VS Code dev container, which contains the toolchain. See *A212 SDK VS Code User Guide*.

To build without the dev container, install the toolchain yourself. The GCC toolchain is a part of the Andes Development Kit which can be downloaded from https://github.com/andestech/Andes-Development-Kit.

The version to use is **v5.4.1**, and the APS applications are built with the compiler found in the **nds32le-elf-mculib-v5f.txz** archive.

Below is an example of Linux compiler installation and build of the applications in the `sdk` folder.

```
$> tar Jxf nds32le-elf-mculib-v5f.txz
$> sudo mv nds32le-elf-mculib-v5f /opt/
$> export NDS32LE_V5F_ROOT=/opt/nds32le-elf-mculib-v5f
$> cd sdk
$> make
```

© 2026 by Acconeer AB - All rights reserved

4.3 Examples

The SDK provides the examples described in this section.

4.3.1 Getting Started

These examples show basic concepts and can help during bring-up.

- `example_service_iq.c` - Shows basic usage of the Sparse IQ Service
- `example_service_iq_polling.c` - Shows how to poll for a finished radar measurement
- `example_heatmap_capon.c` - Shows how to calculate a Capon heatmap from Sparse IQ data
- `example_heatmap_conventional.c` - Shows how to calculate a conventional heatmap from Sparse IQ data
- `example_heatmap_coherent.c` - Shows how to calculate a coherent heatmap from Sparse IQ data
- `example_diagnostics.c` - Extracts diagnostics from the sensor and prints the result via debug UART. See Section 13.1.

4.3.2 Advanced Control

These examples show advanced control concepts.

- `example_service_iq_double_buffering.c` - Shows usage of double buffering with the Sparse IQ Service
- `example_self_test.c` - Shows usage of the self-test API together with the Sparse IQ Service

4.3.3 System

These examples show how to control system functionality.

- `example_system_gpio_interrupt.c` - Shows how to handle GPIO interrupts
- `example_system_periodic_timer.c` - Shows how to use the PIT as a periodic timer
- `example_system_rtc_sleep_shallow_wakeup.c` - Shows how to put the system to SLEEP-SHALLOW and wake up on an RTC event
- `example_system_rtc_sleep_deep_wakeup.c` - Shows how to put the system to SLEEP-DEEP and wake up on an RTC event
- `example_system_rtc_sleep_ultra_deep_wakeup.c` - Shows how to put the system to SLEEP-ULTRA-DEEP and wake up on an RTC event
- `example_system_spi_slave.c` - Shows how to use the SPI as a slave
- `example_system_twi_slave.c` - Shows how to use the TWI as a slave
- `example_system_uart_echo.c` - Shows how to receive and transmit data with the UART
- `example_system_watchdog_timeout.c` - Shows how to use the WDT

4.3.4 Proof of Concept

These examples show how radar data can be processed into an application.

- `example_motion_detection.c` - Shows how to detect motion from a Capon heatmap

© 2026 by Acconeer AB - All rights reserved

5 RSS API Usage

This section describes how to use the RSS library in more detail. To help with this, a more detailed view of the sensor FW can be seen in the image below.

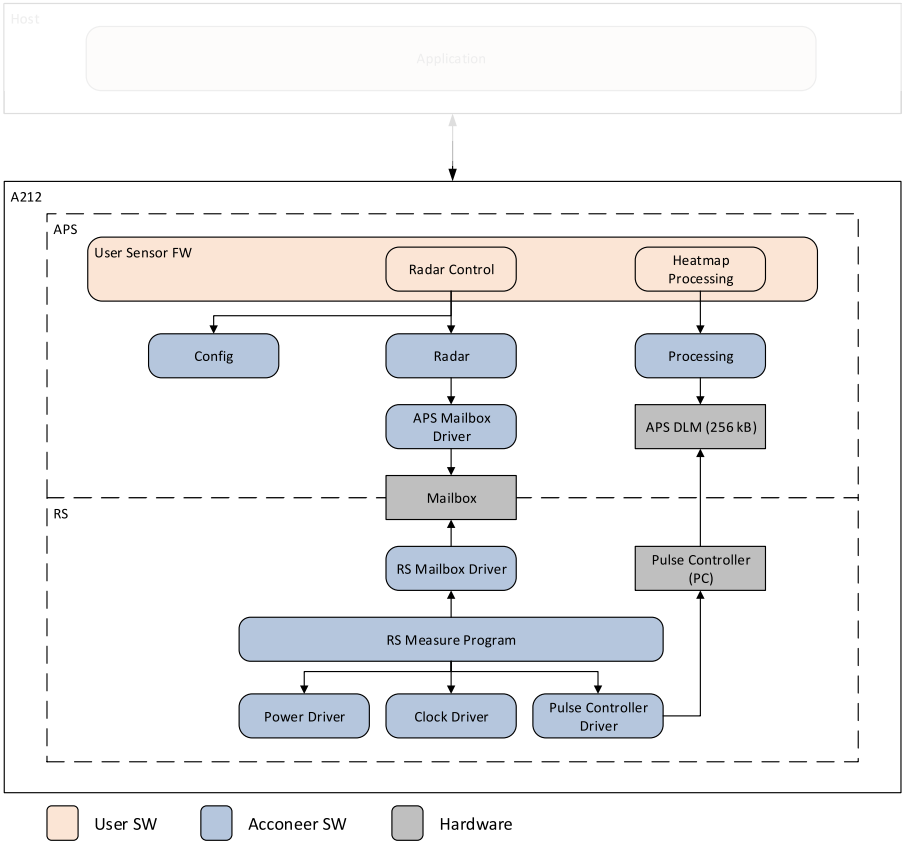


Figure 3: Sensor FW Architecture.

The *Acconeer SW* in Figure 3 is part of the RSS library and is not delivered as source code. The *User SW* is not part of the RSS library, and reference code for it is delivered as source code.

The A212 sensor is divided into two main parts, the *Application Subsystem* (APS) and the *Radar Subsystem* (RS).

In the APS, the customer implements their own SW for their use case (*User SW*). The customer SW interacts with the RS through the RSS library (*Acconeer SW*). The RSS library consists of three API modules, see Section 5.1.

5.1 API SW Modules

The RSS API can be divided into three SW modules. They are each described in the following sections.

5.1.1 Config

SW module used to configure the radar.

The following operations are available in the module:

- Set configurations
- Get configurations

5.1.2 Radar

SW module used to communicate with the Radar Subsystem (RS) FW. The communication is done via a hardware block called *Mailbox*, see Figure 3.

The following operations are available in the module:

- Power On/Off the RS

© 2026 by Acconeer AB - All rights reserved

- Calibrate the radar
- Load a configuration to the RS
- Do a radar measurement
- Wait until radar measurement complete

The radar measurement function will trigger the Radar Subsystem FW to write radar data into the APS DLM.

5.1.3 Processing

SW module used to process a radar measurement into Sparse IQ data.

The following operations are available in the module:

- Process a radar measurement into a Sparse IQ frame
- Extract indications, such as temperature and saturation, from a radar measurement

5.2 API Sequence Diagram

This section shows how the RSS API is typically used through sequence diagrams. The participants in the following diagrams correspond to the blocks in Figure 3. The communication between the *User Sensor FW* and the *RSS library* uses function calls. The communication between the *RSS library* and the *Radar Subsystem FW* is through the *Mailbox*.

5.2.1 Creating and preparing

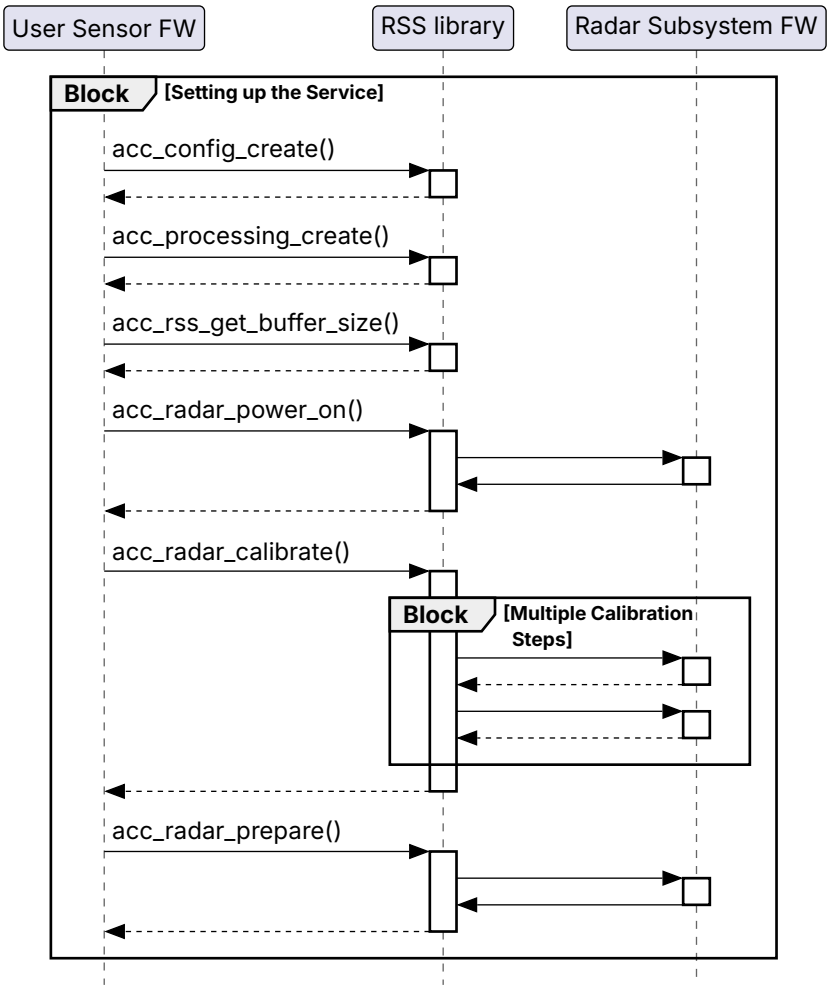


Figure 4: Sequence diagram for how to set up and prepare a radar measurement.

© 2026 by Acconeer AB - All rights reserved

5.2.2 Measure and processing

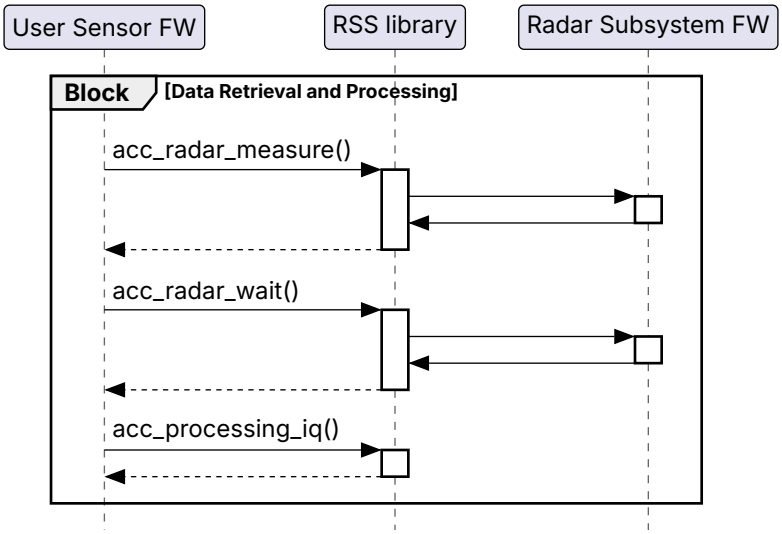


Figure 5: Sequence diagram for how to measure and process radar data.

