A212 Sensor FW

User Guide

A212 Sensor FW

User Guide

Author: Acconeer AB

Version: a2-v0.15.1

Acconeer AB Mar 08, 2025

Contents

Acconeer SDK Documentation Overview

To better understand what SDK document to use, a summary of the documents are shown in the table below. More good information can also 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
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.

Early draft

This is an early version of the document and parts are subject to change. Some of the described features might not be available yet.

1 Introduction

There are two main ways to develop software for the A212 sensor. By using

- *Acconeer Sensor FW* or
- *User Sensor FW*.

The following sections give an overview of both ways, but the main focus of this User Guide will be *User Sensor FW*.

1.1 Acconeer Sensor FW

When developing using *Acconeer Sensor FW*, the software running inside the A212 sensor is provided by Acconeer. The protocol used to communicate with the software inside the A212 is clearly documented.

This means that software that can communicate with the A212 can be developed on a host of your choice. Example host code is usually provided by Acconeer.

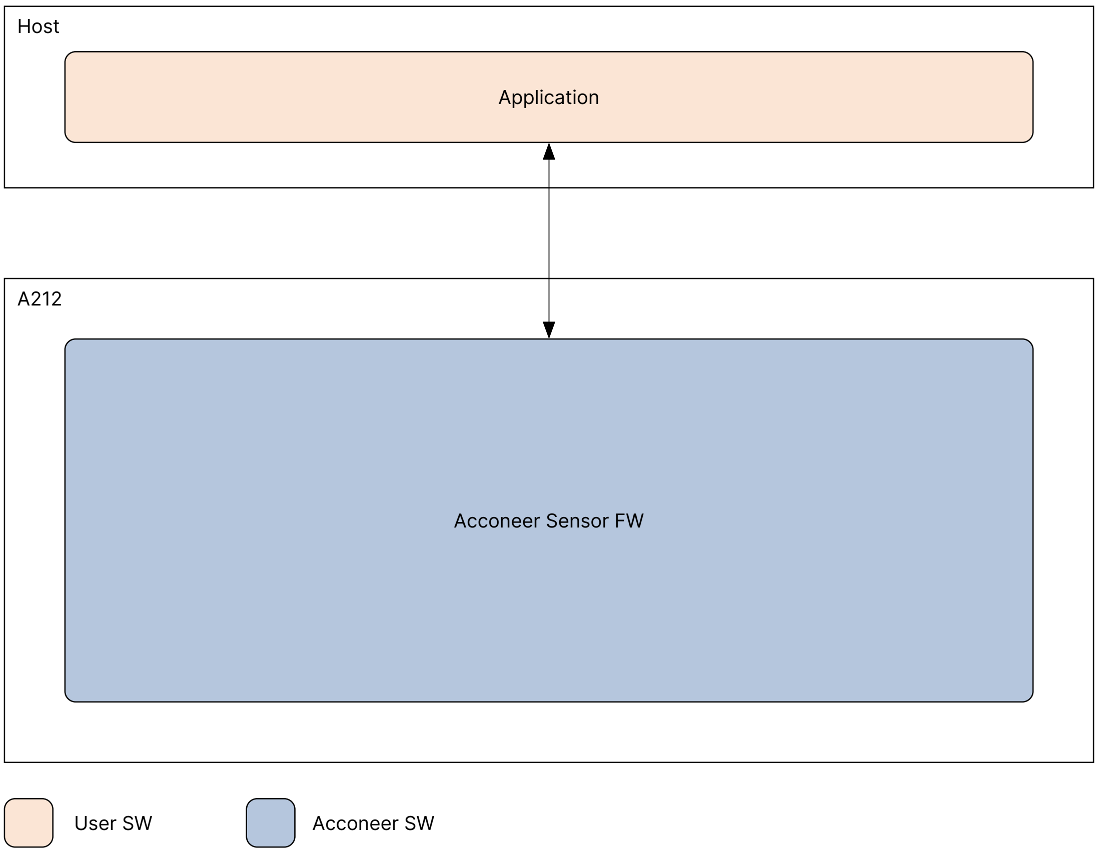


Figure 1: Acconeer Sensor FW Overview.

As can be seen in the figure above, all SW running in the A2 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.

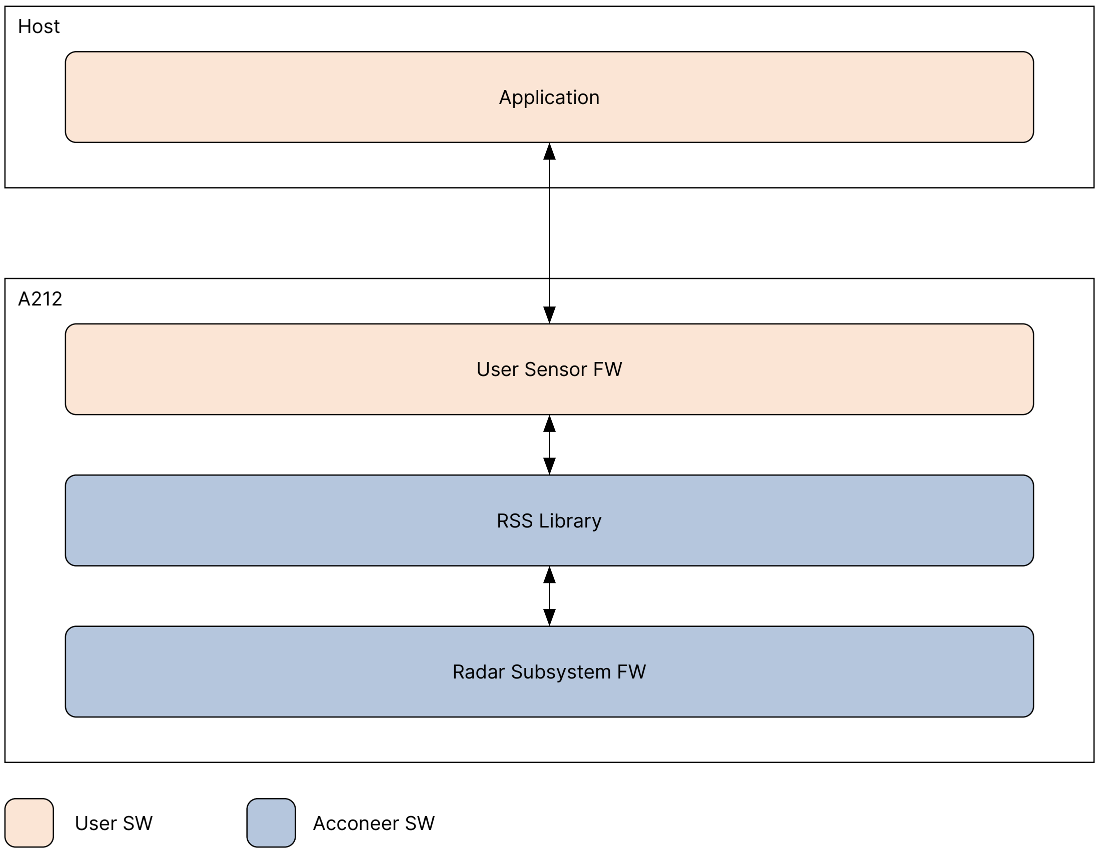


Figure 2: User Sensor FW Overview.

