A121 Sparse IQ Service

User Guide

A121 Sparse IQ Service

User Guide

Author: Acconeer AB

Version:a121-v1.12.0

Acconeer AB October 15, 2025

Contents

1 Acconeer SDK Documentation Overview

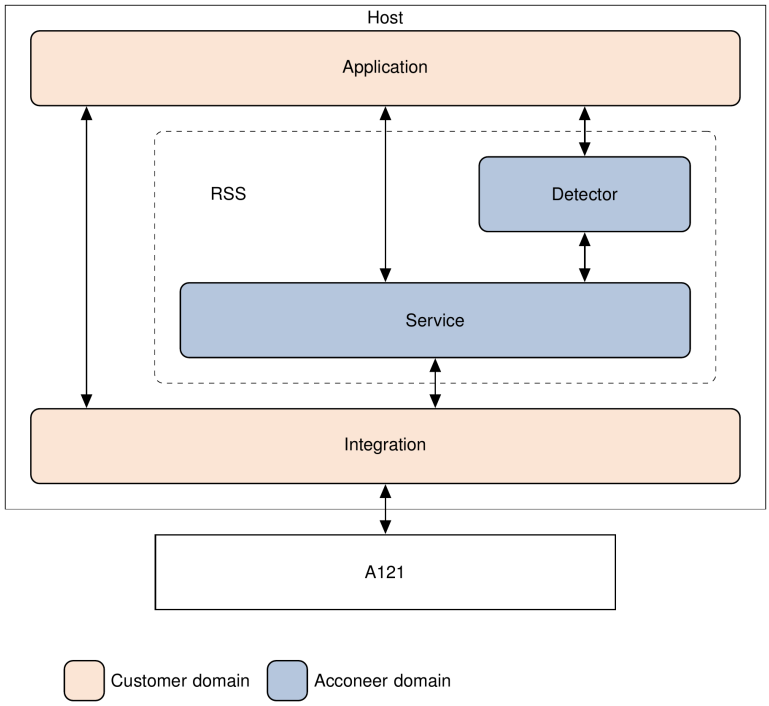
To better understand what SDK document to use, a summary of the documents are shown in the table below.

Table 1: SDK document overview.

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)	
A121 Assembly Test	Describes the Acconeer assembly test functionality.	- Bring-up of HW/SW - Production test implementation
A121 Breathing Reference Application	Describes the functionality of the Breathing Reference Application.	- Working with the Breathing Reference Application
A121 Distance Detector	Describes usage and algorithms of the Distance Detector.	- Working with the Distance Detector
A121 SW Integration	Describes how to implement each integration function needed to use the Acconeer sensor.	- SW implementation of custom HW integration
A121 Presence Detector	Describes usage and algorithms of the Presence Detector.	- Working with the Presence Detector
A121 Smart Presence Reference Application	Describes the functionality of the Smart Presence Reference Application.	- Working with the Smart Presence Reference Application
A121 Sparse IQ Service	Describes usage of the Sparse IQ Service.	- Working with the Sparse IQ Service
A121 Tank Level Reference Application	Describes the functionality of the Tank Level Reference Application.	- Working with the Tank Level Reference Application
A121 Touchless Button Reference Application	Describes the functionality of the Touchless Button Reference Application.	- Working with the Touchless Button Reference Application
A121 Parking Reference Application	Describes the functionality of the Parking Reference Application.	- Working with the Parking Reference Application
A121 STM32CubeIDE	Describes the flow of taking an Acconeer SDK and integrate into STM32CubeIDE.	- Using STM32CubeIDE
A121 Raspberry Pi Software	Describes how to develop for Raspberry Pi.	- Working with Raspberry Pi
A121 Ripple	Describes how to develop for Ripple.	- Working with Ripple on Raspberry Pi
A121 ESP32 User Guide	Describes how to develop with A121 and ESP32 targets.	- Working with ESP32 targets
XM125 Software	Describes how to develop for XM125.	- Working with XM125
XM126 Software	Describes how to develop for XM126.	- Working with XM126
I2C Distance Detector	Describes the functionality of the I2C Distance Detector Application.	- Working with the I2C Distance Detector Application
I2C Presence Detector	Describes the functionality of the I2C Presence Detector Application.	- Working with the I2C Presence Detector Application
I2C Breathing Reference Application	Describes the functionality of the I2C Breathing Reference Application.	- Working with the I2C Breathing Reference Application
I2C Cargo Example Application	Describes the functionality of the I2C Cargo Example Application.	- Working with the I2C Cargo Example Application
	A121 Radar Data and Control (PDF)	
A121 Radar Data and Control	Describes different aspects of the Acconeer offer, for example radar principles and how to configure	- To understand the Acconeer sensor - Use case evaluation
	Readme (txt)	
README	Various target specific information and links	- After SDK download

2 Introduction

The Sparse IQ Service is used to control the Acconeer A121 sensor and retrieve radar data from it. Below is an overview image of how to use the Service API together with the HAL integration in an application.



