Iq Service

User Guide

Iq Service

User Guide

Author: Acconeer AB

Version:a111-v2.15.2

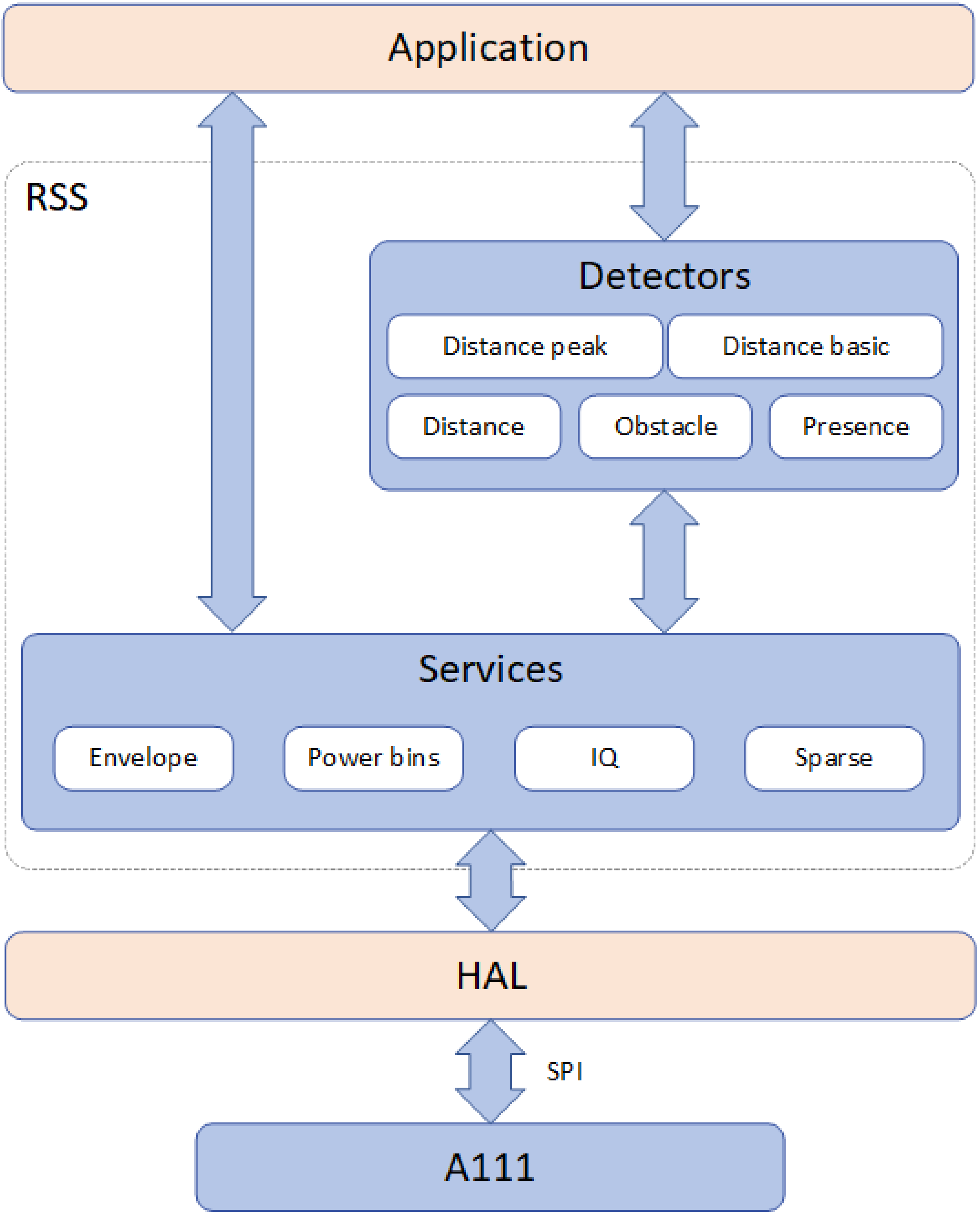
Acconeer AB August 18, 2023

Contents

1 IQ Service

The IQ Service is one of four services that provide an interface for reading out the radar signal from the Acconeer A111 sensor. The IQ-data can be seen as an extension of the Envelope data. In addition to the amplitude, the IQ data also includes information on the phase of the radar signal. The data returned from the IQ Service is represented as complex numbers and the IQ data is typically further processed using various signal processing algorithms. The IQ data can for example be used for measurement of small changes in distance with micrometer accuracy or for efficient background cancellation.