As can be seen in the figure above, the SW running in the A2 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 Section 3 for more information.

It is recommended to use this Guide together with

- `example_service_iq.c` Example of a *User Sensor FW* that uses the *IQ Service*
- `rss_api.html` The full RSS API specification

These are provided in the SDK package.

2 Radar Data Overview

The measured radar data can be processed into two types.

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

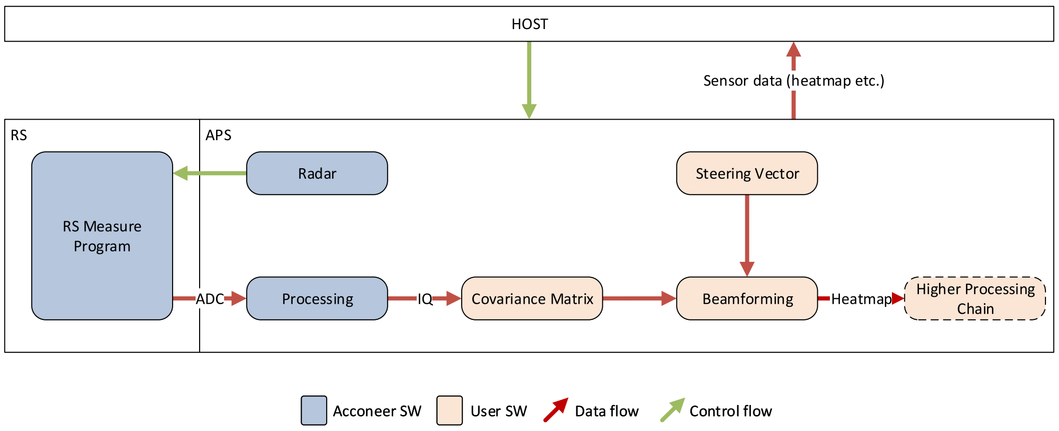


Figure 3: Radar Data Overview.

A measurement is configured and controlled by the *Radar* module (more info in Section 3). 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 8). 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 how the radar data looks like and how it can be used.

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

For more information on Exploration Tool, see docs.acconeer.com.

3 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.

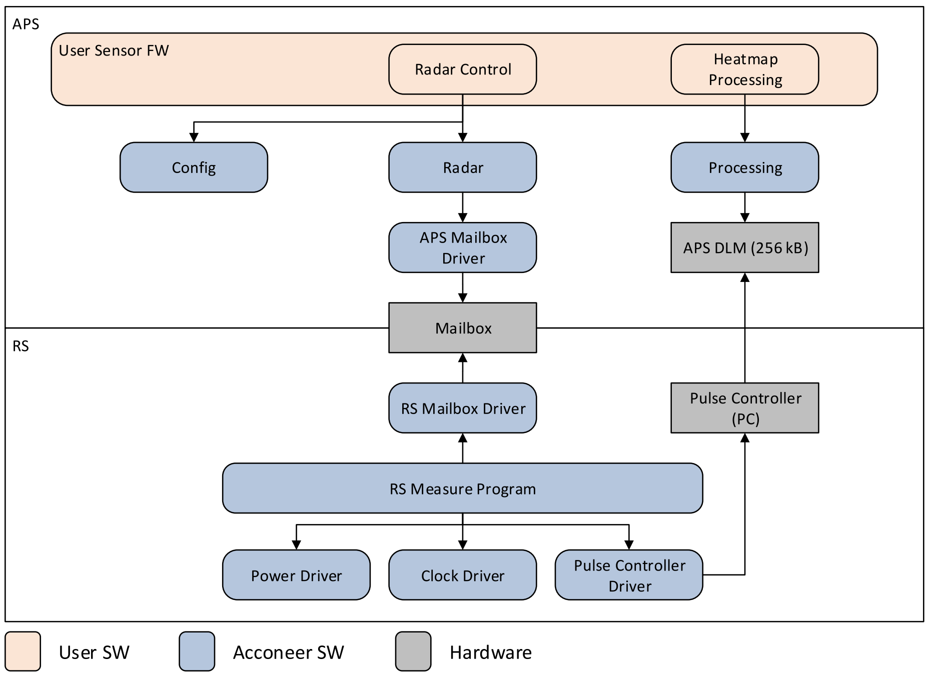


Figure 4: Sensor FW Architecture.

The A2 sensor is divided into two main parts, the *Application Subsystem* (APS) and the *Radar Subsystem* (RS).

In the APS, the customer can implement their own SW to fulfill their use case (*User SW* in image above). The customer SW can interact with the RS by using the RSS library (*Acconeer SW* in image above). The RSS library consists of three API modules, see section Section 3.1.

3.1 API SW Modules

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

3.1.1 Config

SW module used to configure the radar.

The following operations are available in the module:

- Set configurations
- Get configurations

3.1.2 Radar

SW module used to communicate with the Radar Subsystem (RS) FW. The communication is done via a hardware block called *Mailbox*, see figure Figure 4.

The following operations are available in the module:

- Power On/Off the RS
- 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.

3.1.3 Processing

SW module used to process raw data from a radar measurement into *Service* data.

The following operations are available in the module:

- Convert unreadable radar measurement data to indications
- Process unreadable radar measurement data to IQ data
- Process IQ data to Heatmap data

3.2 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 Figure 4. 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*.