The focus of this User Guide is the Service API.

It is recommended to use this Guide together with example_service.c located in the SDK package. The full API specification, rss_api.html, provided in the SDK package is also good to use with this Guide.

3 IQ Radar Data

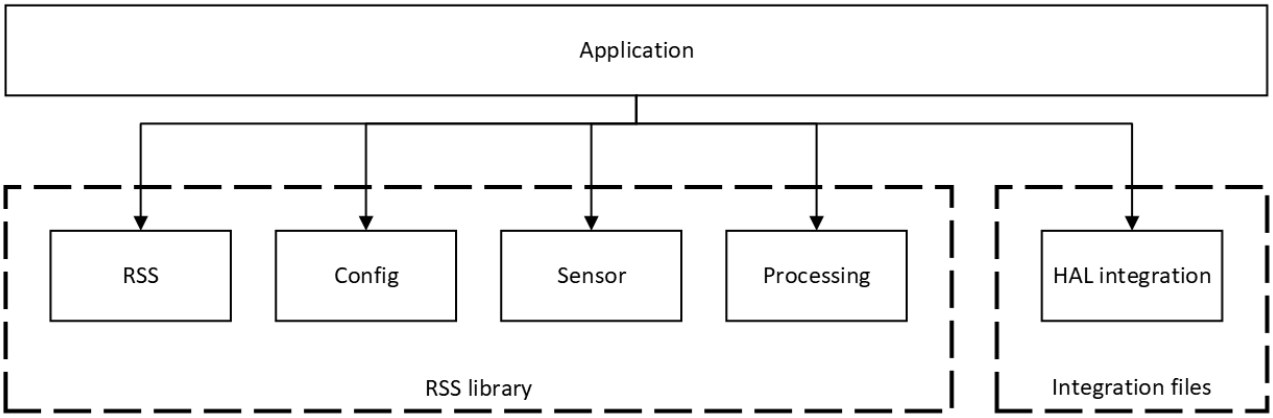
The Sparse IQ Service provides radar data in a complex format. In order to make sense of the data some kind of processing usually needs to be done. Examples of processing can be found in the processing examples provided in the SDK. This includes, among others, amplitude, mean, and interpolation processing.

To read more about how to interpret the IQ data, 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 IQ 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 github.com/acconeer/acconeer-python-exploration.

4 API Usage

The Service API can be divided into five SW modules. See image below.



RSS:

- Initialize the RSS library by registering integration functions
- Get memory size needed by RSS for a specific config
- Set log level

Config:

- Get and set Service configuration
- Print current configuration

Sensor:

- Communicate with the sensor, examples include
  - Calibrate the sensor
  - Configure the sensor
  - Do a measurement
  - Read raw data from sensor

Processing

- Convert unreadable raw data to Sparse IQ data
- Convert unreadable raw data to indications

HAL Integration:

- Retrieve integration functions needed by RSS library
- Direct pin communication with the sensor, such as
  - Enable / disable sensor
  - Wait for interrupt

4.1 Initializing the System

The HAL must be registered to the Radar System Software (RSS) before any other calls are done. The registration requires a pointer to an acc_hal_a121_t struct which contains information on the hardware integration and function pointers to hardware driver functions needed by RSS. The acc_hal_a121_t struct is retrieved by the function acc_hal_rss_integration_get_implementation(). See the “Software Integration” chapter in the document “A121 SW Integration User Guide” for more information on how to integrate the driver layer and populate the hal struct.

4.2 Setting up the Service

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

4.2.1 Register HAL

The HAL must be registered to the Radar System Software (RSS) before any other calls are done.

```
const acc_hal_a121_t *hal = acc_hal_rss_integration_get_implementation();

if (!acc_rss_hal_register(hal))
{
    /* Handle error */
}
```

4.2.2 Create Config

A Config handle must be created before configuration of the sensor 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);
```

4.2.3 Create Processing

A Processing handle needs to be created in order to make sense of the data retrieved from the sensor. The processing handle needs the Config handle to be created.

The Processing creation provides metadata that can be used by the application. Example of metadata is ‘frame data length’, ‘sweep data length’, and ‘maximum sweep rate’.

```
acc_processing_metadata_t proc_meta;

acc_processing_t *processing = acc_processing_create(config, &proc_meta);
if (processing == NULL)
{
    /* Handle error */
}
```

4.2.4 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 and the acc_integration_mem_alloc() function will allocate the “buffer_size” number of bytes for the buffer.

```
uint32_t buffer_size;
if (!acc_rss_get_buffer_size(config, &buffer_size))
{
    /* Handle error */
}

void *buffer = acc_integration_mem_alloc(buffer_size);
if (buffer == NULL)
{
```

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

4.2.5 Create Sensor

A Sensor handle must be created before communication with the sensor can be done. At creation, the ASIC ID is read through SPI. Therefore the sensor must be powered on and enabled before the create function is called.

