



# Power Bins Service

User Guide



Power Bins Service

User Guide

Author: Acconeer AB

Version:a111-v2.15.2

Acconeer AB August 18, 2023



**Contents**

- 1 Power Bins Service 3**
  - 1.1 Disclaimer . . . . . 4
- 2 Memory Requirements 4**
  - 2.1 Flash footprint . . . . . 4
  - 2.2 Heap usage . . . . . 4
- 3 Setting up the Service 4**
  - 3.1 Initializing the System . . . . . 4
  - 3.2 RSS Configuration . . . . . 5
  - 3.3 Service API . . . . . 5
  - 3.4 Power Bins Service Configuration . . . . . 5
    - 3.4.1 Configuration Summary . . . . . 5
    - 3.4.2 Profiles . . . . . 6
    - 3.4.3 Repetition Mode . . . . . 6
    - 3.4.4 Downsampling Factor . . . . . 7
    - 3.4.5 Hardware Accelerated Average Samples (HWAAS) . . . . . 7
    - 3.4.6 Power Save Mode . . . . . 7
    - 3.4.7 Asynchronous Measurement . . . . . 8
    - 3.4.8 Maximum Unambiguous Range (mur) . . . . . 9
  - 3.5 Creating Service . . . . . 9
  - 3.6 Reading Power Bins Data from the Sensor . . . . . 10
  - 3.7 Deactivating and Destroying the Service . . . . . 10
- 4 How to Interpret the Power Bins Data 11**
  - 4.1 Power Bins Metadata . . . . . 11
  - 4.2 Power Bins Result Info . . . . . 11
- 5 Object Detection Example 12**
- 6 Disclaimer 15**