5.2.3 Destroying

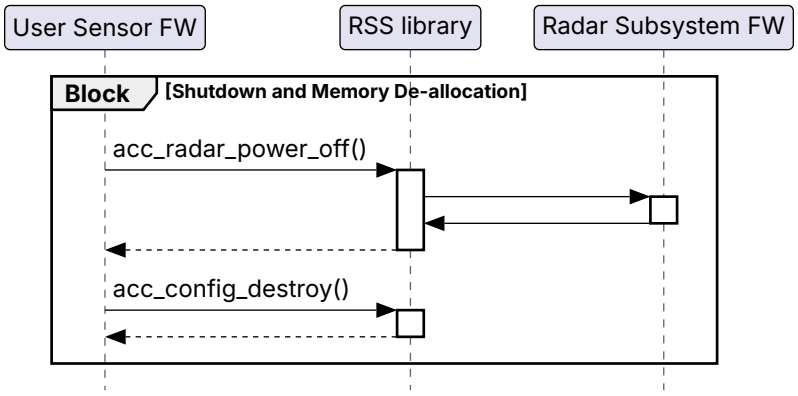


Figure 6: Sequence diagram for how to measure and process radar data.

5.3 Setting up the Service

Below is a step-by-step guide on how to set up and configure the Service.

5.3.1 Create Config

A Config handle must be created before configuration of the radar can be done. The Config handle contains default settings at creation. They can be printed by calling the `acc_config_log()` function. For more information about configuration, see docs.acconeer.com.

```
acc_config_t *config = acc_config_create();

if (config == NULL)
{
    /* Handle error */
}

/* Here the configuration can be changed */
acc_config_start_point_set(config, 80);
acc_config_num_points_set(config, 160);
```

5.3.2 Create Processing

A Processing handle must be created before processing of radar data can be done. The Processing handle is created for a specific configuration.

© 2026 by Acconeer AB - All rights reserved

```
acc_processing_handle_t *processing = acc_processing_create(config);

if (processing == NULL)
{
    /* Handle error */
}
```

5.3.3 Allocate Memory

The memory size needed by RSS depends on the configuration. The `acc_rss_get_buffer_size()` function will get the needed memory size. The application must provide a buffer with `buffer_size` number of bytes, for example allocated with `acc_memory_alloc()` or as a statically declared buffer.

```
uint32_t buffer_size = 0U;

if (!acc_rss_get_buffer_size(config, &buffer_size))
{
    /* Handle error */
}

void *buffer = acc_memory_alloc(buffer_size);
if (buffer == NULL)
{
    /* Handle error */
}
```

5.3.4 Powering on Radar

The Radar must be powered on before any other radar calls are done. As part of the `acc_radar_power_on()` call, the radar is reset.

```
if (!acc_radar_power_on())
{
    /* Handle error */
}
```

5.3.5 Calibrate Radar

The radar needs to be calibrated before radar data can be retrieved. The `acc_radar_calibrate()` function will perform the complete calibration.

```
acc_cal_result_t cal_result;

if (!acc_radar_calibrate(&cal_result, buffer, buffer_size))
{
    /* Handle error */
}
```

The temperature at the time of calibration can be read from the calibration result by calling `acc_radar_get_calibration_temperature()`.

```
int16_t temperature;

if (!acc_radar_get_calibration_temperature(&cal_result, &temperature))
{
    /* Handle error */
}
```

5.3.6 Prepare Radar

The configuration as well as the calibration result are loaded to the radar by calling the `acc_radar_prepare()` function.

© 2026 by Acconeer AB - All rights reserved

```
if (!acc_radar_prepare(config, &cal_result, buffer, buffer_size))
{
    /* Handle error */
}
```

5.3.7 Read Radar Temperature

The current temperature of the sensor, in degrees Celsius, can be read by calling `acc_rss_get_temperature()`.

The function must not be called before `acc_radar_power_on()` and `acc_radar_prepare()` have been called. The radar must be idle when this function is called, i.e. it must not be called between `acc_radar_measure()` and `acc_radar_wait()`.

```
int16_t temperature;

if (!acc_rss_get_temperature(&temperature))
{
    /* Handle error */
}
```

5.3.8 Get Radar Unique ID

Get the unique radar id for this unit.

```
char radar_id[ACC_RADAR_ID_LENGTH];

if (acc_radar_id_get(radar_id))
{
    printf("Radar ID: %s\n", radar_id);
}
else
{
    /* Handle error */
}
```

5.4 Data Retrieval and Processing

5.4.1 Measure and Read Data from Radar - Blocking

The radar measurement is started when the `acc_radar_measure()` function is called. Radar data is continuously written to the `measure_result`. The data is fully written and available in the `measure_result`, after the `acc_radar_wait()` function has returned.

```
acc_measure_result_t *measure_result;

if (!acc_radar_measure())
{
    /* Handle error */
}

if (!acc_radar_wait(&measure_result))
{
    /* Handle error */
}
```

5.4.2 Measure and Read Data from Radar - Polling

The radar measurement is started when the `acc_radar_measure()` function is called. Radar data is continuously written to the `measure_result`. The `acc_radar_poll()` function will return the `radar_ready` flag immediately, allowing the program to handle other tasks while waiting for the radar. The data is fully written and available in the `measure_result`, after the `radar_ready` flag has been set and the `acc_radar_get()` function has returned. This is described in the example `example_service_iq_polling.c`.

© 2026 by Acconeer AB - All rights reserved

```
acc_measure_result_t *measure_result;

if (!acc_radar_measure())
{
    /* Handle error */
}

while (true)
{
    bool radar_ready = false;

    if (!acc_radar_poll(&radar_ready))
    {
        /* Handle error */
    }

    if (radar_ready)
    {
        break;
    }
}

if (!acc_radar_get(&measure_result))
{
    /* Handle error */
}
```

5.4.3 Sparse IQ Processing

To make sense of the radar measurement, it must be processed with `acc_processing_iq()`. This converts the measurement to a Sparse IQ frame and indications. The result is called a *processing result*.

```
acc_processing_iq_result_t proc_result;

acc_processing_iq(processing, config, measure_result, &proc_result);

acc_int16_complex_t *frame = proc_result.frame;

acc_processing_helper_frame_info_t info;

acc_processing_helper_frame_info_get(config, &info);

uint16_t sweeps_per_frame = acc_config_sweeps_per_frame_get(config);
uint16_t num_points       = acc_config_num_points_get(config);
uint16_t num_channels     = acc_config_num_channels_get(config);

printf("%" PRIi16 "degC\n", proc_result.temperature);

for (uint16_t sweep = 0; sweep < sweeps_per_frame; sweep++)
{
    printf("Sweep %" PRIu16 ":\n", sweep);

    for (uint8_t channel = 0; channel < num_channels; channel++)
    {
        printf("Channel %" PRIu8 ":\n", channel);

        for (uint16_t point = 0; point < num_points; point++)
        {
            acc_int16_complex_t value =
                acc_processing_helper_get_value(&info, frame, sweep, 0, point, channel);

            printf("Point %" PRIu16 ": %" PRIi16 "+%" PRIi16 "i\n", point, value.real, value.imag);
        }
    }
}
```

© 2026 by Acconeer AB - All rights reserved

For more information on the indications that are received from a *processing result*, see Section 9.

5.5 Shutdown and Memory De-allocation

5.5.1 Powering off Radar

The Radar can be powered off when not used anymore.

```
acc_radar_power_off();
```

5.5.2 Memory De-allocation

When the radar measurement is done, the handles can be destroyed and the resources can be returned to the system.

```
if (config != NULL)
{
    acc_config_destroy(config);
}

if (processing != NULL)
{
    acc_processing_destroy(processing);
}

if (buffer != NULL)
{
    acc_memory_free(buffer);
}
```

© 2026 by Acconeer AB - All rights reserved

6 RSS Configuration

The radar can be configured with multiple different settings to allow for optimized usage for a specific use case. Each configuration can be set using a function on the format `acc_config_[setting]_set()` in the RSS library.

Some configurations apply to the whole service, such as `num_subsweeps`. Some configurations apply to the radar data, such as `start_point` and `hwaas`.

For more information about configuration, see docs.acconeer.com.

For a complete description of all configurations, see `rss_api.html` provided in the SDK package.

6.1 Configuration Parameter List

Below is a condensed list of all the parameters possible to configure in the RSS API. Note that some of the parameters have more complex constraints than a simple limit, noted by an *. These are described in Section 6.2.

All configuration parameters, including their default, min, and max values are subject to change.

Name	Type	Default Value	Min	Max
start_point	int32_t	80	−200	12000
num_points	uint16_t	40	1	*
step_length	uint16_t	1	1	3072
profile	enum	profile_3	profile_1	profile_7
tx_antenna_mask	uint8_t	tx_antenna_mask_0	n/a	n/a
rx_antenna_mask	uint8_t	rx_antenna_mask_0	n/a	n/a
hwaas_max	uint16_t	50	1	2047
hwaas_min	uint16_t	1	1	2047
hwaas_exponent	uint8_t	0	0	4
receiver_gain	uint8_t	10	0	14
enable_tx	bool	true	n/a	n/a
prf	enum	prf_20_8_mhz	prf_5_3_mhz	prf_27_7_mhz
frame_rate	float32_t	0.0	0.0008	*
sweeps_per_frame	uint16_t	1	1	512
sweep_rate	float32_t	0.0	0.0008	*
num_subsweeps	uint8_t	1	1	5
double_buffering	bool	false	n/a	n/a
inter_frame_idle_state	enum	armed	ready	armed
inter_sweep_idle_state	enum	armed	ready	armed
jitter_compensation	bool	false	n/a	n/a
tx_power_lvl	enum	tx_power_lvl_max	tx_power_lvl_low	tx_power_lvl_max

Table 2: RSS Configuration Parameters

6.2 Configuration Limits

Some of the parameters in Table 2 have more complex constraints than a simple limit that can be visualized in a list. This section describes these parameter limits in more detail.

6.2.1 Limits of Num Points

The max value of `num_points` is limited by the available memory to store radar data in the sensor. The memory used to store radar data is generally DLM in the Application Subsystem (APS). The size of this memory is 192 KB.

This means that `num_points` should be set in a way so that the radar data size does not exceed the available memory. The radar data size is dependent on multiple configuration parameters:

© 2026 by Acconeer AB - All rights reserved

- `num_points` - N_d
- `tx_antenna_mask` - Contributes to the number of channels, N_{ch}
- `rx_antenna_mask` - Contributes to the number of channels, N_{ch}
- `sweeps_per_frame` - N_s
- `num_subsweeps` - N_{ss}

The number of Sparse IQ points in a single subsweep is $N_d * N_{ch}$. One or multiple subsweeps constitute a sweep. If $N_d(0)$ denotes `num_points` of the first subsweep and $N_{ch}(0)$ denotes the number of channels of the first subsweep, the number of Sparse IQ points in a sweep with two subsweeps will be $N_d(0) * N_{ch}(0) + N_d(1) * N_{ch}(1)$, where $N_d(1)$ & $N_{ch}(1)$ denote the `num_points` & number of channels of the second subsweep, respectively.