```
/* Power on sensor */
acc_hal_integration_sensor_supply_on(SENSOR_ID);

/* Enable sensor */
acc_hal_integration_sensor_enable(SENSOR_ID);

/* Create Sensor */
acc_sensor_t *sensor = acc_sensor_create(SENSOR_ID);
if (sensor == NULL)
{
    /* Handle error */
}
```

4.2.6 Calibrate Sensor

The sensor needs to be calibrated before radar data can be retrieved. The calibration has multiple steps, so it needs to be done in a loop where the acc_sensor_calibrate() function is called interleaved with waiting for the sensor interrupt.

```
/* Calibrate Sensor */
bool              status;
bool              cal_complete = false;
acc_cal_result_t  cal_result;

do
{
    status = acc_sensor_calibrate(sensor, &cal_complete, &cal_result, buffer
        , buffer_size);

    if (status && !cal_complete)
    {
        status = acc_hal_integration_wait_for_sensor_interrupt(SENSOR_ID,
            SENSOR_TIMEOUT_MS);
    }
} while (status && !cal_complete);

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

/* Reset sensor after calibration by disabling it */
acc_hal_integration_sensor_disable(SENSOR_ID);
```

4.2.7 Prepare Sensor

The configuration is loaded to the sensor by calling the acc_sensor_prepare() function. The sensor must be powered on and enabled before the acc_sensor_prepare() function is called.

```
acc_hal_integration_sensor_enable(SENSOR_ID);

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

4.3 Data Retrieval and Processing

4.3.1 Measure and Read Data from Sensor

The radar measurement is started when the acc_sensor_measure() function is called. The sensor will generate an interrupt to the host by setting the interrupt pin high when the radar measurement is done. After the interrupt is received, the acc_sensor_read() can be called to read raw data from the sensor over SPI.

```
if (!acc_sensor_measure(sensor))
{
    /* Handle error */
}

if (!acc_hal_integration_wait_for_sensor_interrupt(SENSOR_ID,
   SENSOR_TIMEOUT_MS))
{
    /* Handle error */
}

if (!acc_sensor_read(sensor, buffer, buffer_size))
{
    /* Handle error */
}
```

4.3.2 Data Processing

To make sense of the raw data, it must be processed with acc_processing_execute(). This converts the raw data to a Sparse IQ frame and indications. The result is called a 'processing result'.

```
acc_processing_result_t proc_result;

acc_processing_execute(processing, buffer, &proc_result);

uint16_t sweeps_per_frame = acc_config_sweeps_per_frame_get(config);
uint16_t sweep_length = proc_meta.sweep_data_length;

for (uint16_t sweep_index = 0; sweep_index < sweeps_per_frame; sweep_index
   ++)
{
    for (uint16_t distance_index = 0; distance_index < sweep_length;
       distance_index++)
    {
        uint16_t index_in_frame = sweep_index * sweep_length +
           distance_index;

        printf("%" PRIi16 "+%" PRIi16 "i\n", proc_result.frame[
           index_in_frame].real, proc_result.frame[index_in_frame].imag);
    }
}
```

The 'proc_meta' above is retrieved from the function 'acc_processing_create()', described in the Create Processing section.

As can be seen above, the IQ frame layout is in the order sweep and then distance. One could think of the layout as having sweeps in the rows and distances in the columns. If more than one subsweep is used, the layout is in the order sweep, subsweep, and distance. See below.

```
uint16_t sweeps_per_frame = acc_config_sweeps_per_frame_get(config);
uint8_t number_of_subsweeps = acc_config_num_subsweeps_get(config);
uint16_t sweep_length = proc_meta.sweep_data_length;


for (uint16_t sweep_index = 0; sweep_index < sweeps_per_frame; sweep_index
   ++)
```

```
{
    for (uint8_t subsweep_index = 0; subsweep_index < number_of_subsweeps;
        subsweep_index++)
    {
        for (uint16_t distance_index = 0; distance_index < proc_meta.
            subsweep_data_length[subsweep_index]; distance_index++)
        {
            uint16_t index_in_frame = sweep_index * sweep_length + proc_meta
                .subsweep_data_offset[subsweep_index] + distance_index;

            printf("%" PRIi16 "+%" PRIi16 "i\n", proc_result.frame[
                index_in_frame].real, proc_result.frame[index_in_frame].imag)
                ;
        }
    }
}
```

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

4.4 Shutdown and Memory De-allocation

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

```
/* Disable Sensor */
acc_hal_integration_sensor_disable(SENSOR_ID);

/* Power off Sensor */
acc_hal_integration_sensor_supply_off(SENSOR_ID);

if (sensor != NULL)
{
    acc_sensor_destroy(sensor);
}

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

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

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

4.5 Configuration

The sensor 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(). See the Sparse IQ Configuration Parameters section for a condensed list of the configurations that can be set.

Some configurations apply to the whole service, such as 'num_subsweeps' and 'inter_frame_idle_state'. 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.

4.5.1 Number of Subsweeps

This configuration sets the number of subsweeps to use. See docs.acconeer.com for more information on the concept of subsweeps.

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.

How to set number of subsweeps.