For applications where the phase information is not needed you may consider using the Envelope service or Power Bins service instead. They both provide amplitude data and are easier to understand and work with compared to the IQ Service. The Envelope service provides full resolution, whereas the power bins interface provides less processed subsampled amplitude data.



Before using the IQ service, we recommend that you have a basic understanding of complex numbers and how they are used to represent phase and amplitude in signal processing.

Acconeer provide an example of how to use the IQ service: example_service_iq.c

For more details on the IQ data it is recommended to use our exploration tool. Check it out on GitHub Acconeer

Exploration Tool.

1.1 Disclaimer

Profile 3-5 will not have optimal performance using A111 with batch number 10467, 10457 or 10178 (also when mounted on XR111 and XR112). XM112 and XM122 are not affected since they have A111 from other batches.

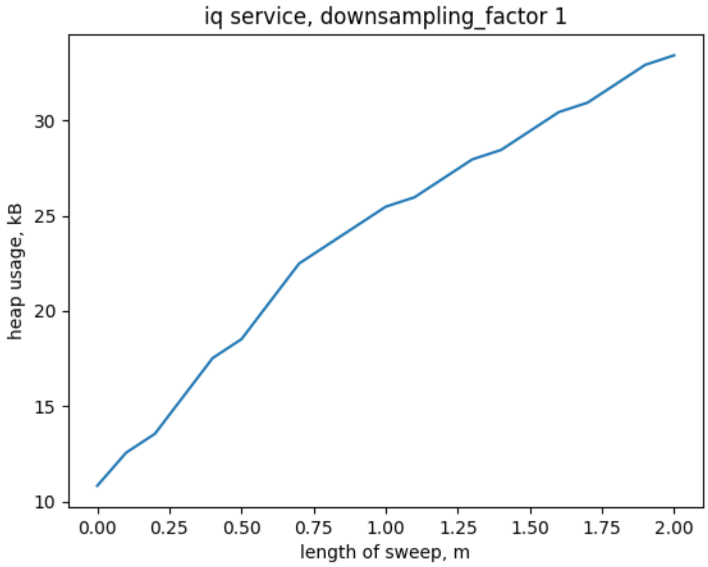
2 Memory Requirements

2.1 Flash footprint

The different services in RSS have different flash footprint depending on the complexity of the algorithm used to process the data. The example program for the IQ service requires 112 kB flash when built in STM32Cube environment.

2.2 Heap usage

The amount of heap used by the IQ service depends on how the application configures the service. Different sweep length and downsampling factor impact the size of the buffer needed to keep the service data. A downsampling factor of 1 gives better accuracy in the measurement but a downsampling factor of 4 can be used in applications where less heap usage is more important than accuracy. The client can also choose to use acc_service_iq_get_next() if it wants to save the envelope data in a separate buffer or acc_service_iq_get_next_by_reference() if it wants to use the same buffer as is used in the service.



The figure above gives an indication of how much memory that is allocated on the heap in relation to sweep length and acc_service_iq_get_next() is used to retrieve the data.

3 Setting up the Service

3.1 Initializing the System

The Radar System Software (RSS) must be activated before any other calls are done. The activation requires a pointer to an acc_hal_t struct which contains information on the hardware integration and function pointers to hardware driver functions that are needed by RSS. See chapter 4 in the document "HAL Integration User Guide" for more information on how to integrate the driver layer and populate the hal struct.

In Acconeer's example integration towards STM32 and the drivers generated by the STM32Cube tool, there is a function acc_hal_integration_get_implementation to obtain the hal struct.

```
const acc_hal_t *hal = acc_hal_integration_get_implementation();

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

The corresponding code looks slightly different in software packages for the Raspberry Pi and other software packages from Acconeer where peripheral drivers for the host are included. The hal struct is then obtained with the function acc_hal_integration_get_implementation.

```
const acc_hal_t *hal = acc_hal_integration_get_implementation();

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

3.2 RSS Configuration

There is one configuration for RSS that takes effect for all services and detectors. That configuration is 'Override Sensor ID Check at Creation' and makes it possible to create multiple services and/or detectors for the same sensor ID. The configuration can be set by calling:

```
acc_rss_override_sensor_id_check_at_creation(true);
```

A normal situation where this can be of benefit is when an application wants to switch between services and/or detectors easily and efficiently or when an application wants to switch between configurations of the same service/detector. An example of how to do this can be found in example_multiple_service_usage.c.