This can be expressed more formally as

$$(N_d * N_{ch}) * N_s \text{ Sparse IQ points if } N_{ss} = 1, \tag{1}$$

$$\left(\sum_{k=0}^{N_{ss}-1} N_d(k) * N_{ch}(k) \right) * N_s \quad \text{Sparse IQ points if} \quad N_{ss} > 1. \tag{2}$$

The number of bytes a frame *theoretically* occupies is calculated as $(N_d * N_{ch}) * N_s * 4$, if $N_{ss} = 1$[1]. The number of bytes needed to store a frame is the same number rounded up to the closest multiple of 64, due to memory alignment. The number of bytes needed to store a frame is formally expressed as

$$64 \left\lceil \frac{(N_d * N_{ch}) * N_s * 4}{64} \right\rceil, \text{if } N_{ss} = 1 \tag{3}$$

6.2.1.1 Single subsweep example

The number of bytes needed to store a frame when the sensor is configured with

- $N_{ss} = 1$
- $N_d = 40$,
- `tx_antenna_mask` $=$ `TX_0` | `TX_1`, `rx_antenna_mask` $=$ `RX_0` $\Rightarrow N_{ch} = 2$,
- $N_s = 3$,

is

$$64 \left\lceil \frac{(N_d * N_{ch}) * N_s * 4}{64} \right\rceil = 64 \left\lceil \frac{(40 * 2) * 3 * 4}{64} \right\rceil = 64 \left\lceil \frac{960}{64} \right\rceil = 960 \text{ bytes}. \tag{4}$$

6.2.1.2 Multiple subsweeps example

The number of bytes needed to store frames from the sensor when configured with

- $N_{ss} = 3$
- first subsweep
  - $N_d(0) = 30$,
  - `tx_antenna_mask` $=$ `TX_0`, `rx_antenna_mask` $=$ `RX_0` $\Rightarrow N_{ch}(0) = 1$,
- second subsweep
  - $N_d(1) = 20$,
  - `tx_antenna_mask` $=$ `TX_0` | `TX_1`, `rx_antenna_mask` $=$ `RX_0` | `RX_1` $\Rightarrow N_{ch}(1) = 4$,
- third subsweep
  - $N_d(2) = 10$,
  - `tx_antenna_mask` $=$ `TX_0` | `TX_1` | `TX_2`, `rx_antenna_mask` $=$ `RX_0` | `RX_1` | `RX_2` | `RX_3` $\Rightarrow N_{ch}(2) = 12$,
- $N_s = 2$

is

[1] The 4 is due to each Sparse IQ point being stored as 2 `int16` numbers: one `int16` for the real part & one `int16` for the imaginary part. Therefore, one Sparse IQ point occupies 4 bytes.

© 2026 by Acconeer AB - All rights reserved

$$
\begin{aligned}
&64\left\lceil \frac{(N_d(0) * N_{ch}(0) + N_d(1) * N_{ch}(1) + N_d(2) * N_{ch}(2)) * N_s * 4}{64} \right\rceil \\
&= 64\left\lceil \frac{(30 * 1 + 20 * 4 + 10 * 12) * 2 * 4}{64} \right\rceil = 64\left\lceil \frac{1840}{64} \right\rceil = 1856 \text{ bytes.}
\end{aligned} \tag{5}
$$

6.2.2 Limits of PRF

The Pulse Repetition Frequency (PRF) is the frequency at which pulses are sent out from the radar system. Each PRF has two associated properties:

- the *Maximum Measurable Distance (MMD)* and
- the *Maximum Unambiguous Range (MUR)*.

MMD sets the upper limit for the furthest measurable point (see Table 3).

MUR is the maximum distance at which an object can be located to guarantee that its reflection corresponds to the most recent transmitted pulse. Objects farther away than the MUR may fold into the measured range.

For example: With a PRF = 27.7MHz (MUR = 5.4 m), an object at 6.4 m could become visible at 1.0 m.

PRF Setting	**PRF** (MHz)	**MMD** (m)	**MUR** (m)	**MMD** (points)[2]
`ACC_CONFIG_PRF_5_3_MHZ`	5.3	25.0	28.0	10008
`ACC_CONFIG_PRF_8_3_MHZ`	8.3	15.0	18.0	6000
`ACC_CONFIG_PRF_12_5_MHZ`	12.5	9.0	12.0	3600
`ACC_CONFIG_PRF_15_8_MHZ`	15.8	6.6	9.6	2640
`ACC_CONFIG_PRF_20_8_MHZ`	20.8	4.2	7.2	1680
`ACC_CONFIG_PRF_27_7_MHZ`	27.7	2.4	5.4	960

Table 3: PRF settings and their corresponding MMD & MUR

As a rule of thumb: A high PRF allows measurements to be done faster but limits the MMD and MUR. A low PRF makes measurements take longer but gives a higher MMD and MUR.

6.2.3 Limits of Frame Rate

The default is 0.0 even though the minimum is around 0.0008 Hz. The reason is that 0.0 is a special case that leaves the frame rate unlimited. The frame rate is then set by how often the application calls `acc_radar_measure()` and `acc_radar_wait()`.

The maximum frame rate must be low enough to have time to do `sweeps_per_frame` number of sweeps. See Section 6.2.5 for more information.

6.2.4 Limits of Sweeps Per Frame

The number of sweeps per frame that can be used is limited by available memory. See Section 6.2.1 for more details.

6.2.5 Limits of Sweep Rate

The default is 0.0 even though the minimum is around 0.0008 Hz. The reason is that 0.0 is a special case that configures the radar to 'run as fast as possible'.

The maximum sweep rate is limited by the duration of a sweep. The duration of a sweep is dependent on:

- `inter_sweep_idle_state` - A higher idling state decreases the sweep duration.
- `prf` - A higher PRF decreases sweep duration
- `hwaas` - A lower HWAAS decreases the sweep duration
- `num_points` - A lower number of points decreases the sweep duration
- `tx_antenna_mask` - A lower number of activated TX antennas decreases the sweep duration
- `num_subsweeps` - A lower number of subsweeps decreases the sweep duration

[2] This is the maximum *end point*, calculated by the formula $(\texttt{start_point} + \texttt{step_length} * (\texttt{num_points} - 1))$.

© 2026 by Acconeer AB - All rights reserved

The exact sweep duration depends on internal radar hardware and parameters. To get the exact sweep duration, set `sweep_rate` to 0.0, start measuring, and read the `actual_sweep_rate` field in the `acc_processing_iq_result_t` result.

6.2.6 Limits of Number of Subsweeps

The number of subsweeps that can be used is limited by available memory. See Section 6.4 for more details.

6.3 HWAAS

To maintain a constant SNR throughout the entire sweep, the number of HWAAS in each distance point must be increased to compensate for the reduction of SNR as the distance to target increases. The A212 does this with dynamic HWAAS, which is controlled by three parameters:

- `hwaas_exponent` - Controls how HWAAS is scaled with respect to distance.
- `hwaas_min` - The minimum number of HWAAS sampled at one distance point.
- `hwaas_max` - The maximum number of HWAAS sampled at one distance point.

6.3.1 Dynamic HWAAS

When aiming for targets with general shapes, the HWAAS exponent should be set to 4. This is the biggest compensation for the decrease in SNR due to distance, meaning that it has the largest difference between the number of HWAAS in the distance close to the sensor compared to the number of HWAAS at the end of the measurement range. For large planar targets, where the energy spreads less, the exponent should be set to 2.

6.3.2 Constant HWAAS

If the HWAAS exponent is set to 0, constant HWAAS is used for the complete sweep.

6.4 Number of Subsweeps

This configuration sets the number of subsweeps to use.

The configurations that apply for the radar data can be set for each subsweep. The only difference is that `_subsweep` is added to the get/set function names.

```
void acc_config_num_subsweeps_set(acc_config_t *config, uint8_t num_subsweeps);
uint8_t acc_config_num_subsweeps_get(const acc_config_t *config);
```

Example of how to set a subsweep config.

```
void acc_config_subsweep_start_point_set(acc_config_t *config, int32_t start_point, uint8_t index);
int32_t acc_config_subsweep_start_point_get(const acc_config_t *config, uint8_t index);
```

6.5 Double Buffering

This configuration enables/disables the use of double buffering.

When double buffering is enabled, it is possible to call `acc_radar_measure()` and then start processing the data frame that was returned in the last `acc_radar_wait()`.

When double buffering is disabled, the data frame returned from `acc_radar_wait()` must be processed before `acc_radar_measure()` is called to get the next data frame.

The amount of needed memory returned by `acc_rss_get_buffer_size()` might be larger when double buffering is enabled.

When double buffering is enabled the API call order, to `acc_radar_measure()`, `acc_radar_wait()` and `acc_processing_iq()`, can be optimized to achieve a higher frame rate, which in turn could decrease the power consumption. This is described in the example `example_service_iq_double_buffering.c`.

6.6 High Speed Mode

High Speed Mode (HSM) allows the radar sensor to measure faster by optimizing the overhead between consecutive sweeps within a frame. It's automatically enabled when certain configuration conditions are met.

© 2026 by Acconeer AB - All rights reserved

6.6.1 Activation Conditions

HSM is activated under the following conditions:

- `sweeps_per_frame > 1`
- `num_subsweeps == 1`

© 2026 by Acconeer AB - All rights reserved

7 Radar Frame Data Format

7.1 Frame data layout

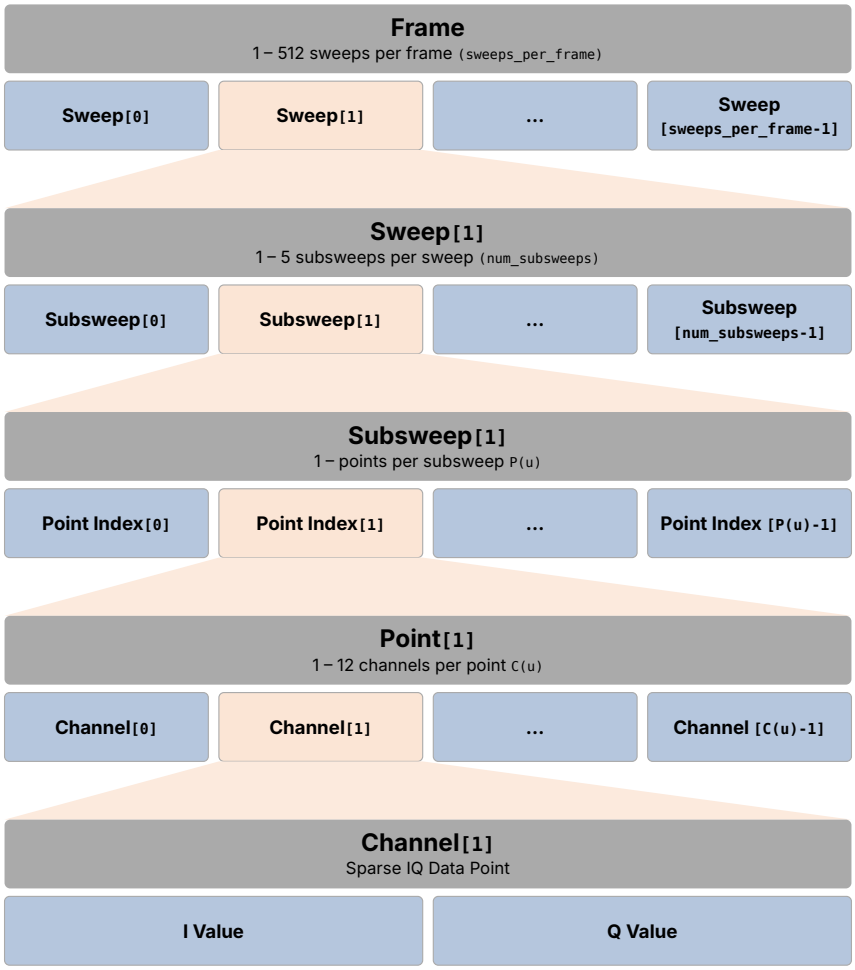
The entire frame is delivered as a single one-dimensional array of `acc_int16_complex_t` elements. Each element occupies 4 bytes: a 16-bit signed real component followed by a 16-bit signed imaginary component. The array contains no headers, padding, or separators between sweeps, subsweeps, points, or channels — it is a flat, contiguous run of complex samples, and the boundaries between sweeps/subsweeps/points/channels exist only implicitly, by position.