```
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);
```

4.5.2 Continuous Sweep Mode

In continuous sweep mode the timing will be identical over all sweeps, not just the sweeps in a frame. The continuous sweep mode will only work when:

More information about continuous sweep mode can be found in the Advanced Sensor Control section.

```
void acc_config_continuous_sweep_mode_set(acc_config_t *config, bool enabled
   );
bool acc_config_continuous_sweep_mode_get(const acc_config_t *config);
```

4.5.3 Idle States

The sensor can enter three different idle states between frames and sweeps. The idle states are DEEP SLEEP, SLEEP and READY. READY is the shallowest idle state and consumes the most power. DEEP SLEEP is the deepest state and consumes the least power. The idle states affect how fast the sensor can measure. READY is the fastest and DEEP SLEEP is the slowest. So setting an idle state is a trade-off between power consumption and measurement speed.

More information about idle states can be found in the Advanced Sensor Control section.

Inter Frame Idle State

```
void acc_config_inter_frame_idle_state_set(acc_config_t *config,
   acc_config_idle_state_t idle_state);
acc_config_idle_state_t acc_config_inter_frame_idle_state_get(const
   acc_config_t *config);
```

Inter Sweep Idle State

```
void acc_config_inter_sweep_idle_state_set(acc_config_t *config,
   acc_config_idle_state_t idle_state);
acc_config_idle_state_t acc_config_inter_sweep_idle_state_get(const
   acc_config_t *config);
```

4.5.4 Double Buffering

If enabled, the sensor can start a new measurement while the host is reading data from the sensor. It can be used to decrease the idle time in between frames from the sensor.

More information about double buffering can be found in the Advanced Sensor Control section.

```
void acc_config_double_buffering_set(acc_config_t *config, bool enable);
bool acc_config_double_buffering_get(const acc_config_t *config);
```

5 Sensor Calibration

The sensor needs to be calibrated before radar data can be retrieved. 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 sensor calibration to work. This means that a (re)calibration can be done at any time.

The sensor calibration is done in multiple steps. This means that during a calibration, multiple SPI transfers will be done. An integration with a fast SPI will reduce the calibration time. As a reference, the calibration takes around 50 ms on the XM125 module.

In each step, the calibration routine extracts data from the sensor and evaluates it. There are sanity checks of this sensor 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, resetting the sensor (by setting enable low/high) and doing the calibration once more should solve the issue. If the issue persists, there might be an issue with the sensor or integration.

The result of a calibration is stored as an uint32_t array in a result struct.

```
typedef struct
{
    uint32_t data[ACC_CAL_RESULT_DATA_SIZE / 4];
} acc_cal_result_t;
```

This result is not portable between architectures. This means that the result must be used on the same platform as it was produced.

5.1 Sensor Re-calibration

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 Indication Handling.

5.2 Sensor Calibration Caching

To reduce time spent on calibration in an application, the calibration result can be cached. This means that a calibration result is saved for a specific temperature. If the sensor needs to be re-calibrated, the saved calibration result can be used instead of doing a new calibration. The saved calibration result should only be used if it was done within a temperature range of at most +- 15 degrees Celsius from the current temperature.

The implementation of sensor calibration caching needs to be done in the application, i.e. it is not part of the RSS library itself. To see an example of how it can be done, please look in 'example_service_calibration_caching.c'.

Sensor calibration caching is typically done to reduce power consumption in applications where temperature changes are common.

6 Indication Handling

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

```
acc_processing_result_t proc_result;

acc_processing_execute(processing, buffer, &proc_result);

if (proc_result.data_saturated)
{
    // Handle data saturated indication
}

if (proc_result.frame_delayed)
{
    // Handle frame delayed indication
}

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 Data Saturated

Indication of sensor data being saturated, which causes data corruption.

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

Handle by lowering the receiver gain configuration.

```
void acc_config_receiver_gain_set(acc_config_t *config, uint8_t gain);
uint8_t acc_config_receiver_gain_get(const acc_config_t *config);
```

6.2 Frame Delayed

Indication that a radar data frame is delayed.

This happens if a frame rate in the sensor has been set and the host doesn't read out the data within the frame period.

Handle by lowering the frame rate.

```
void acc_config_frame_rate_set(acc_config_t *config, float frame_rate);
float acc_config_frame_rate_get(const acc_config_t *config);
```

Alternative solutions could be to enable the 'double_buffering' mode or to reduce the radar data being read and processed. This can be done by reconfiguring the sensor. The configurations impacting the size of the radar data the most are

- num_points
- step_length
- sweeps_per_frame

6.3 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.

Handle by resetting sensor (setting enable low/high) and redo the calibration.