3.3 Service API

All services in the Acconeer API are created and activated in two distinct steps. In the first creation step the configuration settings are evaluated and all necessary resources are allocated. If there is some error in the configuration or if there are not enough resources in the system to run the service, the creation step will fail. However, when the creation is successful you can be sure that the second activation step will not fail due to any configuration or resource issues. When the service is activated the radar is activated and the radar data starts to flow from the sensor to the application.

3.4 IQ Service Configuration

Before the IQ service can be created and activated, we must prepare a service configuration. First a configuration is created.

```
acc_service_configuration_t iq_configuration =
   acc_service_iq_configuration_create();

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

The newly created service configuration contains default settings for all configuration parameters and can be passed directly to the acc_service_create function. However, in most scenarios there is a need to change at least some of the configuration parameters. See acc_service_iq.h and acc_service.h for a complete description of configuration parameters.

3.4.1 Configuration Summary

Below is two tables of all possible configurations for the IQ Service. Note that some configurations can have other limits than the ones listed below. For example, 'Length' in combination with 'Downsampling Factor' is dependent on the available memory of the system.

Generic Configurations :

Parameter	Description	Type	Unit	Limits
Sensor	Sensor ID	integer	N/A	[1 -]
Start	Start of measurement	float	meters	[-0.7 - 7.0]
Length	Length of measurement	float	meters	[0.0 - 7.7]
Repetition Mode	See below	On demand / Streaming	N/A	N/A

Parameter	Description	Type	Unit	Limits
Power Save Mode	See below	enum	N/A	N/A
Receiver Gain	Sensor receiver gain	float	N/A	[0.0 - 1.0]
TX Disable	Disable Radio Transmitter	bool	N/A	N/A
HWAAS	See below	integer	N/A	[1 - 63]
Profile	See below	enum	N/A	N/A
Maximize Signal Attenuation	Maximize signal attenuation in sensor	bool	N/A	N/A
Asynchronous Measurement	Enable Asynchronous Measurement	bool	N/A	N/A

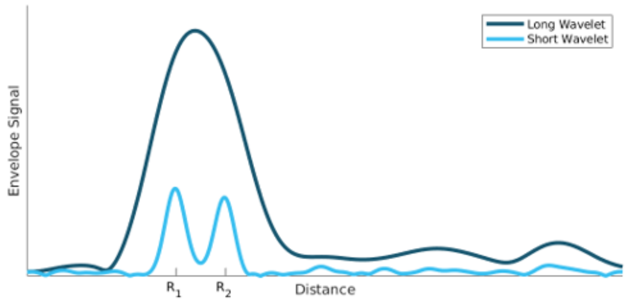
IQ Specific Configurations :

Parameter	Description	Type	Unit	Limits
Depth Lowpass Cutoff Ratio	Cutoff frequency ratio for distance domain low-pass filter of data	float	N/A	[0.0 - 0.5]
Downsampling Factor	See below	integer	N/A	[1, 2, 4]
Noise Level Normalization	Flag to determine if data should be normalized with noise floor	bool	N/A	N/A
Output Format	See below	enum	N/A	N/A

The following sections describe some of the configurations in more detail.

3.4.2 Profiles

The services and detectors support profiles with different configuration of emission in the sensor. The different profiles provide an option to configure the pulse length and optimize on either depth resolution or radar loop gain. More information regarding profiles can be read in the Radar sensor introduction document.



The figure above shows the envelope signal of the same objects with two different profiles, one with a short pulse and one with a long pulse.

The IQ service supports five different profiles which are defined in acc_definitions_a111.h. Profile 1 has the shortest pulse and should be used in applications which aim to see multiple objects or with short distance to the object. Profiles with higher numbers have longer pulse and are more suitable to use in applications which aim to see objects with weak reflection or objects further away from the sensor. The highest profiles, 4 and 5, are optimized for maximum radar loop gain which leads to lower precision in the distance estimate. It is recommended to start using profile 2 and 3. Profile 1, 4 and 5 are not yet optimized for all configurations for the IQ service.

Profiles can be configured by the application by using a set function in the service api. The default profile is ACC_SERVICE_PROFILE_2.