3.2.1 Creating and preparing

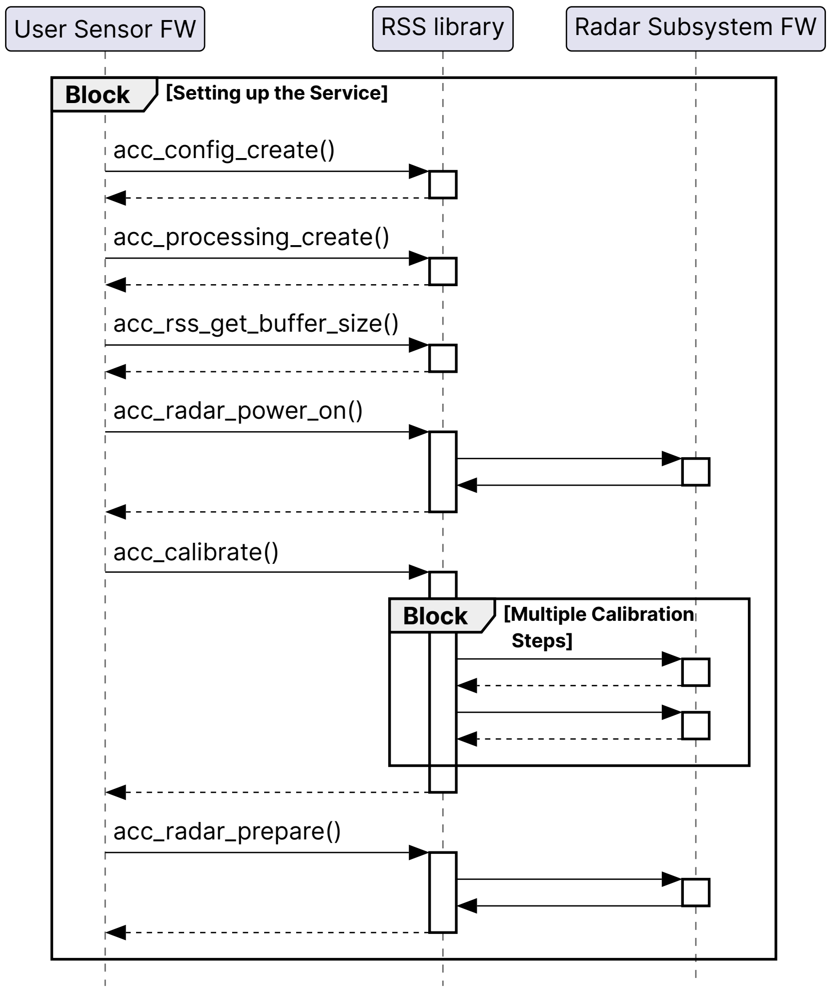


Figure 5: Sequence diagram for how to setup and prepare a radar measurement.

3.2.2 Measure and processing

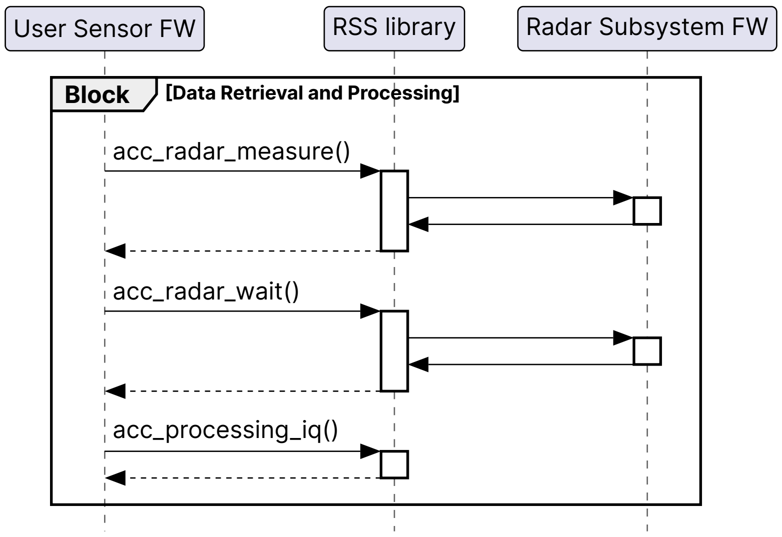


Figure 6: Sequence diagram for how to measure and process radar data.

3.2.3 Destroying

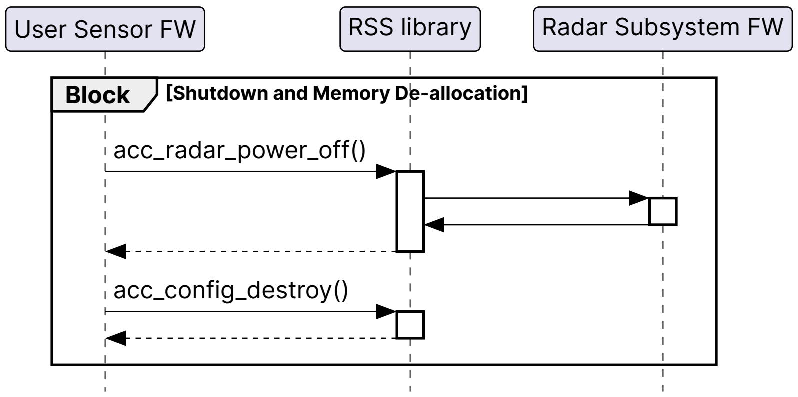


Figure 7: Sequence diagram for how to measure and process radar data.

3.3 Setting up the Service

Below is a step by step guide of how to setup and configure the Service.

3.3.1 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);
```

3.3.2 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.

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

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

3.3.3 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 using the `malloc()` function or with a declared buffer.

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

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

3.3.4 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 */
}
```

3.3.5 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))
{
    /* Handle error */
}
```

3.3.6 Prepare Radar

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

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

3.3.7 Read Radar Temperature

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

The function must not be called before `acc_radar_power_on()` and `acc_radar_prepare()` has 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 */
}
```

3.4 Data Retrieval and Processing

3.4.1 Measure and Read Data from Radar

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 `acc_radar_wait()` has returned.

```
acc_measure_result_t *measure_result;

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

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

3.4.2 Sparse IQ Processing

To make sense of the raw data, it must be processed with `acc_processing_iq()`. This converts the raw data 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);
        }
    }
}
```

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

3.5 Shutdown and Memory De-allocation

3.5.1 Powering off Radar

The Radar can be powered off when not used anymore.

```
acc_radar_power_off();
```