7.1.1 Nesting order

The array is laid out in strict row-major order, following the same nesting hierarchy as the frame structure itself, from **slowest-varying** (outermost) to **fastest-varying** (innermost):

1. Sweep – index `s`, ranges over `0` to `sweeps_per_frame` − 1
2. Subsweep – index `u`, ranges over `0` to `num_subsweeps` − 1
3. Point – index `p`, ranges over `0` to `P(u)` − 1
4. Channel – index `c`, ranges over `0` to `C(u)` − 1

This means: the **channel** index increments fastest (all channels for a given point are stored consecutively), then the **point** index advances (all points for a given subsweep follow each other), then the **subsweep** index advances, and finally the **sweep** index – the outermost, slowest-changing dimension – advances last, once an entire sweep's worth of subsweeps/points/channels has been written.



Each highlighted box is the one "expanded" into the level below: a frame holds sweeps, a sweep holds subsweeps, a subsweep holds points, and a point holds one sample per channel.

Figure 7: Radar Frame Data Format.

© 2026 by Acconeer AB - All rights reserved

Note: Each subsweep has its own configuration. Its number of points `P(u)` comes from num_points, and its number of channels `C(u)` is determined by `tx_antenna_mask` and `rx_antenna_mask`.

Therefore, subsweeps within the same sweep may have different point and channel counts. The inner dimensions of the array are variable, and the size of each subsweep block depends on the current subsweep `u`.

The storage order is still fixed: subsweep `u+1` always follows immediately after subsweep `u`, regardless of their sizes.

7.1.2 Equivalent nested-loop view

The array is exactly what you would get by writing out the samples with four nested loops, channel innermost:

```
void measure_frame(acc_int16_complex_t *data)
{
  size_t array_index = 0;

  for (int s = 0; s < sweeps_per_frame; s++) {
    for (int u = 0; u < num_subsweeps; u++) {
      for (int p = 0; p < num_points[u]; p++) {
        for (int c = 0; c < num_channels[u]; c++) {
          printf("Data point at sweep=%d, subsweep=%d, point=%d, channel=%d = (%i, %i)\n",
                 s,
                 u,
                 p,
                 c,
                 data[array_index].real,
                 data[array_index].imag);
          array_index++;
        }
      }
    }
  }
}
```

7.2 Using the processing helpers

The SDK provides a set of processing helpers, declared in `acc_processing_helpers.h`, that implement the arithmetic needed to acquire a specific sample from `(sweep_index, subsweep_index, point, channel_index)`. Instead of computing subsweep offsets and strides manually, applications can use these helpers to locate individual samples in the frame data array.

7.2.1 Creating the frame info from the configuration

The frame data array itself contains no layout information – it is a flat run of `acc_int16_complex_t` samples. All information needed to navigate it (subsweep offsets, subsweep lengths, channel counts, and the sweep stride) is derived from the sensor configuration. This derivation is performed by `acc_processing_helper_frame_info_get`:

```
acc_processing_helper_frame_info_t frame_info;
acc_processing_helper_frame_info_get(config, &frame_info);
```

The function walks through the subsweeps of the supplied `acc_config_t` in order. For each subsweep it reads the per-subsweep `num_points` and channel count from the configuration, and accumulates a running offset:

- `subsweeps[u].data_offset` – where subsweep `u` starts within a sweep, equal to the sum of the lengths of all preceding subsweeps ($\text{sub_offset}(u)$).
- `subsweeps[u].data_length` – the number of elements in subsweep `u`, $\text{size}(u) = P(u) \cdot C(u)$.
- `subsweeps[u].num_channels` – the subsweep's channel count $C(u)$, used as the point stride.

After the last subsweep, the accumulated offset equals the total length of one sweep and is stored as `sweep_length` (sweep_stride in the index formula).

Since the layout is determined entirely by the configuration, the frame info is valid for every frame produced with that configuration. It should therefore be created once, after the configuration is finalized, and reused for all subsequent frames. It only needs to be recreated if the configuration changes.

© 2026 by Acconeer AB - All rights reserved

7.2.2 Computing the index of a sample

`acc_processing_helper_get_frame_offset` converts the four logical coordinates – sweep, subsweep, point, and channel – into a flat array index:

```
uint32_t offset = acc_processing_helper_get_frame_offset(&frame_info,
                                                         sweep_index,
                                                         subsweep_index,
                                                         point,
                                                         channel_index);
```

$$\text{offset} = s \cdot \text{sweep_length} + \text{data_offset}(u) + p \cdot \text{num_channels}(u) + c \tag{6}$$

which corresponds term by term to

$$\text{index}(s, u, p, c) = s \cdot \text{sweep_stride} + \text{sub_offset}(u) + p \cdot C(u) + c \tag{7}$$

The returned value is an **element** index into the `acc_int16_complex_t` array, not a byte offset.

7.2.3 Reading a sample directly

For convenience, `acc_processing_helper_get_value` combines the offset calculation with the array access and returns the sample itself:

```
acc_int16_complex_t value = acc_processing_helper_get_value(&frame_info,
                                                            frame,
                                                            sweep_index,
                                                            subsweep_index,
                                                            point,
                                                            channel_index);
```

This is equivalent to `frame[acc_processing_helper_get_frame_offset(...)]`.

7.2.4 Example: iterating over all samples of a subsweep

The helpers make it straightforward to traverse the frame in any order without knowledge of the underlying memory layout. For example, averaging a given subsweep over all sweeps in the frame – a common noise-reduction step – iterates point by point and channel by channel, using the helper to resolve each sample's position:

```
for (uint32_t p = 0U; p < num_points; p++)
{
    for (uint8_t ch = 0U; ch < num_channels; ch++)
    {
        float sum_real = 0.0f;
        float sum_imag = 0.0f;

        for (uint16_t s = 0U; s < sweeps_per_frame; s++)
        {
            uint32_t offset = acc_processing_helper_get_frame_offset(&frame_info, s, subsweep_idx, p, ch);
            sum_real += crealf(frame[offset]);
            sum_imag += cimagf(frame[offset]);
        }

        mean[(p * num_channels) + ch] = (sum_real / sweeps_per_frame) + (sum_imag / sweeps_per_frame) * I;
    }
}
```

© 2026 by Acconeer AB - All rights reserved

8 Radar Calibration

The radar needs to be calibrated before radar data can be retrieved. The calibration is done by calling the `acc_radar_calibrate()` function in the RSS library.

Without a successful calibration, the behavior of the radar data is undefined. It can for example degrade the quality of the data or make the data completely useless. The environment does not need to be known for the radar calibration to work. This means that a (re)calibration can be done at any time.

The calibration routine extracts data from the radar and evaluates it. There are sanity checks on this radar data. A temporary external disturbance might affect the data, which will trigger failures in the sanity checks. This should only happen very rarely. If it happens, a retry of the calibration should solve the issue. If the issue persists, there might be an issue with the radar or integration.

The Radar Calibration can internally be divided into three different types of calibrations:

1. *Factory calibrations*: Most calibrated settings are calibrated in the factory and stored in a read-only memory. These settings are fetched from the read-only memory in the call to `acc_radar_calibrate()`. This includes:
   - Transmitter and receiver RF frequency settings
   - Other hardware settings internal to the radar
2. *Temperature compensations*: Some of the factory calibrated settings are also compensated for changes in current temperature in `acc_radar_calibrate()`. This includes:
   - Transmitter and receiver RF frequency settings, to compensate for frequency drift with temperature.
   - Transmitter power amplifier bias setting, to compensate for reduced gain at higher temperatures.
3. *Runtime calibrations*: The settings related to shape and width of the radar pulses are not factory calibrated or temperature compensated but instead measured and calculated in `acc_radar_calibrate()`.

8.1 Radar Re-calibration

Sometimes, the sensor needs to be re-calibrated. This is indicated by the `calibration_needed` indication. It is set if

- the temperature differs more than 30 degrees Celsius from the temperature at the last successful calibration,
- the calibration was done below −10 degrees Celsius and the temperature is now above 0 degrees Celsius, or
- the calibration was done at or above −10 degrees Celsius and the temperature is now below −20 degrees Celsius.

For more information, see Section 9.

© 2026 by Acconeer AB - All rights reserved

9 RSS Indication Handling

The result from a call to `acc_processing_iq()` includes both the radar data frame as well as indications. The example below shows how to fetch indications from the result.

```
acc_processing_iq_result_t proc_result;

acc_processing_iq(processing, config, measure_result, &proc_result);

if (proc_result.calibration_needed)
{
    // Handle calibration needed indication
}
```

Some of the indications are good-to-have metadata while some are crucial to handle for stable operation of the sensor. The full list of indications and appropriate handling of them is described below.

9.1 Calibration Needed

Indication that the sensor calibration isn't valid anymore and needs to be redone.

This happens when the temperature has changed compared to when the calibration was done. See Section 8.1 for the conditions.

9.2 Temperature

Indication of the sensor temperature, in degrees Celsius, for the current measurement.

9.3 Calibration Temperature

Indication of the sensor temperature during the last successful calibration, in degrees Celsius.

9.4 Data Saturated

Indication of sensor data being saturated, which causes data corruption. There is one indication per subsweep.

This happens when a reflection is so strong that the ADC in the sensor gets saturated.

Handle by lowering the receiver gain configuration.

9.5 Frame Delayed

Indication of a delayed frame.

This can only happen when `frame_rate` is not 0.0. It happens when the configured frame rate cannot be kept, for example because the application does not call `acc_radar_measure()` often enough.

Handle by lowering the frame rate configuration.

© 2026 by Acconeer AB - All rights reserved

10 RSS Error Handling

This section covers how RSS reports errors and how to handle those errors.

10.1 Error convention

RSS reports errors with status returns and logs. Some functions that might fail return a `bool` to indicate success (`true`) or failure (`false`), for example the `acc_radar_*` functions (`acc_radar_prepare`, `acc_radar_calibrate`, etc.). A failed call (that returns `false`) will also log what went wrong. Logging is not exclusive to functions that return a `bool` but is used throughout the RSS, for example in the `create()` functions, which return a pointer and not a `bool`. If a `create()` function fails, it will return a `NULL` pointer and log what went wrong.

The logs are categorized in groups by a prefix, for example `'Memory:'` will indicate that a memory allocation has failed. For more information on the error categories, see Section 10.3.

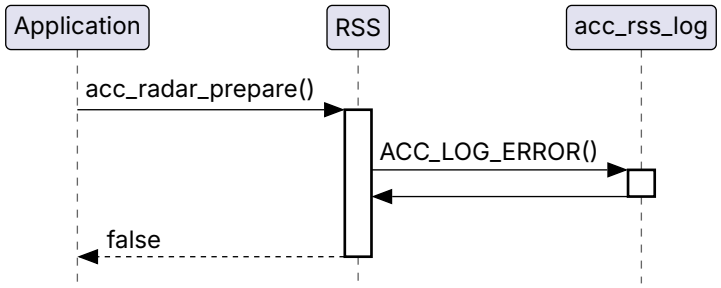


Figure 8: Example of the sequence where acc_radar_prepare() fails.