```
void acc_service_profile_set(acc_service_configuration_t
    service_configuration,
                             acc_service_profile_t       profile);
```


3.4.3 Repetition Mode

RSS supports two different repetition modes which configure the control flow of the sensor when it's producing data. In both modes, the application initiates the data transfer from the sensor and is responsible to keep the timing by fetching data from the service. The repetition modes are called on_demand and streaming and the default mode is on_demand.

Repetition mode on_demand lets the application decide when the sensor produces data. This mode is recommended to be used if the application is not dependent of a fixed update rate and it's more important for the application to control the timing. An example could be if the application requests data at irregular time or with low frequency and it's more important to enable low power consumption. Repetition mode on_demand should also be used if the application set a length which requires stitching or want to use power save mode off.

```
void acc_service_repetition_mode_on_demand_set(service_configuration_t
   service_configuration);
```

Repetition mode streaming configures the sensor to produce data based on a hardware timer which is very accurate. It is recommended to use repetition mode streaming if the application requires very accurate timing. An example could be if the data should be processed with an FFT. This mode can not be used it the application set a length which requires stitching.

```
void acc_service_repetition_mode_streaming_set(service_configuration_t
   service_configuration, float update_rate);
```

3.4.4 Downsampling Factor

In the IQ service, the base step length is ˜0.5mm. The default configuration enables the sensor to produce data at every point and will give the highest resolution. Applications that don't require as high resolution can downsample the data on the sensor by increasing the step length. For example setting downsampling factor to 4 makes the distance between two points in the measured range ˜2mm. Less data require less processing and could be useful in applications which require low power consumption. The IQ service supports a downsampling factor of 1, 2, or 4.

```
void acc_service_iq_downsampling_factor_set(acc_service_configuration_t
   service_configuration, uint16_t downsampling_factor);
```

The actual step length is reported back from the service in acc_service_iq_metadata_t.

3.4.5 Hardware Accelerated Average Samples (HWAAS)

The sensor can be configured with the number of samples measured and averaged to obtain a single point in the data. These samples are averaged directly in the sensor hardware and only one value for each point is transferred over SPI. Therefore, increasing HWAAS is a both memory and computationally inexpensive way to increase the SNR. The time needed to measure a sweep is roughly proportional to the number of averaged samples. Hence, if there is a need to obtain a higher update rate, HWAAS could be decreased but this leads to lower SNR. The HWAAS value must be at least 1 and not larger than 63, the default value for the IQ service is 10.

```
void acc_service_hw_accelerated_average_samples_set(
   acc_service_configuration_t configuration, uint8_t samples);
```

3.4.6 Power Save Mode

The power save mode configuration sets what state the sensor waits in between measurements in an active service. There are five power save modes and the modes differentiate in current dissipation and response latency, where the most current consuming mode 'ACTIVE' gives fastest response and the least current consuming mode 'OFF' gives the slowest response. The absolute response time and also maximum update rate is determined by several factors besides the power save mode configuration. These are length, and hardware accelerated average samples. In addition, the host capabilities in terms of SPI communication speed and processing speed also impact on the absolute response time. The power consumption of the system depends on the actual configuration of the application and it is recommended to test both maximum update rate and power consumption when the configuration is decided.

Mode 'HIBERNATE' means that the sensor is still powered but the internal oscillator is turned off and the application needs to clock the sensor by toggling a GPIO a pre-defined number of times to enter and exit this mode. Mode 'HIBERNATE' is currently only supported by the Sparse service and require additional functions to be implemented in the HAL.


```
typedef enum
{
    ACC_POWER_SAVE_MODE_OFF,
    ACC_POWER_SAVE_MODE_SLEEP,
    ACC_POWER_SAVE_MODE_READY,
    ACC_POWER_SAVE_MODE_ACTIVE,
    ACC_POWER_SAVE_MODE_HIBERNATE,
} acc_power_save_mode_enum_t;
typedef uint32_t acc_power_save_mode_t;
```