3.5.2 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)
{
   free(buffer);
}
```

4 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.

4.1 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 4.2.

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

name	type	default	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_6
tx_antenna_mask	uint8_t	acc_tx_antenna_mask_0	n/a	n/a
rx_antenna_mask	uint8_t	acc_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_8_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	4
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

Table 2: RSS Configuration Parameters

4.2 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.

4.2.1 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 DLM1 in the Application Subsystem (APS). The size of this memory is 192KB.

Note, for A212 R1B, the memory used to store radar data is instead located in the Radar Subsystem (RS). The size of this memory is 8KB, but most of it is used to configure the RS. The available memory size left for radar data is $1008*4 = 4032$ bytes.

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:

- 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 radar data size is calculated as $(N_d * N_{ch} * 4) * N_s + N_a$, if $N_{ss} = 1$. N_a is a value up to 64 bytes and is needed due to internal alignment handling.

4.2.2 PRF

The Pulse Repetition Frequency (PRF) limits the maximum end range. The maximum end range is calculated as start_point + step_length * num_points.

This is not enforced yet and thus the exact limit is not known, but for a PRF of 20.8 MHz the limit is around 7.13m.

4.2.3 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 configures the radar to 'run as fast as possible'.

The maximum sweep rate must be low enough to have time to do sweeps_per_frame number of sweeps. See section Section 4.2.5 for more information.

4.2.4 Sweeps Per Frame

The number of sweeps per frame that can be used is limited by available memory. See section Section 4.2.1 for more details.

4.2.5 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

The exact sweep duration calculation is very complicated and dependent 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.

4.2.6 Number of Subsweeps

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

Note, for A212 R1B, num_subsweeps is limited to 1.

4.3 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);
```

4.4 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()` `acc_radar_process()`, 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`.

4.5 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.

4.5.1 Revision R1 Activation Conditions

HSM is activated under the following conditions on R1:

- `sweeps_per_frame > 1`
- `nbr_of_subsweeps == 1`
- `inter_sweep_idle_state == ARMED`

4.5.2 Revision R2 Activation Conditions

HSM is activated under the following conditions on R2:

- `sweeps_per_frame > 1`
- `nbr_of_subsweeps == 1`

5 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 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. 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.

5.1 Radar Re-calibration

Not implemented yet.

Sometimes, the sensor needs to be re-calibrated. This is indicated by the `calibration_needed` indication. It triggers if the temperature difference between the last successful calibration and the current measurement exceeds 15 degrees Celsius.

For more information, see section Section 6.

6 RSS Indication Handling

Not implemented yet.

The result from a call to `acc_processing_iq()` includes both the radar data frame as well as indications. Below can be seen 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.

6.1 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.

6.2 Temperature

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

7 RSS Error Handling

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

7.1 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. 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 7.3.

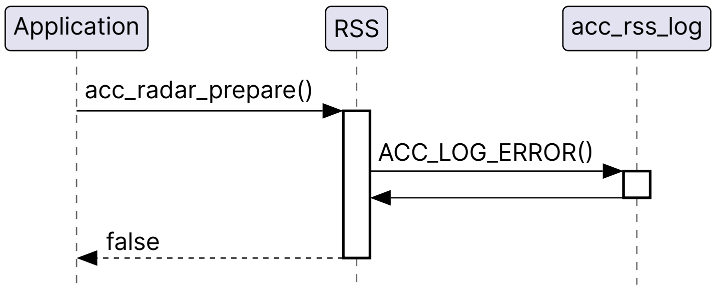


Figure 8: Example of the sequence where acc_radar_prepare() fails.

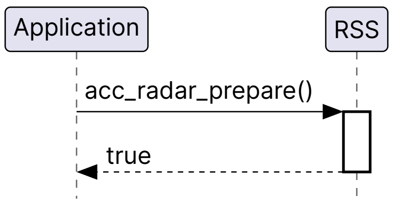


Figure 9: Example of the sequence where acc_radar_prepare() succeeds.

7.2 Bring up- & 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.

7.2.1 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 occured (where a function returns `false` and *what* was wrong (by reading the log).

7.2.2 In-field errors

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

If one of these errors would occur, the most simple mitigation is to restart the radar system. Some specific scenarios allow a smarter mitigation. Those are listed in Table 3:

Error description	**Mitigation**
Something unforseen went wrong	Restart the radar system
Sensor calibration fails	Retry calibration once more
X	Y

Table 3: Mitigation table. **Note: under construction**

7.3 Error Categories

7.3.1 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.

7.4 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.

7.4.1 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.

7.4.2 Configuration Errors

The configuration of the radar will be checked during `acc_rss_get_buffer_size()`, `acc_create_processing()` 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.

7.4.3 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 by calling `acc_radar_power_off()` followed by `acc_radar_power_on()` before the function `acc_radar_calibrate()` is called. If the calibration error persists, there might be an issue with the radar sensor or integration.

7.4.4 Timeout Errors

A timeout error should never occur in a fully functional radar sensor. The functions `acc_radar_power_on()`, `acc_radar_calibrate()`, `acc_radar_prepare()` 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 are called. If the timeout error persists, there might be an issue with the radar sensor or integration.

7.4.5 System Errors

A system error should never occur in a fully functional radar sensor. The functions `acc_radar_power_on()`, `acc_radar_calibrate()`, `acc_radar_prepare()` and `acc_radar_wait()` can potentially return `false` 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 are called. If the system error persists, there might be an issue with the radar sensor or integration.

8 Heatmap calculation

8.1 Measure and calculation

Heatmap calculation is done in two steps. The first step is to update heatmap state every time new Sparse IQ data is available. The heatmap must be updated `num_channels` number of times before it is initialized.

```
acc_processing_iq_result_t proc_result;

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

acc_processing_helper_calculate_mean_subsweep(config, &proc_result, subsweep_idnex, mean_subsweep)

acc_heatmap_capon_update(heatmap_handle, mean_subsweep);
```