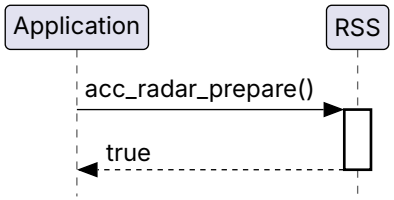


Figure 9: Example of the sequence where acc_radar_prepare() succeeds.

10.2 Bring-up and in-field errors

Acconeer separates errors into *errors that happen during bring up* and *errors that happen in the field*. The following sections describe them in more detail.

10.2.1 Bring up errors

These kinds of errors can originate from

- Misusing the RSS API (For example, not handling `NULL` pointers.)
- Faulty integration (Errors in memory allocation or communication)

During bring up, the error handling convention enables developers to quickly discover what is wrong as they are able to find *where* the error occurred (where a function returns `false`) and *what* was wrong (by reading the log).

10.2.2 In-field errors

When all bring-up errors are handled, what is left is the in-field errors.

If one of these errors occurs, the simplest mitigation is to restart the radar system. Some specific scenarios allow a smarter mitigation. Those are listed in Table 4:

Error description	Mitigation
Sensor calibration fails	Reset the radar and retry the calibration, see Section 10.3.5
Timeout or system error	Reset the radar and retry the failing function, see Section 10.3.6 and Section 10.3.7
Something unforeseen went wrong	Restart the radar system

Table 4: Mitigation table

© 2026 by Acconeer AB - All rights reserved

10.3 Error Categories

Table 5 lists the error categories that can occur in the RSS and which functions can generate them. The error categories are based on the log prefixes, for example `'Memory:'` indicates that the error is an Out of Memory error.

Error category	Log prefix	Function
Out of Memory Error	Memory:	`acc_config_create()` `acc_processing_create()`

API Call Order Error	Usage:	`acc_radar_calibrate()`
		`acc_radar_get()`
		`acc_radar_measure()`
		`acc_radar_poll()`
		`acc_radar_prepare()`
		`acc_rss_get_temperature()`

API Input Parameter Error	Input Parameter:	All API functions check their input parameters and can return input parameter errors if they are wrong, for example if `buffer_size` is `NULL` in `acc_rss_get_buffer_size(const acc_config_t *config, uint32_t *buffer_size)`

Configuration Error	Config:	`acc_processing_create()`
		`acc_radar_prepare()`
		`acc_rss_get_buffer_size()`

Calibration Error	Calibration:	`acc_radar_calibrate()`
Timeout Error	Timeout:	`acc_radar_calibrate()` `acc_radar_wait()`

System Error	System:	`acc_radar_calibrate()`
		`acc_radar_get()`
		`acc_radar_power_on()`
		`acc_radar_prepare()`
		`acc_radar_wait()`
		`acc_self_test_adc_test()`

Table 5: Error Categories

10.3.1 Out of Memory Errors

Memory allocation occurs in the `create()` functions, for example in `acc_config_create()` and `acc_processing_create()`. If the memory allocation fails, there will be an error message in the log and the function will return a `NULL` pointer.

The memory allocation errors have a `'Memory:'` prefix in the log.

The memory allocation errors should be handled during the bring-up phase.

10.3.2 API Call Order Errors

The RSS API call order is important, for example the sensor needs to be calibrated and prepared before a radar measurement can be performed. If the API call order is wrong, the called function will fail (return `false`).

The API call errors have a `'Usage:'` prefix in the log.

The API call errors should be handled during the bring-up phase.

10.3.3 API Input Parameter Errors

The RSS API functions will check the input parameters to make sure they are correct. If one or several of the API input parameters are wrong, there will be an error message in the log and the called function will fail (return `false`).

The API input parameter errors have a `'Input Parameter:'` prefix in the log.

The API input parameter errors should be handled during the bring-up phase.

© 2026 by Acconeer AB - All rights reserved

10.3.4 Configuration Errors

The configuration of the radar will be checked during `acc_rss_get_buffer_size()`, `acc_processing_create()` and `acc_radar_prepare()`. If an invalid configuration is detected, there will be an error message in the log and the function will return `false`.

The configuration errors have a `'Config:'` prefix in the log.

The configuration errors should be handled during the bring-up phase.

10.3.5 Calibration Errors

The radar sensor is calibrated in multiple steps. In each step, the calibration routine extracts data from the radar and evaluates it. There are sanity checks of this radar data. A temporary external disturbance might affect the data, which will trigger failures in the sanity checks. This should only happen very rarely.

The calibration errors have a `'Calibration:'` prefix in the log. The application needs to check for failure (`false`) each time the function `acc_radar_calibrate()` is called.

If a calibration error occurs, the calibration process should be retried. The radar must be reset according to Section 10.3.8 before the function `acc_radar_calibrate()` is called again. If the calibration error persists, there might be an issue with the radar sensor or integration.

10.3.6 Timeout Errors

A timeout error should never occur in a fully functional radar sensor. The functions `acc_radar_calibrate()` and `acc_radar_wait()` can potentially return `false` due to a timeout failure.

The timeout errors have a `'Timeout:'` prefix in the log. The application needs to check for failure (`false`) each time one of these functions is called.

If a timeout error occurs, the radar must be reset according to Section 10.3.8 before the failing function is called again. If the timeout error persists, there might be an issue with the radar sensor or integration.

The timeout is dynamically set to match the time it takes to measure a frame plus a 1000 ms margin.

10.3.7 System Errors

A system error should never occur in a fully functional radar sensor. See Table 5 for the full list of functions that can potentially fail due to a system failure.

The system errors have a `'System:'` prefix in the log. The application needs to check for failure (`false`) each time one of these functions is called.

If a system error occurs, the radar must be reset according to Section 10.3.8 before the failing function is called again. If the system error persists, there might be an issue with the radar sensor or integration.

Another source of this error is if the factory calibrated values are not used. See `acc_factory_calibration.h` for more information.

10.3.8 Radar Reset Sequence

The radar system can be reset by calling `acc_radar_power_off()` followed by `acc_radar_power_on()`. This will reset the radar system and allow it to recover from most errors. If the error persists after this, there might be an issue with the radar sensor or integration.

Calling `acc_radar_power_off()` followed by `acc_radar_power_on()` takes around 2.1 ms to perform.

© 2026 by Acconeer AB - All rights reserved

11 Radar Data

11.1 Radar Data Overview

The measured radar data can be processed into two types.

- *Sparse IQ*: Raw radar data in a complex format
- *Heatmap*: Envelope of Sparse IQ data for multiple angles

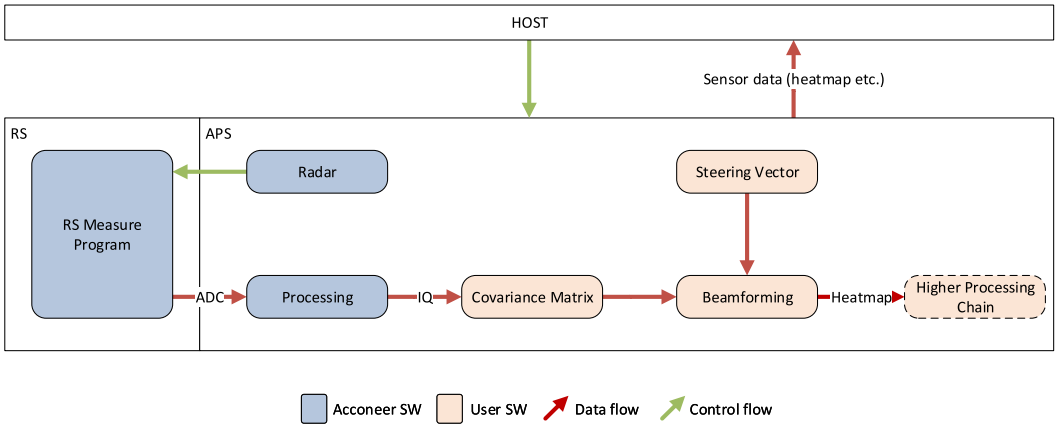


Figure 10: Radar Data Overview.

A measurement is configured and controlled by the *Radar* module (more info in Section 5). Sparse IQ data is produced by the *processing* module. This data can be provided to the host by *User SW* or processed further to *Heatmap data* (more info in Section 11.3). The Heatmap data can either be provided to the host or processed further (*Higher Processing Chain*).

To read more about how to interpret the different radar data types, see docs.acconeer.com.

It is highly recommended to use the Acconeer Exploration Tool, together with an Acconeer Evaluation Kit (EVK), to explore what the radar data looks like and how it can be used.

For more information on the Acconeer EVKs, see www.acconeer.com/products.

11.1.1 Multiple Antenna

The A212 radar is a multiple-input multiple-output (MIMO) radar. It consists of three Tx antennas and four Rx antennas, which yield 12 antenna pair combinations, called channels. Three Rx antennas are lined up in one direction, azimuth, while the last one is placed in the orthogonal direction, elevation. Thus, the azimuth direction has better angle resolution. This means that the orientation of the radar is of great importance.

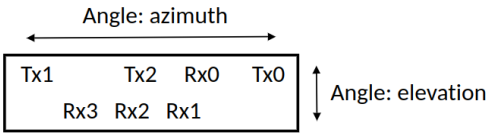


Figure 11: Antenna Overview.

One sweep of data will consist of Sparse IQ data points for each distance in combination with all channels. The multiple antenna array can be adjusted to fit the purpose of the use case in mind. Use all antennas to activate all channels for increased SNR and angle resolution. Depending on the use case and power consumption constraints, one to three Tx antennas can be used together with all four Rx antennas. Note that deactivating Rx antennas only gives a minor decrease in power consumption, hence all Rx antennas should be used to get the best performance. The activation of the antennas is determined by the `tx_antenna_mask` and the `rx_antenna_mask`, see Section 6 for the full list of sensor configuration parameters.

11.2 Heatmap

A heatmap is a 2D representation of data. When using beamforming for multichannel arrays to compute the heatmap, each value corresponds to a distance and an angle. High values (hot) indicate detection in that location,

© 2026 by Acconeer AB - All rights reserved

while low values (cool) indicate that little or no signal is detected. In a graphical presentation, the value range is represented by different colors.

11.2.1 Beamforming

Beamforming is a signal processing method for multichannel arrays which focuses on the energy coming from one direction while minimizing interference from other directions. This can be used to create heatmaps or point clouds.

The number of angles to calculate values for could be limited to reduce computational cost. It should be remembered that due to the physically low resolution in the elevation plane, there is often no gain in estimating values for a large number of angles in that plane.

11.2.2 Conventional Heatmap

The conventional heatmap processor utilizes standard delay and sum beamforming to create heatmaps. The advantage of this algorithm is low computational complexity, while the drawback is lower angular resolution and larger side lobes.

The conventional heatmap processor has a distance mode and a movement mode. The distance mode allows for measurements on both static and moving objects, while movement mode filters out static objects in the scene.

11.2.3 Capon Heatmap

The Capon heatmap processor yields better angular resolution and smaller side lobes compared to the conventional heatmap processor. However, the processing is more computationally expensive.

With the Capon processor there is no option to choose whether to detect static or moving objects. This processor will only detect movement.

11.2.4 Coherent Heatmap

The coherent heatmap was developed to keep the instantaneous phase information in the sampled data. It is therefore a complex-valued heatmap. This could be useful when tracking the phase is of interest. In the same way as the conventional heatmap processor, the coherent heatmap processor can be configured to detect moving or static objects, and it has the same drawbacks as the conventional heatmap processor with lower angular resolution and larger side lobes. This heatmap processor is slightly less computationally complex than the conventional heatmap processor.