```
void acc_service_power_save_mode_set(acc_service_configuration_t
   configuration,
                                     acc_power_save_mode_t
                                        power_save_mode);
```

The achievable update rate and power consumption of the sensor in different use cases vary between different hosts. The computational capacity and data transfer rate over SPI impacts when different modes are used in the most optimal way. A few common use cases are:

- Update rate less than 1 Hz: Mode 'OFF' turns off the sensor between sweeps and is typically used in applications which require low update rate.
- Update rate 1-4 Hz: Mode 'OFF' turns off the sensor between sweeps and should be used in applications with low update rate. In mode 'OFF', the sensor needs to be restarted and the sensor firmware loaded between updates and this have a penalty for hosts with lower SPI frequency. Therefore, it's recommended to measure the power consumption of the system with different power save modes and choose the most optimal settings when reaching update rates of 5-10 Hz.
- Update rate more than 5 Hz: Power mode 'SLEEP' is recommended for applications where the power consumption is important. If expected update rate is not enough with mode 'SLEEP', the application should use 'READY' instead.
- Max update rate: Select power save mode 'ACTIVE' for applications without power constraints that needs to maximize the update rate.
- Fetching a burst of frames: Some applications need to fetch a burst of frames from the sensor and then sleep for a longer period. This kind of application is recommended to use mode 'READY' for fast update rate between the frames in the burst to minimize the execution time when the MCU is active. To save maximum power it is recommended to deactivate the service between the bursts. This will put the sensor in power-off state when it is not used.

3.4.7 Output Data Format

The IQ service have an option to provide two different types of output format, int16 complex and float complex. The default output type is int16 complex and the type can be changed with a configuration function.

```
typedef enum
{
    ACC_SERVICE_IQ_OUTPUT_FORMAT_FLOAT_COMPLEX, // The output format is
       float complex
    ACC_SERVICE_IQ_OUTPUT_FORMAT_INT16_COMPLEX  // The output format is
       acc_int16_complex_t
} acc_service_iq_output_format_enum_t;
typedef uint32_t acc_service_iq_output_format_t;

void acc_service_iq_output_format_set(acc_service_configuration_t
   service_configuration,
                                      acc_service_iq_output_format_t format)
                                         ;
```

3.4.8 Asynchronous Measurement

RSS supports two different measurement modes, synchronous and asynchronous. The default mode is asynchronous.

In synchronous mode, the following will occur when acc_service_iq_get_next() / acc_service_iq_get_next_by_reference() is called:

1. Start the sweep.
2. Wait for the sweep to finish, the time for this will vary depending on the configuration and the sweep length.
3. Transfer sweep data from the sensor.
4. Process data.
5. Return from function.

In asynchronous mode, the following will occur when acc_service_iq_get_next() / acc_service_iq_get_next_by_reference() is called:

1. Wait for previous sweep to finish
2. Transfer sweep data from the sensor.
3. Start the next sweep.
4. Process data.
5. Return from function.

The main difference between the modes is that in asynchronous mode the host can do work while the sensor is finishing the sweep. Since the sensor and the host can do work in parallel the update rate of the system will be higher in asynchronous mode. In asynchronous mode the call to acc_service_iq_get_next() / acc_service_iq_get_next_by_reference() will actually acquire the data from the sweep that was started by the previous call to acc_service_iq_get_next() / acc_service_iq_get_next_by_reference().

In synchrounous mode the sensor is garanteed to be idle outside of the acc_service_iq_get_next() / acc_service_iq_get_next_by_reference() calls.

The synchronous mode in combination with streaming Repetition Mode will result in a failure when trying to create the service.

```
void acc_service_asynchronous_measurement_set(acc_service_configuration_t
    configuration, bool asynchronous_measurement);
```

3.5 Creating Service

After the IQ configuration has been prepared and populated with desired configuration parameters, the actual IQ service instance must be created. During the creation step all configuration parameters are validated and the resources needed by RSS are reserved. This means that if the creation step is successful, we can be sure that it is possible to activate the service and get data from the sensor (unless there is some unexpected hardware error).

```
acc_service_handle_t handle = acc_service_create(iq_configuration);

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

During service create, the service run a calibration sequence on the sensor. The calibration is used once at create and can be used until the service is destroyed. A new calibration is needed if the environment is changed, such as deviation in temperature.

If the service handle returned from acc_service_create is equal to NULL, then some setting in the configuration made it impossible for the system to create the service. One common reason is that the requested sweep length is too long or if the calibration fail, but in general, looking for error messages in the log is the best way to find out why a service creation failed.