6.4 Temperature

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

Note that the absolute accuracy is poor and the temperature must only be used for relative comparisons.

The temperature indication can be used for different types of comparisons or compensations. As an example, the Distance Detector uses temperature compensation, see 'A121 Distance Detector User Guide' for more information.

7 Examples

Multiple different examples are provided in the SDK. They are divided into four categories, see table below.

7.1 Getting Started

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

- example_bring_up - Shows how RSS assembly test can be used during bring-up of custom hardware
- example_service - Shows basic usage of the Sparse IQ Service
- example_detector_distance - Shows basic usage of the Distance Detector API
- example_detector_presence - Shows basic usage of the Presence Detector API
- example_control_helper - Shows how the control helper API can be used

7.2 Processing

These examples show different ways to process the Sparse IQ Data. The name of each example correspond to the processing demonstrated.

- example_processing_amplitude
- example_processing_coherent_mean
- example_processing_noncoherent_mean
- example_processing_peak_interpolation
- example_processing_subtract_adaptive_bg

7.3 Advanced Control

These examples show different, more advanced, ways to use and configure the API. They are useful depending on use case. The name of each example correspond to what is being demonstrated.

- Sparse IQ Service configuration examples
- Sparse IQ Service control examples
- Distance Detector configuration examples
- Presence Detector configuration examples
- Low power examples - only present in the XM125 SDK

7.4 Troubleshooting

These examples can be used when issues arise.

example_diagnostic_test - Shows how to use the RSS diagnostic test to get sensor diagnostic information

8 Communication Issues

All persistent communication issues should be solved during bring-up.

However, unexpected external disturbance can temporarily negatively affect the communication between host and sensor. If this happens, RSS returns failure from its various communication functions or waiting for sensor interrupt times out. Basic communication functions are

- acc_sensor_create()
- acc_sensor_calibrate()
- acc_sensor_prepare()
- acc_sensor_measure()
- acc_sensor_read()

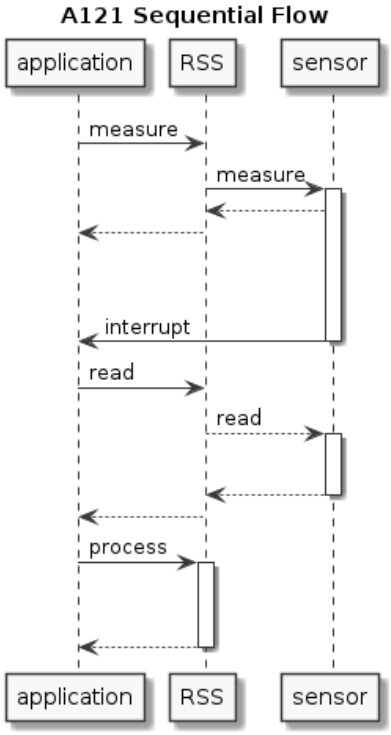
When a temporary communication failure occurs the following should be done

- Reset sensor (setting enable low/high)
- Destroy the Sensor handle
- Call the above communication functions again

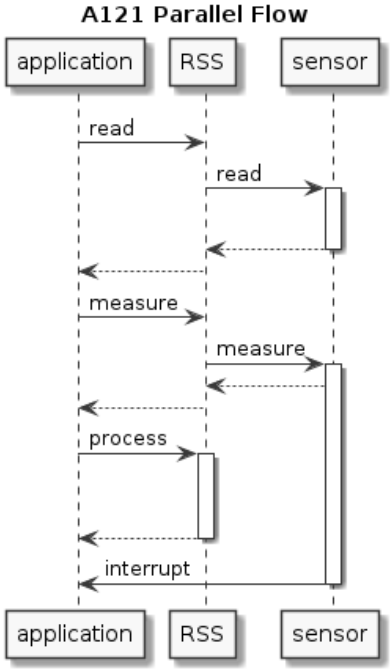
9 Advanced Sensor Control

9.1 Sequential vs Parallel Control Flow

The control flow described in section ‘Data Retrieval and Processing’ is sequential. This means that the sensor measurement, read-out of data, and data processing in host are all done one after the other. See image below.



This is the simplest control flow which is easy to follow as nothing happens in parallel. All examples are using this kind of control flow. An alternative control flow is to let the host do data processing at the same time as the sensor is doing a measurement. See image below.



This control flow assumes that at least one measure and one wait for sensor interrupt has been done previously. Using the parallel control flow increases the maximum possible update rate. It can also lower power consumption by handling the data faster and thus letting the system go to sleep for a longer time.

9.2 Inter Frame and Inter Sweep Idle States

The sensor has different idle states to be able to either save power in and in between measurements or to be able to have a high frame- and/or sweep-rate. The maximum frame-/sweep-rate will decrease when using *Sleep* and *Deep Sleep* idle states due to transition time between the idle states.