11.2.5 Heatmap Parameters

To optimize the processing for a specific use case, the following parameters can be adjusted:

- `covariance_matrix_time_constant`: Adjusts the exponential filtering of the covariance matrices. A higher time constant leads to more averaging, hence better estimation and less sensitivity to time variance. However, a lower time constant makes the processing adapt faster when the environment changes. If the response is too sluggish, it should be decreased, and if the response is too flickery, it should be increased. Not applicable for coherent heatmap processing.
- `high_pass_filter_time_constant`: Sets how fast the background is filtered out. Changes in the background are filtered out faster when set to a lower value. The purpose is to filter out static objects in the scene. For coherent and conventional heatmap processing, this is only applicable when movement mode is active.
- `num_az_angles`: Number of angles to calculate in the azimuth plane (high resolution plane).
- `az_angle_range`: Angle range in the azimuth plane.
- `num_el_angles`: Number of angles to calculate in the elevation plane (low resolution plane).
- `el_angle_range`: Angle range in the elevation plane.

© 2026 by Acconeer AB - All rights reserved

11.3 Heatmap in C

Conventional, Capon and coherent heatmaps are included in the C SDK. All these types of heatmap have a similar API and the same control flow. The coherent heatmap differs in that its result is complex-valued, where the phase can be retrieved with `cargf()` and the amplitude with `cabsf()`.

These examples show how to configure and calculate a heatmap:

- `example_heatmap_capon.c` - Shows how to calculate a Capon heatmap from IQ data
- `example_heatmap_coherent.c` - Shows how to calculate a coherent heatmap from Sparse IQ data
- `example_heatmap_conventional.c` - Shows how to calculate a conventional heatmap from Sparse IQ data

11.3.1 API Sequence Diagram

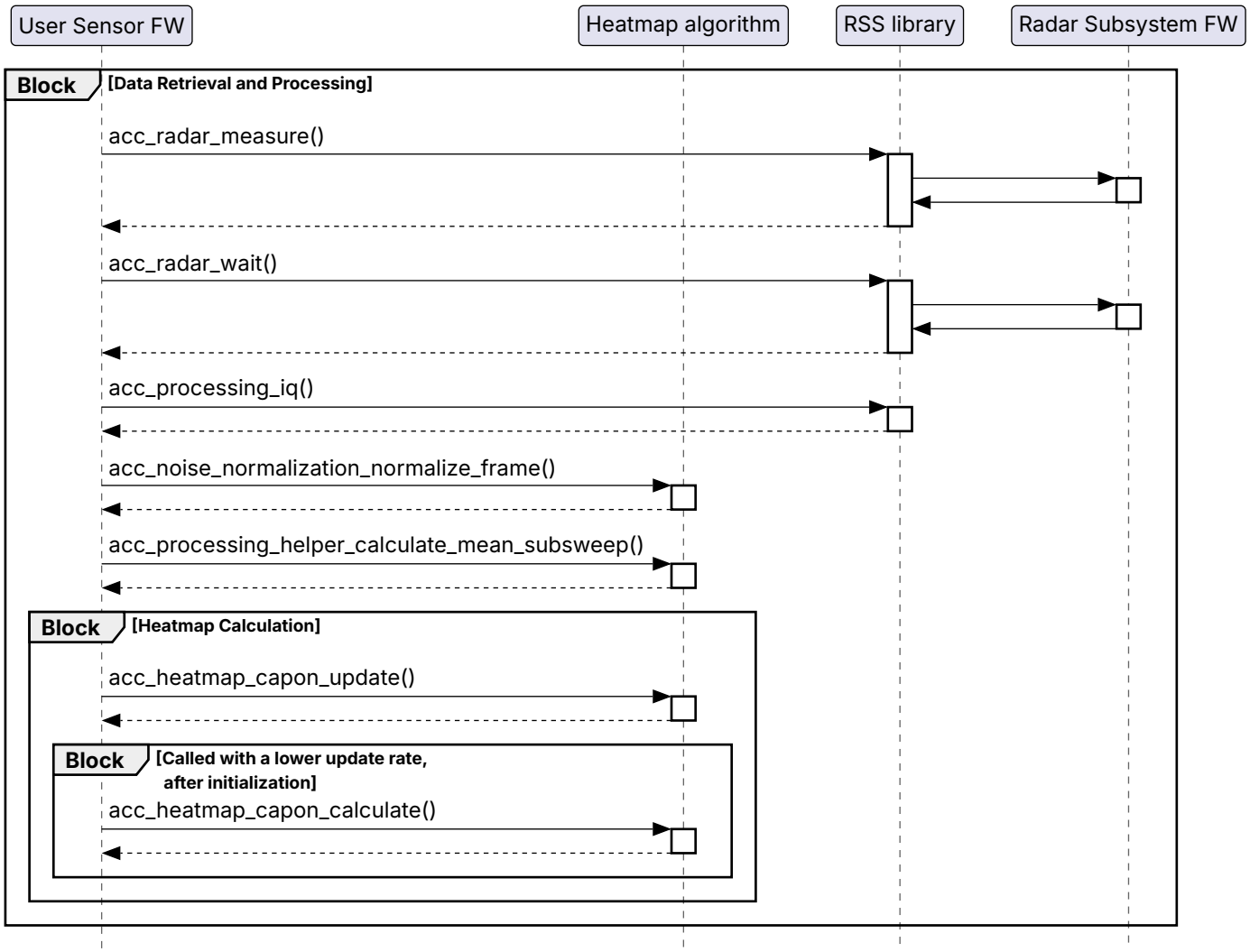


Figure 12: Sequence diagram for how to measure and process radar data.

11.3.2 Create heatmap configuration and allocate handle

The heatmap configuration handle needs to be created and is used to configure the heatmap processing.

```
acc_heatmap_capon_config_t *heatmap_config = acc_heatmap_capon_config_create();

if (heatmap_config == NULL)
{
    /* Handle error */
}
```

© 2026 by Acconeer AB - All rights reserved

Together with an `acc_config_t config`, the needed resources can be calculated and allocated. The persistent state buffer holds the heatmap state between updates. The work buffer is used when a heatmap is calculated.

```
uint8_t heatmap_subsweep_idx = 0U;

uint32_t persistent_state_buffer_size = acc_heatmap_capon_get_persistent_size(config,
                                                                              heatmap_subsweep_idx);

void *heatmap_persistent_state_buffer = acc_memory_calloc(1, persistent_state_buffer_size);

if (heatmap_persistent_state_buffer == NULL)
{
    /* Handle error */
}

uint32_t heatmap_buffer_size = 0U;

acc_heatmap_capon_get_buffer_size(heatmap_config, config, heatmap_subsweep_idx, &heatmap_buffer_size);

void *heatmap_buffer = acc_memory_alloc(heatmap_buffer_size);

if (heatmap_buffer == NULL)
{
    /* Handle error */
}

if (!acc_heatmap_capon_init(heatmap_config, config, heatmap_subsweep_idx, heatmap_persistent_state_buffer))
{
    /* Handle error */
}
```

11.3.3 Steering vectors

Heatmap uses steering vectors to perform its beamforming. These need to be allocated and calculated. This can be done once at initialization. Steering vectors can be reused for heatmaps with the same heatmap configuration, profile and antenna masks. `acc_steering_vectors_get_length()` returns 0 if the number of angles is too large.

```
uint16_t len_steering_vectors =
    acc_steering_vectors_get_length(acc_config_num_channels_get(config),
                                    acc_heatmap_capon_config_num_az_angles_get(heatmap_config),
                                    acc_heatmap_capon_config_num_el_angles_get(heatmap_config));

if (len_steering_vectors == 0U)
{
    /* Handle error */
}

float complex *steering_vectors = acc_memory_alloc(len_steering_vectors * sizeof(*steering_vectors));

if (steering_vectors == NULL)
{
    /* Handle error */
}

if (!acc_steering_vectors_init(acc_heatmap_capon_config_package_get(heatmap_config),
                               acc_config_profile_get(config),
                               acc_config_tx_antenna_mask_get(config),
                               acc_config_rx_antenna_mask_get(config),
                               acc_heatmap_capon_config_num_az_angles_get(heatmap_config),
                               acc_heatmap_capon_config_az_angles_range_start_get(heatmap_config),
                               acc_heatmap_capon_config_az_angles_range_end_get(heatmap_config),
                               acc_heatmap_capon_config_num_el_angles_get(heatmap_config),
                               acc_heatmap_capon_config_el_angles_range_start_get(heatmap_config),
                               acc_heatmap_capon_config_el_angles_range_end_get(heatmap_config),
                               steering_vectors))
{
```

© 2026 by Acconeer AB - All rights reserved

```
    /* Handle error */
}
```

11.3.4 Noise normalization

The measured noise is used to normalize the data, so that the measurements are consistent across individual sensors and temperatures. Noise normalization is done in two separate steps.

The first step is to record and calculate noise for each channel using `acc_noise_normalization_measure_noise()`. The noise measurement uses the RSS buffer, which must be at least as large as both `acc_rss_get_buffer_size()` and `acc_noise_normalization_get_buffer_size()` report.

`acc_noise_normalization_measure_noise()` needs to be called after each call to `acc_radar_calibrate()` to update the noise estimate after a sensor calibration. It prepares the radar for its own measurement, so `acc_radar_prepare()` must be called after it.

```
acc_noise_normalization_t noise_context;
uint32_t                  noise_length = 0U;

acc_noise_normalization_get_length(config, &noise_length);
noise_context.noise_values = acc_memory_alloc(sizeof(*noise_context.noise_values) * noise_length);
if (noise_context.noise_values == NULL)
{
    /* Handle error */
}

if (!acc_noise_normalization_measure_noise(&noise_context,
                                           processing,
                                           config,
                                           &cal_result,
                                           ACC_NOISE_NORMALIZATION_CONFIG_DEFAULT,
                                           buffer,
                                           buffer_size))
{
    /* Handle error */
}
```

The second step is to compensate the calculated noise for temperature and apply the compensation to the frame using `acc_noise_normalization_normalize_frame()`.

```
acc_noise_normalization_normalize_frame(&noise_context,
                                        &proc_result,
                                        config,
                                        norm_frame_mean_sweep);
```

11.3.5 Measure and calculation

Heatmap calculation is done in two steps. The first step is to update the heatmap state every time new Sparse IQ data is available. The heatmap state must be updated more than `num_channels` times before the first heatmap can be calculated. This first step also includes normalizing the data frame with the noise (see Section 11.3.4).

```
acc_processing_iq_result_t proc_result;

acc_processing_iq(processing, config, measure_result, &proc_result);

acc_noise_normalization_normalize_frame(&noise_context,
                                        &proc_result,
                                        config,
                                        norm_frame_mean_sweep);

acc_processing_helper_calculate_mean_subsweep(config,
                                              norm_frame_mean_sweep,
                                              heatmap_subsweep_idx,
```

© 2026 by Acconeer AB - All rights reserved

```
                                            norm_frame_mean_sweep);

acc_heatmap_capon_update(heatmap_persistent_state_buffer, norm_frame_mean_sweep);
```

The second step is to produce a heatmap. It is possible to produce a heatmap on demand, but the output will only change when the state has been updated. Most use cases will not require producing a heatmap for every update. Instead, this would be called less frequently, e.g. for every fifth update. If the heatmap calculations haven't been initialized or if the internal covariance matrix is singular and non-invertible, `acc_heatmap_capon_calculate()` will return `false`. When this happens the heatmap algorithm needs to be updated with more data to be able to produce a heatmap. Normally, this should not happen as the noise from the sensor is enough to make the covariance matrix non-singular.