When the service has been created it is possible to get the actual number of samples (data_length) we will get for each sweep. This value can be useful when allocating buffers for storing the IQ data.

```
acc_service_iq_metadata_t iq_metadata;
acc_service_iq_get_metadata(handle, &iq_metadata);

uint16_t data[iq_metadata.data_length];
```


It is now also possible to activate the service. This means that the radar sensor starts to do measurements.

```
if (!acc_service_activate(handle))
{
    /* Handle error */
}
```

3.6 Reading IQ Data from the Sensor

IQ data is read from the sensor by a call to the function acc_service_iq_get_next. This function blocks until the next sweep arrives from the sensor and the IQ data is then copied to the data array. When using this function, it is up to the application to allocate memory for the result, as can be seen below.

```
float complex                   data[iq_metadata.data_length];
acc_service_iq_result_info_t result_info;

for (int i = 0; i < 10; i++)
{
    if (!acc_service_iq_get_next(handle, data, iq_metadata.data_length, &
        result_info))
    {
        /* Handle error */
    }
}
```

Another way to get the data is to call the function acc_service_iq_get_next_by_reference. This function also blocks until the next sweep arrives from the sensor and the result is available in data. The difference from the function above is that RSS will provide the memory in the resulting data. The length of the data is still provided in iq_metadata.data_length. The memory provided is owned by RSS and should not be freed. The application is however free to manipulate the data until the next call to acc_service_iq_get_next_by_reference. The reason to use this function is the reduced ram usage for the application as well as increased speed for the function call. Note that this function is only compatible with the acc_int16_complex_t output format.

```
acc_int16_complex_t          *data;
acc_service_iq_result_info_t result_info;

for (int i = 0; i < 10; i++)
{
    if (!acc_service_iq_get_next_by_reference(handle, &data, &result_info))
    {
        /* Handle error */
    }
}
```

3.7 Deactivating and Destroying the Service

Call the acc_service_deactivate function to stop measurements.

```
if (!acc_service_deactivate(handle))
{
    /* Handle error */
}
```

After the service has been deactivated it can be activated again to resume measurements or it can be destroyed to free up the resources associated with the service handle.

```
acc_service_destroy(&handle);
```

Finally, call acc_rss_deactivate when the application doesn't need to access the Radar System Software anymore. This releases any remaining resources allocated in RSS.

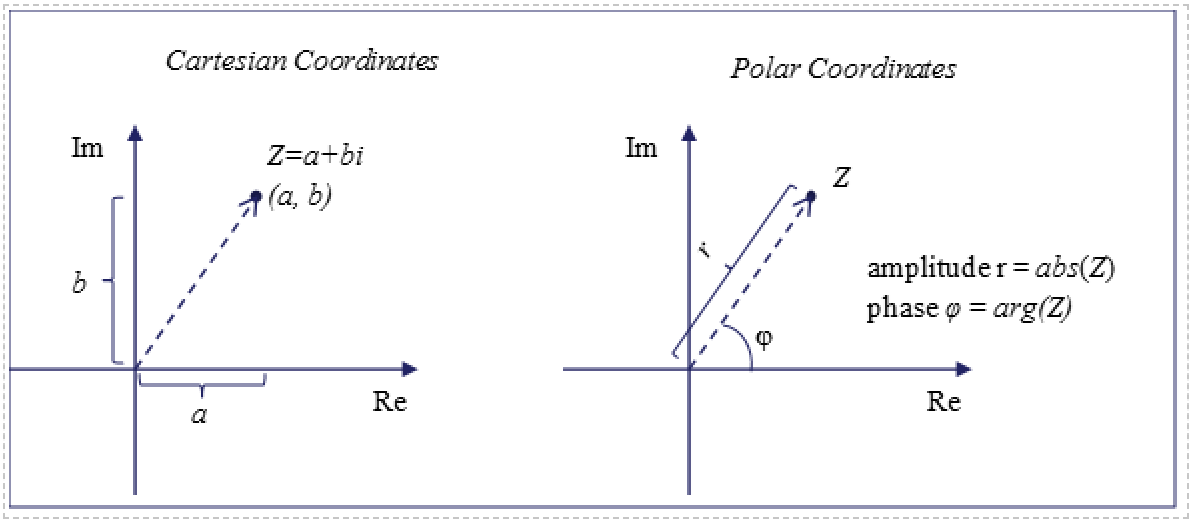
```
acc_rss_deactivate();
```

4 How to Interpret the IQ Data

4.1 Calculating Amplitude and Phase

Each IQ data sample is a complex number consisting of two parts, a real component and an imaginary component. All complex numbers can be written in the form a + bi, where a and b are two ordinary real numbers and i is the imaginary unit that can be thought of having the value sqrt(-1). A complex number z = a + bi is said to have the real part a and the imaginary part b.