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 to produce 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.

```
acc_heatmap_capon_result_t heatmap_result;

if (!acc_heatmap_capon_calculate(heatmap_handle, heatmap_buffer, &heatmap_result))
{
    printf("acc_heatmap_capon_calculate() failed\n");
    cleanup(...);
    return EXIT_FAILURE;
}

uint16_t num_points    = acc_heatmap_capon_config_num_points_get(heatmap_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++)
            {
                uint16_t index = point * num_el_angles + el_idx * num_az_angles + az_idx;
                printf("%f, ", (double)heatmap_result.heatmap[index]);
            }

            printf("\n");
        }

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

8.2 API Sequence Diagram

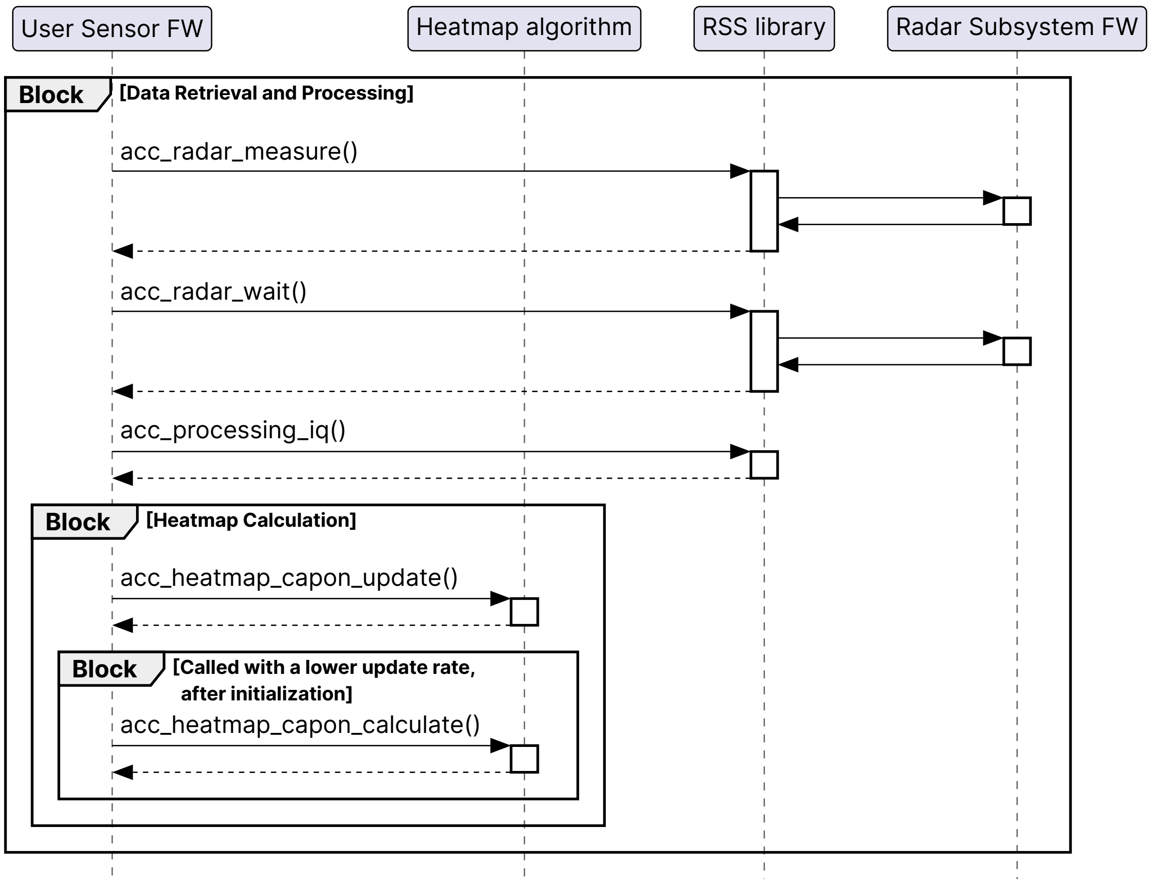

User Sensor FW
Heatmap algorithm
RSS library
Radar Subsystem FW
Block [Data Retrieval and Processing]
acc_radar_measure()
acc_radar_wait()
acc_processing_iq()
Block [Heatmap Calculation]
acc_heatmap_capon_update()
Block [Called with a lower update rate, after initialization]
acc_heatmap_capon_calculate()

Figure 10: Sequence diagram for how to measure and process radar data.

9 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 setup 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 11 shows an overview of the dependencies.

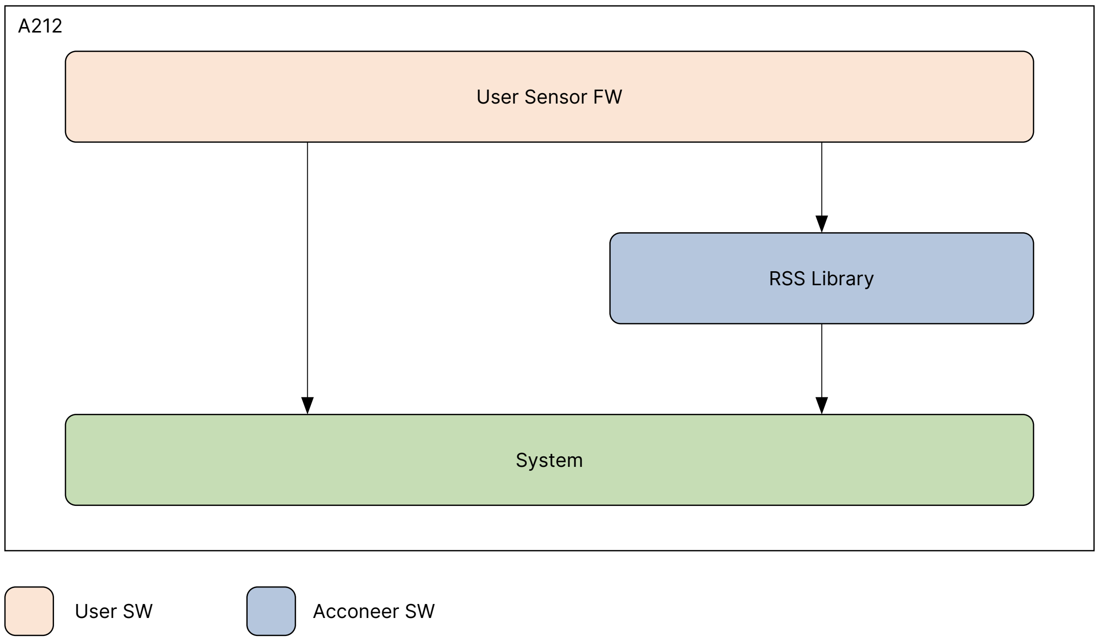


Figure 11: Sensor FW Architecture.

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 Figure 12 below for a more detailed image of the system block from figure Figure 11.

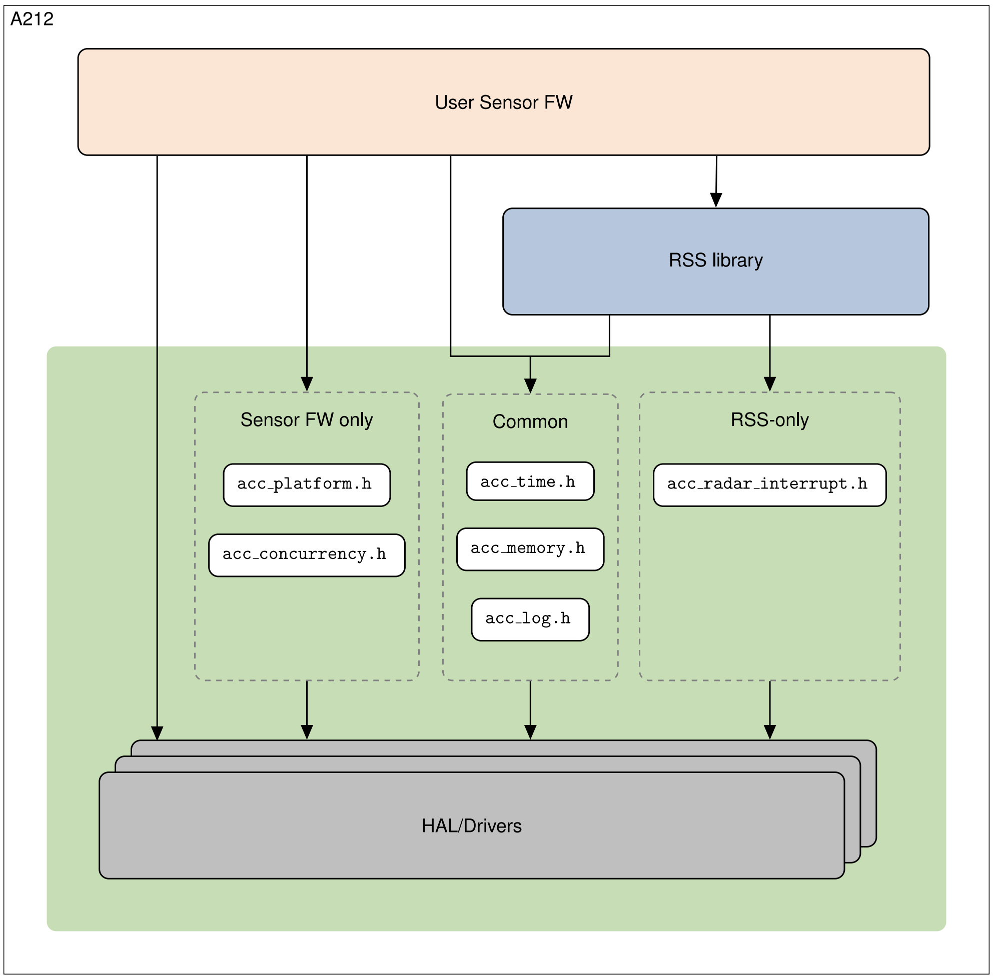


Figure 12: System overview with a closer look at the **System** part

`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 Figure 12). The reason is that the modifications might change the RSS library behavior in unwanted ways.

10 Diagnostics

There is functionality in RSS to get diagnostics of a sensor that can be for troubleshooting. (See example in Section 11.3)

10.1 Performed Tests

10.1.1 Power Toggle

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

10.1.2 LDO

Measures an average output voltage 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`

10.1.3 Calibration

Performs the sensor calibration (see Section 5).

10.1.4 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*.

In R1, the direct leakage saturates. Due to the saturated signal components, the calculated SNR might be lower than the actual SNR.

11 Examples

Multiple different examples are provided in the SDK, see table below.