The heatmap is stored with the elevation angle index varying fastest, then the azimuth angle index, and the point index slowest.

```
acc_heatmap_capon_result_t heatmap_result;

if (!acc_heatmap_capon_calculate(heatmap_persistent_state_buffer,
                                 steering_vectors,
                                 len_steering_vectors,
                                 heatmap_buffer,
                                 &heatmap_result))
{
    /* Handle error */
}

uint16_t num_points    = acc_config_num_points_get(config);
uint16_t num_az_angles = acc_heatmap_capon_config_num_az_angles_get(heatmap_config);
uint16_t num_el_angles = acc_heatmap_capon_config_num_el_angles_get(heatmap_config);

// Check if a heatmap has been produced and print the heatmap
if (heatmap_result.heatmap_ready)
{
    for (uint16_t point = 0; point < num_points; point++)
    {
        printf("Heatmap for point %" PRIu16 ":\n", point);
        for (uint16_t el_idx = 0; el_idx < num_el_angles; el_idx++)
        {
            printf("El angle index %" PRIu16 ": ", el_idx);
            for (uint16_t az_idx = 0; az_idx < num_az_angles; az_idx++)
            {
                uint32_t index = ((uint32_t)point * num_el_angles * num_az_angles) + ((uint32_t)az_idx *
num_el_angles) + el_idx;
                printf("%f, ", (double)heatmap_result.heatmap[index]);
            }

            printf("\n");
        }

        printf("\n");
    }
}
```

© 2026 by Acconeer AB - All rights reserved

12 System and Drivers

In addition to the RSS library, the sensor FW also needs to use system and peripheral drivers. This can for example be to set up clocks properly or to use the SPI block to communicate out from the sensor. The RSS library itself also needs a subset of the system drivers.

Figure 13 shows an overview of the dependencies.

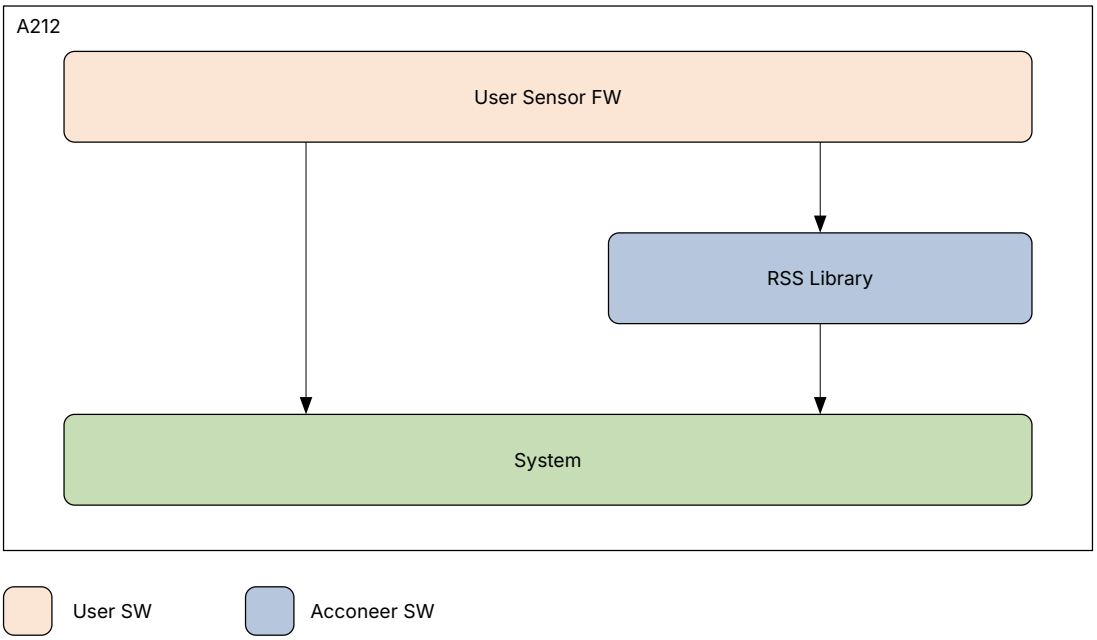


Figure 13: Sensor FW Architecture.

The *System* block in the figure above is hardware specific code and includes both system and peripheral drivers. It consists of two main parts:

- System and peripheral HAL/driver code
- Header files to interface them from *User Sensor FW* and *RSS library*

See Figure 14 below for a more detailed image of the system block from Figure 13.

© 2026 by Acconeer AB - All rights reserved

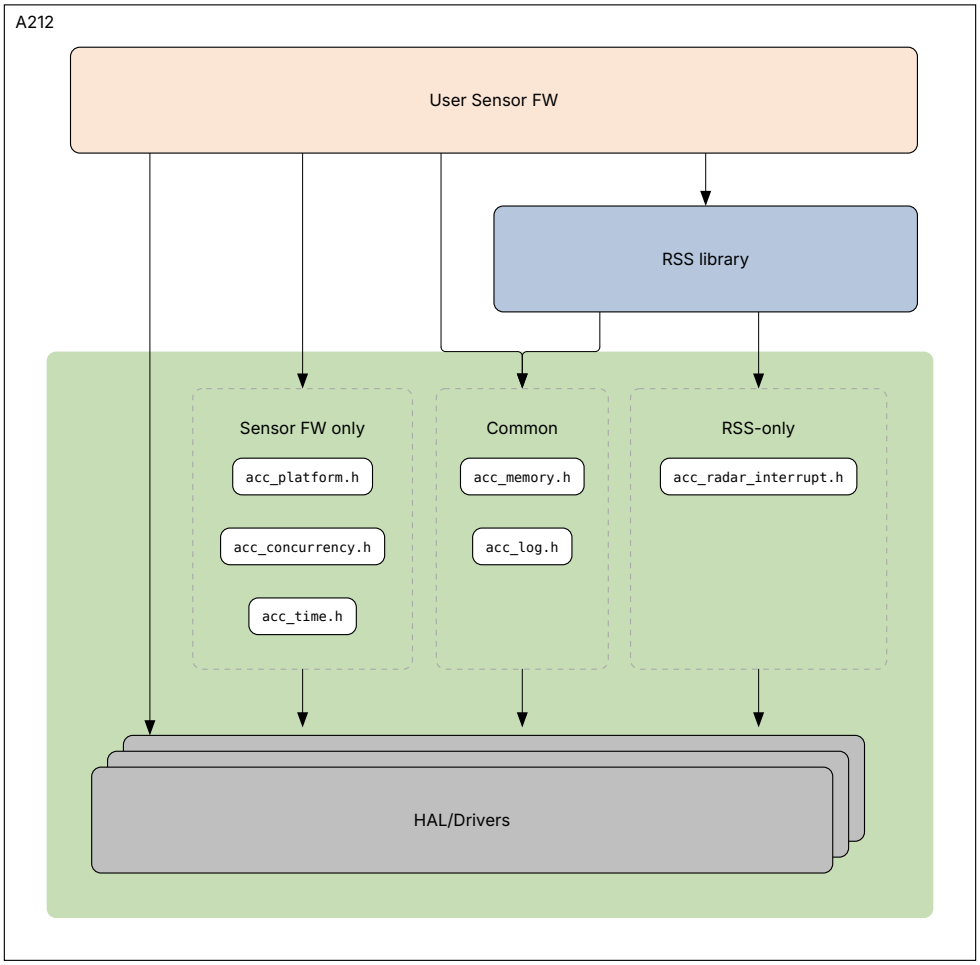


Figure 14: System overview with a closer look at the **System** part

`acc_memory.h` contains `<stdlib.h>`-like `malloc/free` functions, `acc_log.h` defines the `ACC_LOG_*` logging macros, etc.

Standard implementations of these modules are part of the SDK and they are open for modification.

Be careful when modifying the implementations, especially the RSS library dependencies (the outbound arrows from the RSS library in Figure 14). The reason is that the modifications might change the RSS library behavior in unwanted ways.

12.1 RTC

The RTC, or RC-OSC, has a frequency of 32kHz. It is the clock mainly used in the different sleep states. See `example_system_rtc_sleep_shallow_wakeup.c` for how to set up and use it.

The RTC can have a frequency error of up to +/-20%. The error is dependent on temperature. To reduce the frequency error to nominally <+/-5% and worst case +/-10%, the trimming function `aps_clock_control_auto_trim_rc_osc()` can be used. The function declaration and implementation can be found in `aps_clock_control.h` and `aps_clock_control.c`.

- The function takes around 8ms to execute and is performed by hardware
- The XTAL needs to be enabled when running the function
- The function can be executed in parallel with radar measurements

12.2 APS-PLL

The APS-PLL is a block providing a clock to the processor within the sensor. The frequency of the clock can be configured, but is 286 MHz by default.

There is a small risk that the APS-PLL unlocks during operation. An unlocked APS-PLL introduces a small (+/-0.7%) frequency error to the processor clock. Measurements have shown no negative impact on operation when this happens.

© 2026 by Acconeer AB - All rights reserved

The sensor is, by default, configured to reset if an unlock happens. To make the sensor not reset due to unlock, the `aps_system_disable_reset_on_aps_pll_error()` function can be called during application initialization. The function is implemented in `aps_system.h`.

To be able to take action when the APS-PLL unlocks, the code snippet below can be used to read the APS-PLL lock counter. The function declaration and implementation can be found in `aps_clock_control.h` and `aps_clock_control.c`.

```
// Read unlock counter
uint8_t unlock_count = aps_clock_control_read_and_clear_pll_unlock_count();
```

12.3 Interrupt Handlers

12.3.1 Avoid Using Floating-Point Operations in Interrupt Contexts

Floating-point operations should not be used within interrupt service routines (ISRs). The processor does not automatically preserve the floating-point unit (FPU) state during interrupts. If floating-point instructions are executed in an ISR, the FPU registers may be corrupted, leading to unpredictable behavior or data corruption elsewhere in the application.

In addition, floating-point operations are generally slower than integer operations and can significantly increase interrupt latency, negatively impacting real-time performance.

Recommendation: If floating-point calculations are required, capture the necessary data in the ISR and perform the computation later in the main loop or in a background task.

© 2026 by Acconeer AB - All rights reserved

13 Device Monitoring

13.1 Diagnostics

RSS can collect diagnostics of a sensor, which can be used for troubleshooting. See Section 13.1.2 for an example.

13.1.1 Performed Tests

13.1.1.1 Power On

A simple test that makes sure the RS can be powered on. Its only purpose is to catch this error.

13.1.1.2 LDO

Measures the output-voltage distribution of LDOs when adjusted with a default adjust value.

The following LDOs are measured:

- `AON_LDO`
- `APS_LDO`
- `RET2_LDO`
- `RET3_LDO`
- `RS_LDO`
- `RTM_LDO`
- `ADC_LDO`
- `WG_RX_1V2_LDO`
- `WG_RX_1V5_LDO`
- `RS_PLL_CLK_LDO`
- `RS_PLL_VCO_LDO`
- `RX_RF_LDO`
- `RX_RF_LO_LDO`
- `WG_TX_1V2_LDO`
- `WG_TX_1V5_LDO`
- `RX_1_LDO`
- `RX_2_LDO`
- `RX_3_LDO`
- `RX_DIG_LDO`
- `TX_1V2_LDO`
- `TX_1V5_LDO`

13.1.1.3 Calibration

Performs the sensor calibration (see Section 8).

13.1.1.4 SNR

Measures direct leakage and noise in order to estimate *signal-to-noise ratio* (SNR).

The test does a *signal measurement* of the direct leakage and a *noise measurement* with TX off. Both measurements are done with fixed HWAAS.

The *signal-* and *noise measurements* are processed into a *signal component* and a *noise component*, which are finally combined into the normalized metric *SNR per HWAAS*.