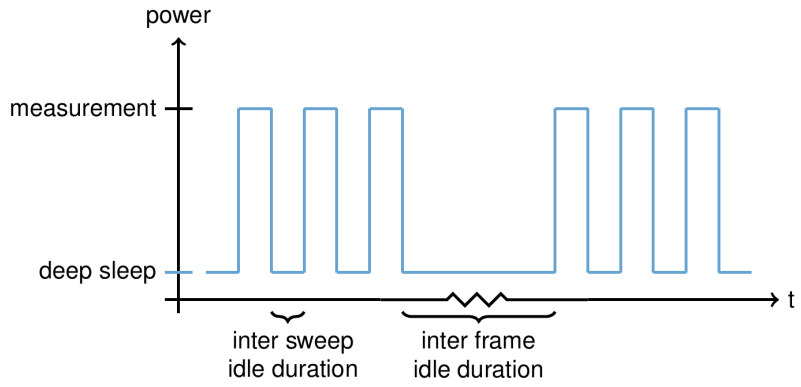
The **inter_frame_idle state** is the state the sensor idles in between each frame. The **inter_sweep_idle_state** is the state the sensor idles in between each sweep.

The **inter_frame_idle_state** of the frame must be deeper or the same as the **inter_sweep_idle_state**.

The graphs below show the power usage in between frames/sweeps depending on the inter frame and inter sweep idle states.

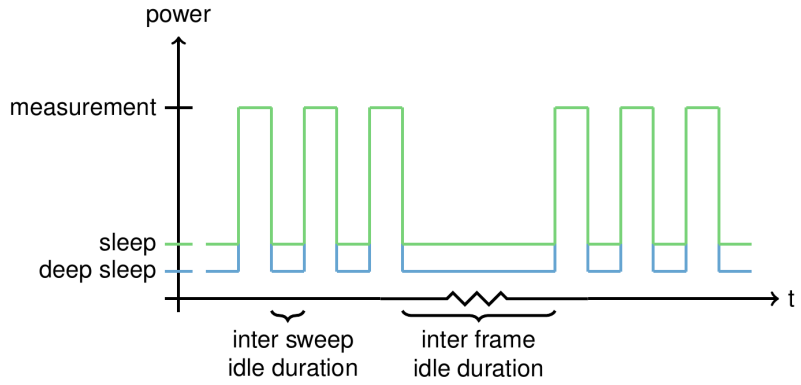
9.2.1 Deep Sleep

Deep sleep is the deepest state where as much of the sensor hardware as possible is shut down, it is also the state that is slowest to transition from.



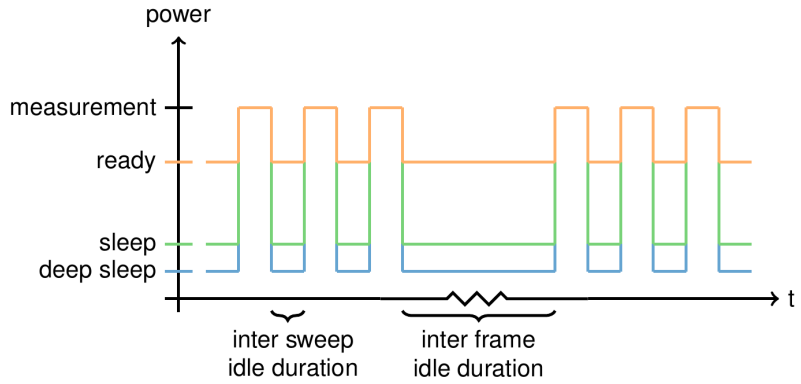
9.2.2 Sleep

Sleep is a compromise between power save and transition time.



9.2.3 Ready

Ready is the shallowest state where most of the sensor hardware is kept on, it is also the state that is the fastest to transition from.



9.3 Double Buffering

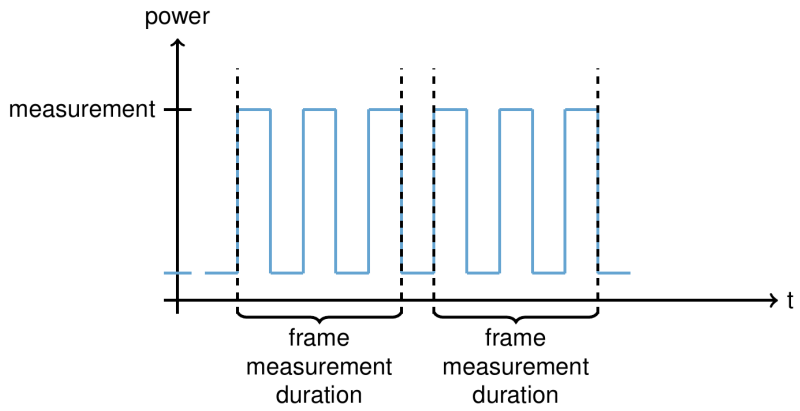
If enabled, the sensor buffer will be split in two halves reducing the maximum number of samples. A frame can be read using the acc_sensor_read() function from one half of the sensor buffer while sampling is done into the other half. Switching of buffers is done automatically by the acc_sensor_measure() function, so no extra functionality needs to be implemented by the application.