Complex numbers can also be seen as points or vectors in the complex plane and be represented in polar coordinates with a radius and an angle. In the context of IQ data, the radius corresponds to the signal amplitude and angle is the phase of the signal.



The Acconeer IQ data API rely on the c99 representation of complex float. Use the functions crealf and cimagf to extract the real and imaginary parts of the complex number.

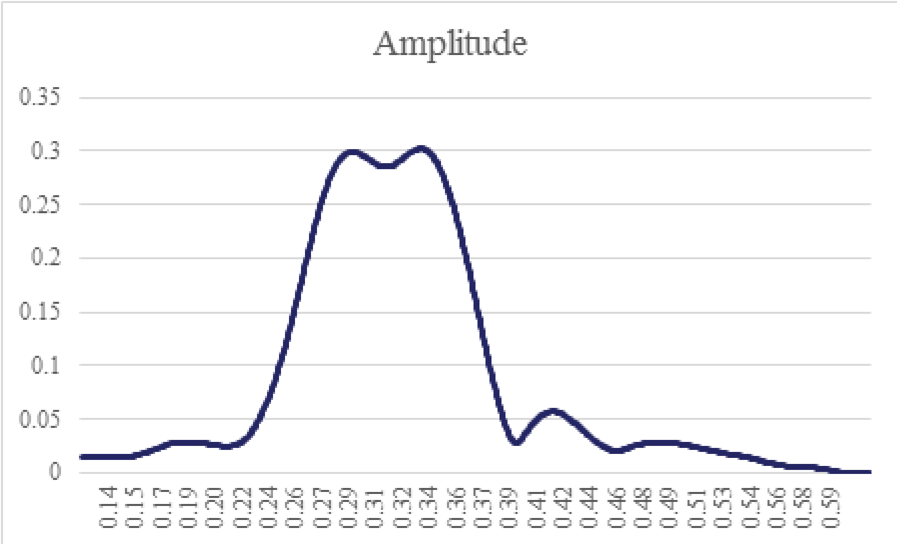
```
#include <complex.h>

float complex z = 2 + 3*I;

float a = crealf(z);
float b = cimagf(z);

float amplitude = cabsf(z); // same as sqrtf(a*a + b*b)
float phase = cargf(z);     // same as atan2f(b,a)
```

The phase difference between two IQ data samples z1 and z2 can be calculated using the expression cargf(z2 * conjf(z1)).

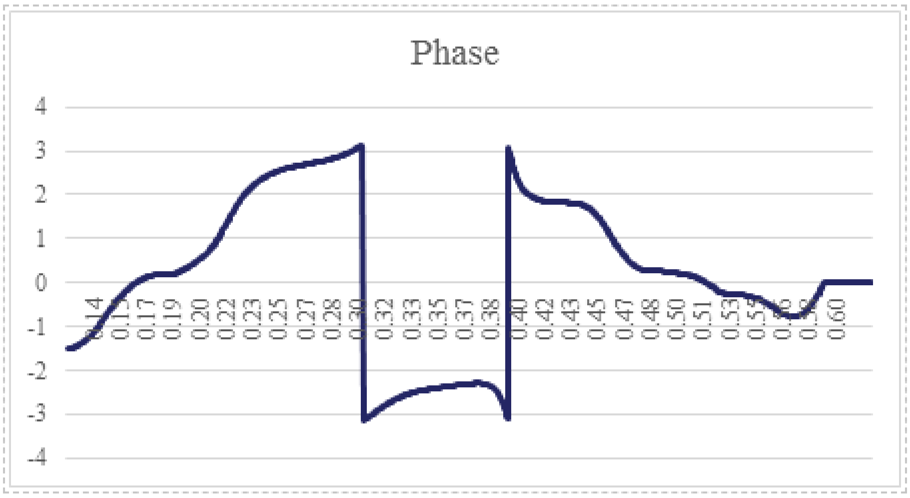
```
float phase_shift = cargf(z2 * conjf(z1));
```

4.2 Plotting Amplitude and Phase

The graphs below show the amplitude and phase response from an object placed about 28 cm from the sensor. In the amplitude graph we can see shadow reflections 6 and 12 cm behind the object. To achieve as stable phase as possible, we are running the A111 radar sensor in a different mode in the IQ Service compared to the Envelope and Power Bins services. The phase gets more stable in the IQ service but as a side effect we are getting shadow reflections behind the object.