11.1 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_heatmap_capon.c` - Shows how to calculate Heatmap from IQ data
- `example_screening_test.c` - A basic screening test to sort out bad sensors
- `example_diagnostics.c` - Shows how to extract diagnostics from the sensor

11.2 Advanced Control

These examples show advanced control concepts.

- `example_service_iq_double_buffering.c` - Shows usage of double buffering with the Sparse IQ Service

11.3 Diagnostics

Diagnostics is a self-check of the sensor that reports different metrics related to the sensor's health.

- `example_diagnostics.c` - Extracts diagnostics from the sensor and print the result via debug UART.

12 Memory

12.1 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 (Q)SPI
- From an external MCU via (Q)SPI or UART

For more information on this, see *link to document*.

12.2 Volatile Memory

The A212 Application Subsystem (APS) has 288kB of instruction memory (ILM) and 256kB 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 which means that the memory there is not important. Please see figure Figure 4 for an overview of the different blocks described here.

12.3 ILM Usage

Below is a table showing the ILM usage for a couple of sensor FWs provided in the SDK. To be compatible with multiple sensor revisions, the sensor FW might include multiple versions of the Radar Subsystem FW, each currently taking up around 20 kB of memory. The sensor FWs in the table below contain two versions of the Radar Subsystem FW. In the long term, only one sensor revision will be supported.

Sensor FW	Size (kB)
`example_service_iq`	74
`example_heatmap_capon`	87
`acc_exploration_server_a2`	119

12.4 DLM Usage

The DLM can be divided into three categories, static RAM, heap, and stack. Below is a table for approximate DLM for different applications.

Sensor FW	Static (kB)	Heap (kB)	Stack (kB)	Total (kB)
`example_service_iq`	1	1	3	5
`example_heatmap_capon`	3	39	3	45

Note that heap is very dependent on the configuration and the amount of processing. But the numbers in the table above is a pretty good estimate for many configurations.

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

13 Tools

13.1 GCC Toolchain

A GCC toolchain is needed to build applications for the A212 Application Subsystem (APS). 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 **v5.2.2**, and the APS applications are built with the compiler found in the **nds32le-elf-mculib-v5f.txz** archive.

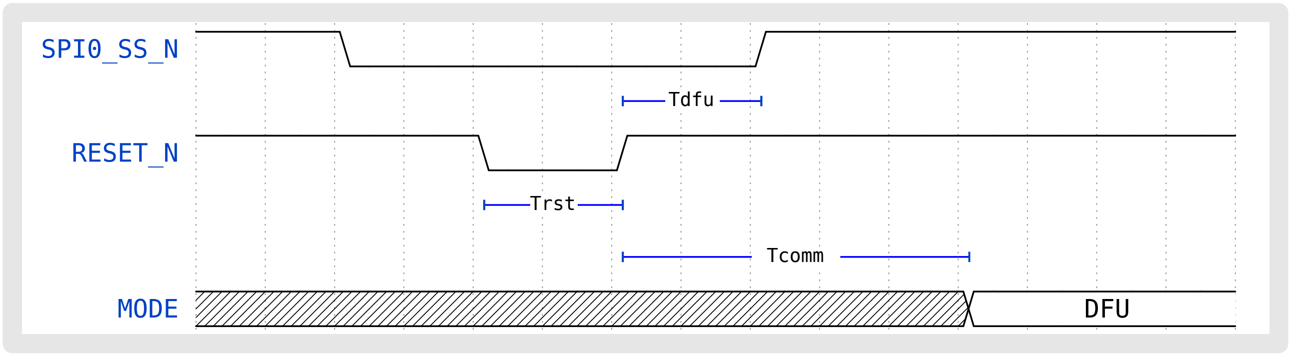
Below is an example of Linux compiler installation and build of the applications in the `sdk_r1` folder.