When using double buffering, measurements coinciding with SPI activity may have distorted phase. To mitigate this issue, applying a median filter is recommended.

An example of how to do this median filtering can be found in acc_algorithm.c provided in the Acconeer SW packages. See example below of how to use it The function does not correct disturbances that may appear in the initial or final sweeps.

```
/**
 * Brief example on how to use the double buffering median frame filter
 */

#include "acc_algorithm.h"

#define NUM_POINTS 60
#define SWEEPS_PER_FRAME 32


...


/* Create configuration */
config = acc_config_create();
if (config == NULL)
{
    /* ERROR */
}

acc_config_num_points_set(config, NUM_POINTS);
acc_config_sweeps_per_frame_set(config, SWEEPS_PER_FRAME);


...


int32_t work_buffer[SWEEPS_PER_FRAME - 2];

/* Read sensor data */
if (!acc_sensor_measure(sensor))
{
    /* ERROR */
}

if (!acc_hal_integration_wait_for_sensor_interrupt(SENSOR_ID,
   SENSOR_TIMEOUT_MS))
{
    /* ERROR */
}

if (!acc_sensor_read(sensor, buffer, buffer_size))
{
    /* ERROR */
}

/* Process sensor data */
acc_processing_execute(processing, buffer, &proc_result);

/* Apply filter to frame */
acc_algorithm_double_buffering_frame_filter(proc_result.frame,
   SWEEPS_PER_FRAME, NUM_POINTS, work_buffer);
```

The graph below show the timing and power for a double buffering configuration.



9.4 Continuous Sweep Mode

With Continuous Sweep Mode (CSM), the sensor timing is set up to generate a continuous stream of sweeps even if more than one sweep per frame is used. The interval between the last sweep in one frame to the first sweep in the next frame becomes equal to the interval between sweeps within a frame (given by the sweep rate).

If only one sweep per frame is used, CSM has no use since a continuous stream of sweeps is already given (if a fixed frame rate is used).

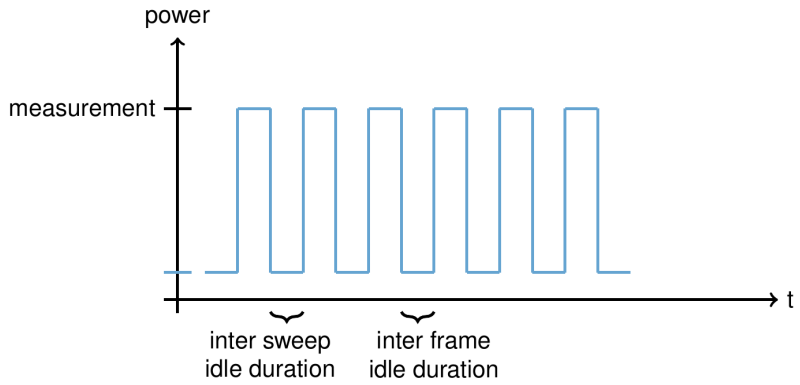
The main use for CSM is to allow reading out data at a slower rate than the sweep rate, while maintaining that sweep rate continuously.

Note that in most cases, double buffering must be enabled to allow high rates without delays.

Constraints:

- **frame_rate** must be 0 (unlimited)
- **sweep_rate** must be > 0
- The **inter_frame_idle_state** of the frame must be equal to **inter_sweep_idle_state**

The graph below show the timing and power for a CSM configuration.

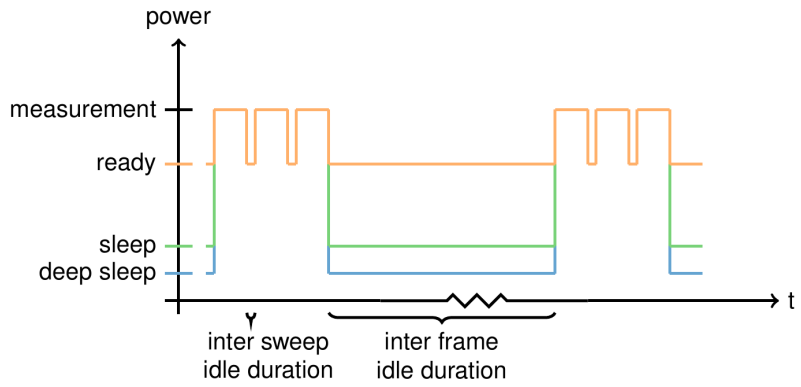


9.5 High Speed Mode

High speed mode means that the sensor is configured in a way where it can optimize its measurements to obtain as high sweep rate as possible. In order for the sensor to operate in high speed mode, the following configuration constraints apply:

- **continuous_sweep_mode** must be disabled
- **inter_sweep_idle_state** must be *Ready*
- **num_subsweeps** must be 1
- **profile** must be 3, 4 or 5

The graph below show the timing and power for a high speed mode configuration.