Note that in the phase plot below, the signal wraps around from pi to -pi at a distance around 0.30m and then it goes back from -pi to pi a little bit later.



4.3 IQ Metadata

In addition to the array with IQ data samples, a metadata data structure provides side information that can be useful when interpreting the IQ data. This metadata can be retrieved after creating the service. It will not change during operation, so it is only needed to be retrieved once for the created service.

```
acc_service_iq_metadata_t iq_metadata;
acc_service_iq_get_metadata(handle, &iq_metadata);
```

The most important member variable in the meta data structure is data_length which holds the length of the IQ data array. For other member variables see acc_service_iq_metadata_t.

4.4 IQ Result Info

Result info is another kind of metadata which might change for each retrieved result. Result info is provided at the same time as the resulting array, either when calling get_next() or when a callback is triggered.

```
acc_service_iq_result_info_t result_info;
acc_service_iq_get_next(handle, data, data_length, &result_info);
```

The member variables sensor_communication_error and data_quality_warning are intended for continuous monitoring from the application. A true value of sensor_communication_error indicates a hardware-related failure to obtain data from the sensor. The sensor can end up in a state that the service does not recover from. Therefore, it's recommended to destroy the service and create it again if there is a communication error. A true value of data_quality_warning indicates that an internal sensor parameter is outside its interval for normal operation. This issue is likely to occur when the temperature of the sensor has changed and the sensor needs a new calibration. The service performs a calibration when it is created and it is recommended to destroy the service and create it again when an application receive a data_quality_warning.

For other member variables see acc_service_iq_result_info_t.

4.5 Micro Motion Measurement Example

In this example we will implement a simple phase tracking algorithm that can detect micro motions about 25 cm from the sensor. It will look at differences in the phase information between consecutive sweeps and from that calculate how much the object has moved. For each sweep we will look at one sample in the middle of the sweep array so the sweep length can be decreased to a few centimeters. A short sweep range also means that we can run at a high sweep frequency. That is good, because between two sweeps, we can only measure phase differences up to pi radians – which corresponds to object movements of up to 1.25mm.

```
/* Set up the IQ service as described in chapter 2 and 3, use the
   configuration below */

float frequencey = 300;
acc_service_requested_start_set(iq_configuration, 0.2);
acc_service_requested_length_set(iq_configuration, 0.4);
acc_service_repetition_mode_streaming_set(iq_configuration, frequencey);
```

The IQ data sample from the previous sweep is stored in the variable z0 and the sample from the current sweep is stored in z1. The phase difference between the two sweeps is then calculated and we will get the movement between the sweeps by multiplying by wavelength / (4 * pi). The variable acc_dist holds the accumulated distance changes since start of tracking. If the amplitude falls under the threshold the accumulated distance is reset to 0. Note that we are tracking relative movements about 25 cm from the sensor, wo do not measure any absolute distances in this example.

```
const float wavelength = 5.0; // wavelength in mm
const float pi = 3.14159265359;
const float amplitude_threshold = 0.1;
float complex z0 = 0;
float acc_dist = 0;
bool status;

while (true) {
    status = acc_service_iq_get_next(iq_handle, iq_data, iq_metadata.
       data_length,
                                     &result_info);
    if (!status) {
        /* handle error */
    }

    float complex z1 = iq_data[iq_metadata.data_length/2];

    if (cabsf(z1) > amplitude_threshold) {
        if( z0 != 0) {
            float delta_dist = cargf(z1 * conjf(z0)) * wavelength / (4 * pi)
               ;
            acc_dist += delta_dist;
            printf("delta distance % 0.2f mm, accumulated distance % 0.2f mm
               , speed = "
                   "% 0.2f m/s\n", delta_dist, acc_dist, delta_dist *
                      frequencey /1000);
        }
        z0 = z1;
    } else if (z0 != 0) {
        printf("no object detected, resetting tracking\n");
        z0 = 0;
        acc_dist = 0;
    }
}
```

5 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.