```
$> 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_r1
$> make
```

14 DFU - Device Firmware Upgrade

14.1 Enter DFU Mode

The A2 enters DFU mode if the SPI0_SS_N signal is held low during release of RESET_N.



Description	Symbol	Min	Max
DFU enter hold time of SPI0_SS_N	Trst	10us	
RESET hold time of RESET_N	Tdfu	1us	500000us
Communication ready time	Tcomm		10000us

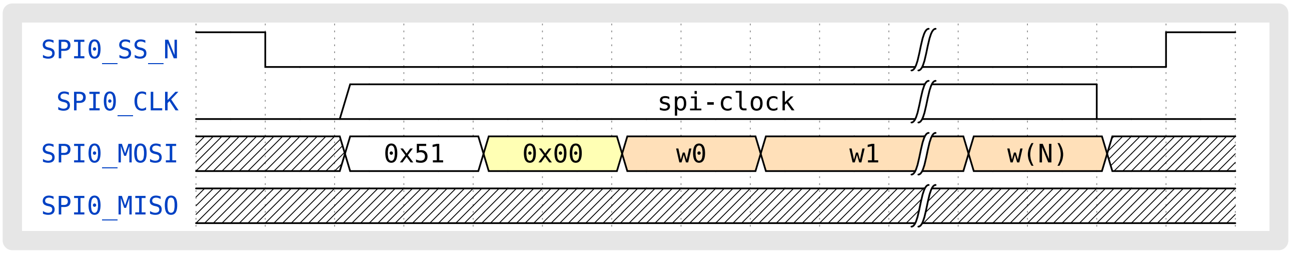
Timing 1: Timing Diagram of DFU mode startup

14.2 SPI Protocol

The A2 uses an SPI protocol that can perform *write data*, *read status* or *read data*.

14.2.1 SPI Write Data Transaction

Writing data to the A2 is done by sending the SPI command `0x51` followed by a single dummy byte from the Host. The A2 will thereafter consume data *w0, w1, ..., w(N)* as long as the SPI clock is running. The maximum data length for the DFU protocol is 2048 bytes.



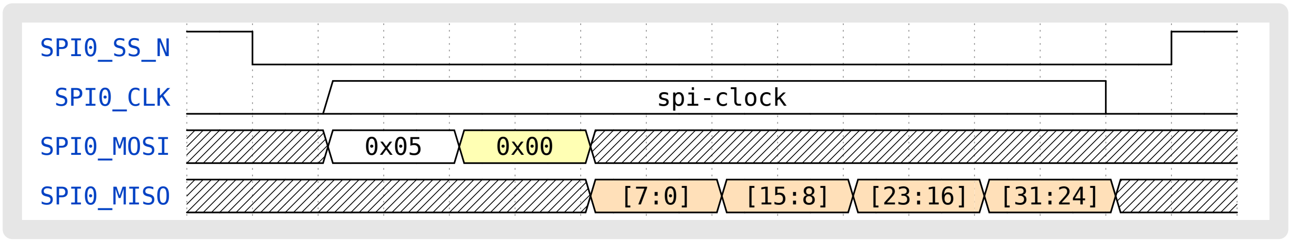
Timing 2: Timing Diagram of Data-Writing

14.2.2 SPI Read Status Register Transaction

Reading the status register from the A2 is done by sending the SPI command `0x05`, followed by a single dummy byte from the Host. The A2 will thereafter deliver the content of the status register, 32-bits.

Name	Bit(s)	Description
Unused	31:19	Unused
UnderRun	18	Data underrun occurred in the last transaction
OverRun	17	Data overrun occurred in the last transaction
Ready	16	This bit is set when the A2 SPI interface is ready for data transaction
Length	15:0	The number of bytes that should be read by the host

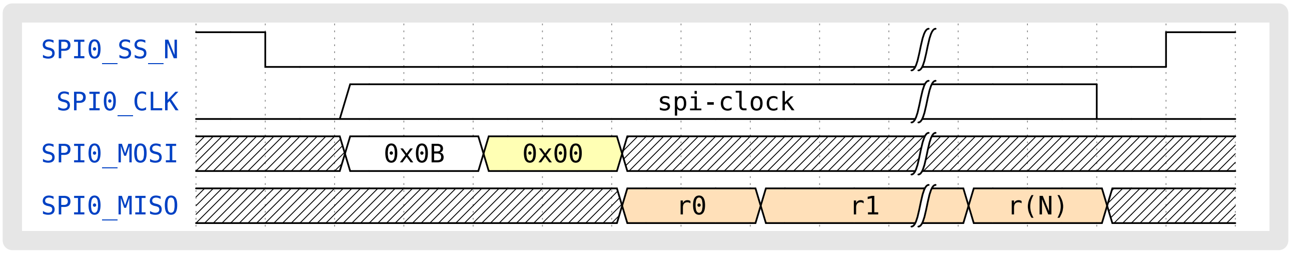
Table 4: Status Register Fields.



Timing 3: Timing Diagram of Status-Reading

14.2.3 SPI Read Data Transaction

Reading data from the A2 is done by sending the SPI command `0x0B`, followed by a single dummy byte from the Host. The A2 will thereafter deliver data *r0, r1, ..., r(N)* as long as the SPI clock is running. The maximum data length for the DFU protocol is 2048 bytes.



Timing 4: Timing Diagram of Data-Reading

14.3 DFU Protocol

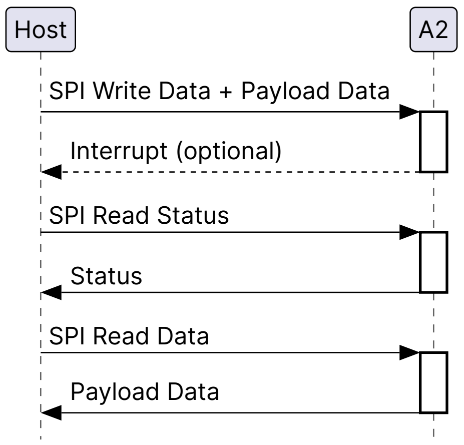


Figure 13: SPI Communication Sequence.

14.3.1 Communication Sequence without Interrupt

1. The host executes an SPI Write Data transaction with a request packet to the A2.
2. The A2 processes the packet and generates a response.
3. The host executes SPI Read Status Register transactions until the **Ready** field is set
4. The host gets the response packet length from Status Register **Length** field.
5. The host executes an SPI Read Data transaction to get the response packet from the A2.

14.3.2 Communication Sequence with Interrupt

1. The host executes an SPI Write Data transaction with a request packet to the A2.
2. The host waits for an interrupt signal from the A2.
3. The A2 processes the packet and generates a response.
4. The A2 sets the interrupt signal high.
5. The host is notified by the interrupt.
6. The host executes an SPI Read Status Register transaction to get the response packet length from the A2.
7. The host executes an SPI Read Data transaction to get the response packet from the A2.

14.3.3 Command Packet

A command packet is sent to the A2 with the SPI Write Data command.

Cmd	Length	Data	CRC

The CRC is calculated for each byte in the Cmd, Length and Data fields.

14.3.4 Response Packet

A response packet is read from the A2 with SPI Read Data command, the number of bytes to be read is acquired from the SPI Read Status command.

Sta	Length	Data	CRC

The CRC is calculated for each byte in the Sta, Length and Data fields.

14.4 Protocol Commands

14.4.1 Set Configuration

Enable or Disable Data Ready interrupt pin.

Command

Name	Bytes	Description
Cmd	1	Command ID (`0x05`)
Length	2	Data Length (`0x0002`)
CFG0	1	Configuration Byte 0
CFG1	1	Configuration Byte 1