10 Memory

10.1 Flash

The example application compiled from example_service.c on the XM125 module requires around 70 kB.

10.2 RAM

The RAM can be divided into three categories, static RAM, heap, and stack. Below is a table for approximate RAM for an application compiled from example_service.c.

RAM	Size (kB)
Static	1
Heap	4
Stack	5
Total	10

Note that heap is very dependent on the configuration. 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’, and ‘step_length’.

11 A121 vs A111

The Service APIs for A121 and A111 are quite similar. They are both designed to control the sensor and retrieve data from it.

The main updates to control for A121 compared to A111 are:

- The pin handling, like ENABLE, is completely handled by the application
- The majority of the memory is allocated in the application allowing for memory reuse between Services
- The Service handle is divided into a Sensor handle and a Processing handle
- The Sensor handle can be used to separately do a calibration
- The Sensor handle can be used to (re)configure the sensor faster using a ‘prepare’ function
- The get_next function is replaced by a function sequence of ‘measure’ -> ‘wait_for_interrupt’ -> ‘read’ -> ‘process’

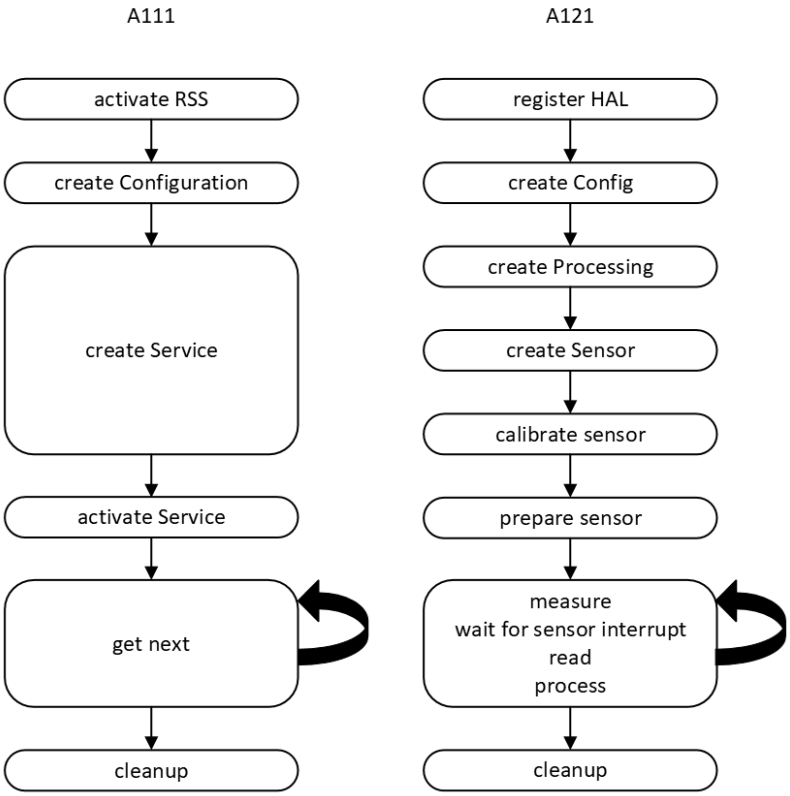
The main updates to radar data for A121 compared to A111 are:

- There is only one Service, the Sparse IQ Service, providing complex IQ data
- The radar data can be divided into subsweeps

The differences generally allow for:

- More advanced multi-sensor scheduling
- Detailed sweep configuration
- More power optimized applications
- More memory optimized applications

Below is an image showing the differences in control flow.



For more differences, see docs.acconeer.com.

12 Sparse IQ Configuration Parameters

Name	Type	Default Value	Min	Max
start_point	int32_t	80		
num_points	uint16_t	160		
step_length	uint16_t	1	1	
hwaas	uint16_t	8	1	511
receiver_gain	uint8_t	16	0	23
enable_tx	bool	true	n/a	n/a
phase_enhancement	bool	false	n/a	n/a
iq_imbalance_compensation	bool	false	n/a	n/a
enable_loopback	bool	false	n/a	n/a
prf	enum	prf_15_6_mhz	prf_5_2_mhz	prf_19_5_mhz
profile	enum	profile_3	profile_1	profile_5
sweep_rate	float	0.0		
frame_rate	float	0.0		
sweeps_per_frame	uint16_t	1		
continuous_sweep_mode	bool	false	n/a	n/a
double_buffering	bool	false	n/a	n/a
num_subsweeps	uint8_t	1	1	4
inter_frame_idle_state	enum	deep_sleep	deep_sleep	ready
inter_sweep_idle_state	enum	ready	deep_sleep	ready

Table 3: Sparse IQ Configuration Parameters

13 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.