The reported value can give insight into whether there is an issue with the antennas and/or the TX and RX chains.

13.1.2 Diagnostics Report

`example_diagnostics.c` runs the tests in Section 13.1.1 and prints the `acc_diagnostics_t` struct in a machine-readable but human-friendly format:

```
00:00:08.639 (I) (diagnostics) DIAGNOSTICS_REPORT START
00:00:08.643 (I) (diagnostics) package: AP212
```

© 2026 by Acconeer AB - All rights reserved

```
00:00:08.647 (I) (diagnostics) power_on_test.status: RUN
00:00:08.652 (I) (diagnostics) ldo.status: RUN
00:00:08.656 (I) (diagnostics) ldo.aon.mean: 1125.160034 (mV)
00:00:08.662 (I) (diagnostics) ldo.aon.stddev: 1.172604 (mV)
                               ...
00:00:08.879 (I) (diagnostics) ldo.tx_1v5.mean: 1485.199951 (mV)
00:00:08.885 (I) (diagnostics) ldo.tx_1v5.stddev: 0.707107 (mV)
00:00:08.890 (I) (diagnostics) calibration.status: RUN
00:00:08.895 (I) (diagnostics) snr.status: RUN
00:00:08.899 (I) (diagnostics) snr.signal[ch_nbr=0]: 88.885559 (dB)
                               ...
00:00:08.964 (I) (diagnostics) snr.signal[ch_nbr=11]: 88.936988 (dB)
00:00:08.970 (I) (diagnostics) snr.noise[ch_nbr=0]: 12.251677 (dB)
                               ...
00:00:09.034 (I) (diagnostics) snr.noise[ch_nbr=11]: 13.792186 (dB)
00:00:09.040 (I) (diagnostics) snr.snr_per_hwaas[ch_nbr=0]: 55.561782 (dB)
                               ...
00:00:09.111 (I) (diagnostics) snr.snr_per_hwaas[ch_nbr=11]: 54.072708 (dB)
00:00:09.118 (I) (diagnostics) DIAGNOSTICS_REPORT END
```

13.2 Self test

In addition to the diagnostic functionality presented in Section 13.1, there is self-test functionality designed for in-field testing and functional safety. For details of how to use it, see `example_self_test.c` provided in the SDK.

`acc_self_test.h` defines an API for doing selected self tests. The different tests are described below.

13.2.1 ADC verification

Assesses the internal ADC's function by measuring the output voltages of the internal LDOs and comparing them to reference levels stored in EFUSE.

The function `acc_self_test_adc_test()` takes an input parameter `test_ldos` and an output parameter `failed_ldos` of type `acc_self_test_ldo_mask_t` (a `uint32_t` bitmask). Bit N in `test_ldos` indicates whether the LDO at index N should be included in the test. If bit N is not set, the function skips testing that LDO. Bit N in `failed_ldos` is set if the LDO at index N failed.

The LDO order, from bit 0 to bit 24, is: rtm, lvds, test, aon_dig, aps_dig, dly, dly_pi, pll_vclk, ret2, ret3, rs_dig, rx_dig, rx_1, rx_2, rx_3, rx_rf_lo, rx_rf, tx_1v2, tx_1v5, pll_vvco, wr_1v2, wr_1v5, wt_1v2, wt_1v5, xo.

On a passing result `failed_ldos` is zero. On failure, one or more bits are set to indicate which specific LDOs are out of bounds, allowing targeted diagnostics. If the function itself cannot run (execution failed), `failed_ldos` is set to zero.

13.2.1.1 Prerequisites

`acc_self_test_adc_test()` must not be called while a radar measurement is in progress, specifically not between `acc_radar_measure()` and the corresponding `acc_radar_wait()` / `acc_radar_poll()` + `acc_radar_get()`. The radar must have been powered on via `acc_radar_power_on()` before calling this function. `acc_self_test_factory_calibration_checksum()` must have passed, as a corrupt factory calibration can affect the EFUSE reference levels used to evaluate LDO voltages. No specific sensor configuration is required. The function manages all necessary internal state transitions.

13.2.1.2 Operational impact

- **Power consumption**: average current draw during test execution is approximately 130 mA.
- **Timing**: the call is blocking and takes up to 400 μs to complete all LDO tests (bit 0 to 24 set in `test_ldos`). The time it takes depends on the current RS power state when `acc_self_test_adc_test()` is called.
  - Executing `acc_self_test_adc_test()` when RS power state is `READY` takes approximately 400 μs.
  - Executing `acc_self_test_adc_test()` when RS power state is `ARMED` takes approximately 300 μs.
- **Memory**: no heap memory is allocated, and stack usage is limited to local variables inside the function.

13.2.1.3 Postcondition

After the function returns:

© 2026 by Acconeer AB - All rights reserved

- The RS power state is restored to what it was at the time of the call.
- `failed_ldos` is zero if `ACC_SELF_TEST_STATUS_EXECUTION_FAILED` or `ACC_SELF_TEST_STATUS_PASS` is returned,
- `failed_ldos` contains a set bitmask of failed LDOs if `ACC_SELF_TEST_STATUS_FAIL` is returned,
- No persistent register state is left modified by the function.

13.2.1.4 Test workflow

The recommended integration point within a standard measurement cycle is after `acc_radar_wait()` returns and before the next call to `acc_radar_measure()`:

1. Call `acc_radar_power_on()` and `acc_radar_calibrate()`.
2. Call `acc_radar_prepare()`.
3. Start measurement with `acc_radar_measure()`.
4. Wait for result with `acc_radar_wait()`.
5. *Safe window*: call `acc_self_test_adc_test()` here, between the completed wait and the next measure.
6. Repeat from `acc_radar_measure()` step.

The workflow above illustrates the call placement, not the recommended frequency. Running the test on every measurement loop is unnecessary. A periodic schedule, such as once per day or at power-up, is sufficient for most applications.

See `example_self_test.c` in the SDK for a complete integration example.

13.2.1.5 Test results analysis

The function returns an `acc_self_test_status_t` value and populates `failed_ldos`:

Return value	failed_ldos	Meaning
`ACC_SELF_TEST_STATUS_PASS`	0	All LDOs within EFUSE reference bounds
`ACC_SELF_TEST_STATUS_FAIL`	One or more bits set	Specific LDOs out of bounds, see `ACC_SELF_TEST_LDO_MASK_*`
`ACC_SELF_TEST_STATUS_EXECUTION_FAILED`	0	Test could not run, LDO state unknown

Table 6: ADC test result interpretation

Each set bit in `failed_ldos` corresponds to one `ACC_SELF_TEST_LDO_MASK_*` value in `acc_self_test.h`, identifying the affected LDO by name (e.g. bit 17 corresponds to `ACC_SELF_TEST_LDO_MASK_TX_1V2`).

The built-in pass/fail threshold is based on EFUSE reference levels programmed at the factory. Applications may apply additional policy, such as triggering an alert after a number of consecutive failures or treating individual LDO domains differently based on their impact on the specific application.

On `ACC_SELF_TEST_STATUS_FAIL`, radar data may be unreliable. The application should stop using radar data until the condition is resolved. A power cycle (`acc_radar_power_off()` followed by `acc_radar_power_on()`) is the recommended first recovery step. If failures persist across power cycles, the hardware should be considered faulty.

On `ACC_SELF_TEST_STATUS_EXECUTION_FAILED`, the test itself could not run and LDO health is unknown. The recommended action is to power cycle the radar and retry. If `ACC_SELF_TEST_STATUS_EXECUTION_FAILED` recurs, and all prerequisites are met, the hardware should be considered faulty.

13.2.2 Factory calibration verification

The radar sensor is factory-calibrated, and the calibration data is stored in One-Time Programmable (OTP) memory. The self-test API provides a function that verifies the integrity of this memory by checking its CRC. It is the responsibility of the application to handle the case where the verification fails.

If the CRC is not valid, the radar will still continue to work and use default values instead of the factory calibration, which might impact the radar's performance.

© 2026 by Acconeer AB - All rights reserved

14 Memory

14.1 Non-volatile Memory

The A212 does not have any embedded non-volatile memory such as Flash or EEPROM. Sensor FW can be loaded to the A212 in two different ways:

- From an external NVM via SPI or QSPI. See *A212 NVM Image Format Application Note*.
- From an external MCU via SPI, QSPI or UART. See *A212 DFU Protocol Application Note*.

14.2 Volatile Memory

The A212 Application Subsystem (APS) has 288 KB of instruction memory (ILM) and 256 KB of data memory (DLM). Both the ILM and DLM in the APS are shared between the RSS library and *User Sensor FW*. The A212 Radar Subsystem (RS) is completely handled by Acconeer, and its memory is not available to the application. See Figure 3 for an overview of the different blocks described here.

14.3 ILM Usage

Table 7 shows the ILM usage of some of the sensor FWs provided in the SDK.

Sensor FW	Size (KB)
`example_service_iq`	80
`example_heatmap_capon`	108
`acc_exploration_server_a2`	125

Table 7: ILM usage

14.4 DLM Usage

The DLM can be divided into three categories, static RAM, heap, and stack. Table 8 shows the approximate DLM usage of some of the sensor FWs provided in the SDK.

Sensor FW	Static (KB)	Heap (KB)	Stack (KB)	Total (KB)
`example_service_iq`	1	1	3	5
`example_heatmap_capon`	2	39	3	44

Table 8: DLM usage

The heap usage depends on the configuration and the amount of processing, but the numbers in Table 8 are a good estimate for many configurations.

The configurations that have the largest impact on memory are `sweeps_per_frame`, `num_points`, `step_length`, and the use of multiple TX and/or RX antennas via the `tx_antenna_mask`/`rx_antenna_mask` settings.

© 2026 by Acconeer AB - All rights reserved

15 Change Log

Date	Change
20261026	Restructured the guide, added related documents and a glossary, and corrected the code samples.
20260825	Added the coherent heatmap.
20260806	Added the Radar Frame Data Format chapter.
20260722	Added the 12.5 MHz and 15.8 MHz PRFs.
20260422	Extended the ADC verification description.
20260318	Moved the DFU protocol to the A212 DFU Protocol Application Note.

Table 9: Historical Change Log

© 2026 by Acconeer AB - All rights reserved

16 Disclaimer

The information herein is believed to be correct as of the date issued. Acconeer AB (“Acconeer”) will not be responsible for damages of any nature resulting from the use or reliance upon the information contained herein. Acconeer makes no warranties, expressed or implied, of merchantability or fitness for a particular purpose or course of performance or usage of trade. Therefore, it is the user’s responsibility to thoroughly test the product in their particular application to determine its performance, efficacy and safety. Users should obtain the latest relevant information before placing orders

Unless Acconeer has explicitly designated an individual Acconeer product as meeting the requirement of a particular industry standard, Acconeer is not responsible for any failure to meet such industry standard requirements.

Unless explicitly stated herein this document Acconeer has not performed any regulatory conformity test. It is the user’s responsibility to assure that necessary regulatory conditions are met and approvals have been obtained when using the product. Regardless of whether the product has passed any conformity test, this document does not constitute any regulatory approval of the user’s product or application using Acconeer’s product.

Nothing contained herein is to be considered as permission or a recommendation to infringe any patent or any other intellectual property right. No license, express or implied, to any intellectual property right is granted by Acconeer herein.

Acconeer reserves the right to at any time correct, change, amend, enhance, modify, and improve this document and/or Acconeer products without notice.

This document supersedes and replaces all information supplied prior to the publication hereof.



© 2026 by Acconeer AB - All rights reserved