CRC	4	CRC32

Table 5: Set configuration command fields.

`0x05`	`0x02`	`0x00`	CFG0	CFG1	CRC

Name	Bit(s)	Description
Reserved	7:0	Reserved, set to `0x00`

Table 6: CFG0 Fields.

Name	Bit(s)	Description
Reserved	7:3	Reserved, set to `0x00`
DR_PIN	2	Data Ready Interrupt Pin Selection, 0: PIN2, 1: PIN4
DR_EN	1	Data Ready Interrupt Pin Enable, 0: Disabled, 1: Enabled
DR_POL	0	Data Ready Interrupt Pin Polarity, 0: Active Low, 1: Active High

Table 7: CFG1 Fields.

Response

Name	Bytes	Description
Sta	1	Status (`0x05`)
Length	2	Data Length (`0x0002`)
CSR0	1	Configuration Byte 0
CSR1	1	Configuration Byte 1
CRC	4	CRC32

Table 8: Set configuration response fields.

`0x05`	`0x02`	`0x00`	CSR0	CSR1	CRC

Name	Bit(s)	Description
Reserved	7:0	Reserved, ignore

Table 9: CSR0 Fields.

Name	Bit(s)	Description
Reserved	7:3	Reserved, ignore
DR_PIN	2	Data Ready pin selection, 0: PIN2, 1: PIN4
DR_EN	1	Data Ready pin enable, 0: Disabled, 1: Enabled
DR_POL	0	Data Ready pin polarity, 0: Active Low, 1: Active High

Table 10: CSR1 Fields.

14.4.2 Version

Get A2 ROM version information.

Command

Name	Bytes	Description
Cmd	1	Command ID (`0x01`)
Length	2	Data Length (`0x0000`)
CRC	4	CRC32

Table 11: Version command fields.

`0x01`	`0x00`	`0x00`	CRC

Response

Name	Bytes	Description
Sta	1	Status (`0x01`)
Length	2	Data Length (`0x0008`)
Data	8	Version String
CRC	4	CRC32

Table 12: Version response fields.

`0x01`	`0x08`	`0x00`	Version	CRC

14.4.3 Unlock

This command will enable the Write Memory command.

Command

Name	Bytes	Description
Cmd	1	Command ID (`0x81`)
Length	2	Data Length (`0x0005`)
Unlock	1	Unlock (`0x01`)
Key	4	Key (`0x4C, 0x55, 0x4E, 0x44`)
CRC	4	CRC32

Table 13: Unlock command fields.

`0x81`	`0x05`	`0x00`	`0x01`	`0x4C`	`0x55`	`0x4E`	`0x44`	CRC

Response

Name	Bytes	Description
Sta	1	Status (`0x81`)
Length	2	Data Length (`0x0001`)
Unlock	1	Unlock (`0x01`)
CRC	4	CRC32

Table 14: Unlock response fields.

`0x81`	`0x01`	`0x00`	`0x01`	CRC

14.4.4 Write Memory

Write 32-bit words of data to the adddress specified in the write memory command. The full packet length must not exceed 4096 bytes.

Command

Name	Bytes	Description
Cmd	1	Command ID (`0x10`)
Length	2	Data Length
WrAddr	4	Write Address
WrData	4*x	Write Data, x number of 32-bit words
CRC	4	CRC32

Table 15: Write Memory command fields.

`0x10`	Length	WrAddr	WrData	CRC

Response

Name	Bytes	Description
Sta	1	Status (`0x10`)
Length	2	Data Length (`0x0000`)
CRC	4	CRC32

Table 16: Write Memory response fields.

`0x10`	`0x00`	`0x00`	CRC

14.4.5 Start Application

The A2 will start executing at the address specified in the start application command. The application is started the response has been sent.

Command

Name	Bytes	Description
Cmd	1	Command ID (`0x14`)
Length	2	Data Length (`0x0004`)
Addr	4	Address
CRC	4	CRC32

Table 17: Start Application command fields.

`0x14`	`0x04`	`0x00`	Addr	CRC

Response

Name	Bytes	Description
Sta	1	Status (`0x14`)
Length	2	Data Length (`0x0004`)
Addr	4	Address
CRC	4	CRC32

Table 18: Start Application response fields.

`0x14`	`0x04`	`0x00`	Addr	CRC

15 A212 vs A121

The Service APIs for A212 and A121 are quite similar. They are both designed to control the radar and retrieve data from it.

The main updates for A212 compared to A121 are:

- The radar is integrated together with a CPU for embedded applications.
- The radar has multiple transmitters (2) and multiple receivers (4).

Point 1 above means that the RSS library is used within the A212 sensor compared to A121 where it was used outside the sensor. Point 2 above means that the A212 radar is capable of providing angles to objects, which isn’t possible with the A121 sensor.

For more differences, see docs.acconeer.com.

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.