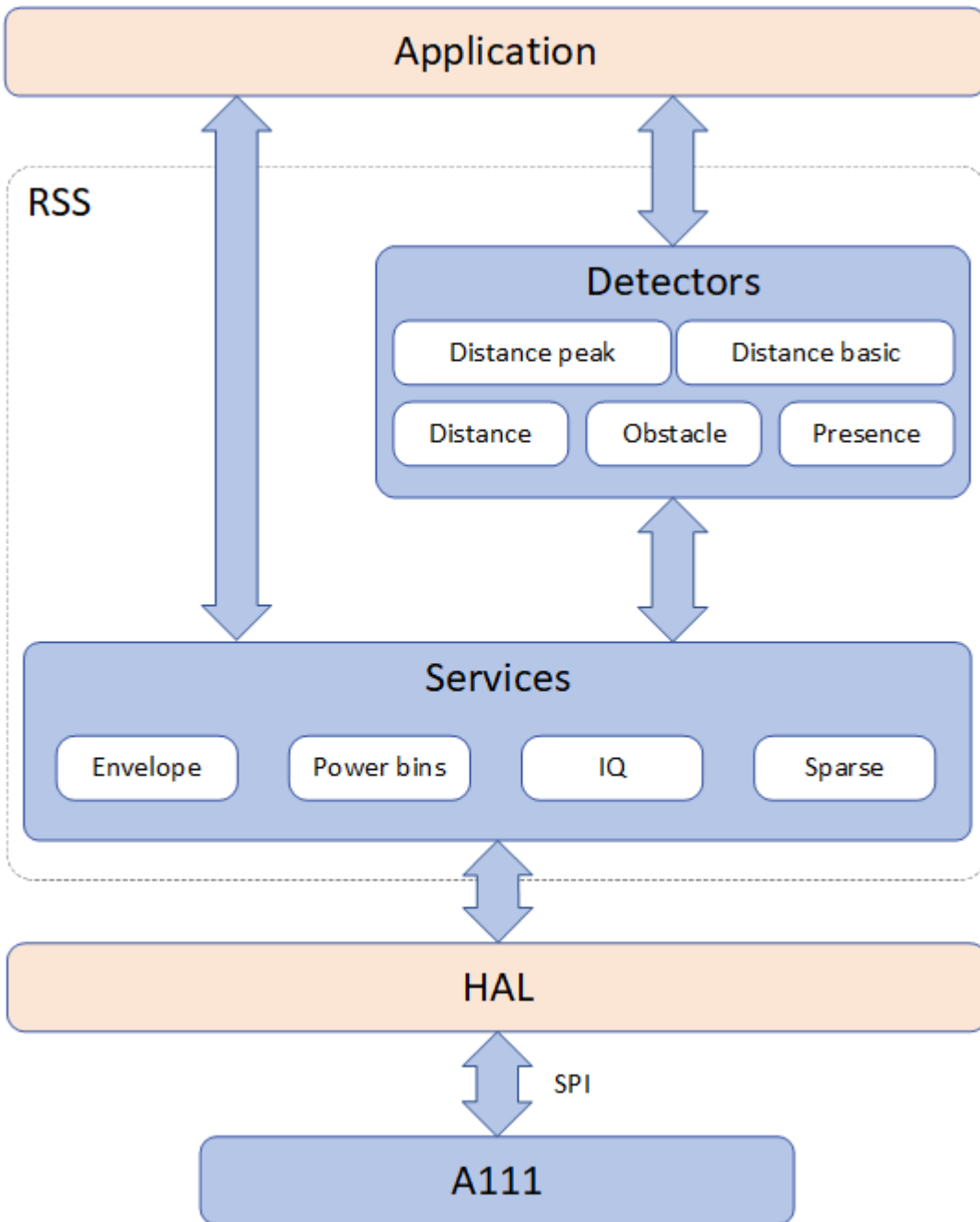


## 1 Power Bins Service

The Power Bins Service is one of four services that provide an interface for reading out the radar signal from the Acconeer A111 sensor. The Power Bins Service is mainly intended for use in low power applications where large objects are measured at short distances from the radar sensor e.g. in parking sensors mounted on the ground.

The data provided by the Power Bins API is essentially subsampled envelope data. However, the Power Bins values are calculated using an algorithm optimized for low computational complexity. This algorithm will not remove as much noise from the signal as the more advanced signal processing algorithms used in the Envelope and IQ Data Services.

For use cases where weaker radar echoes are measured or when the resolution of the signal is important, we recommend using the Envelope or IQ data service instead.



Acconeer provide an example of how to use the Power Bins service: `example_service_power_bins.c`

For more details on the Power Bins 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.

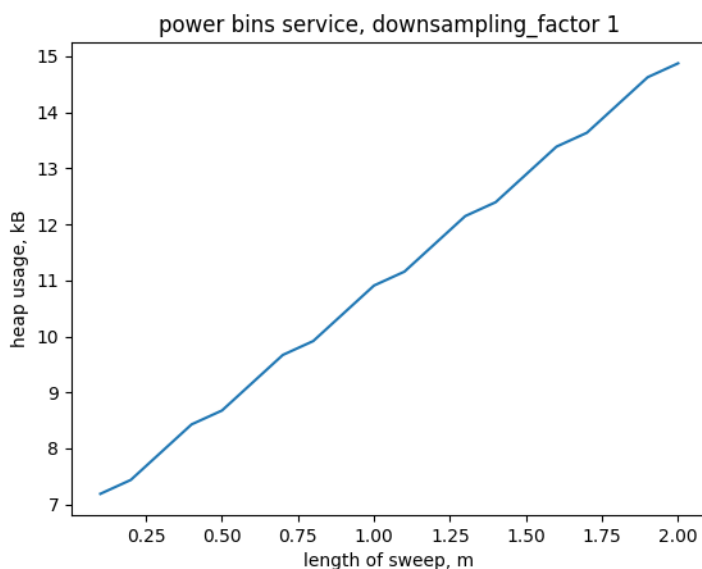
## 2 Memory Requirements

### 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 Power Bins service requires 85 kB flash when built in STM32Cube environment.

### 2.2 Heap usage

The amount of heap used by the Power Bins 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_power_bins_get_next()` if it wants to save the envelope data in a separate buffer or `acc_service_power_bins_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_power_bins_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 to the radar sensor service API. 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 Power Bins Service Configuration

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

```
acc_service_configuration_t power_bins_configuration =
    acc_service_power_bins_configuration_create();

if (power_bins_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_power_bins.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 Power Bins 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

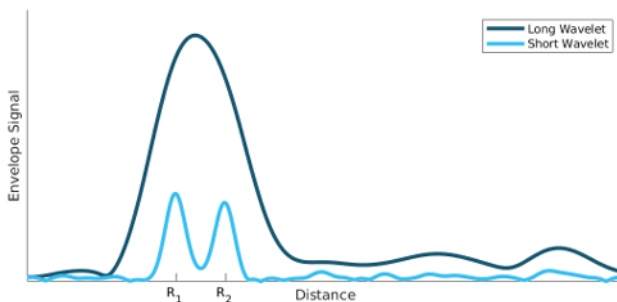
### Power Bins Specific Configurations :

Parameter	Description	Type	Unit	Limits
Downsampling Factor	See below	integer	N/A	[1, 2, 4]
Bin Count	Number of bins in the data	integer	N/A	> Length/0.016
Noise Level Normalization	Flag to determine if data should be normalized with noise floor	bool	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 Power Bins 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.

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(acc_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 if the application set a length which requires stitching.

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

### 3.4.4 Downsampling Factor

In the Power Bins service, the base step length is  $\sim 0.5$ mm. 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  $\sim 2$ mm. Less data require less processing and could be useful in applications which require low power consumption. The Power Bins service supports a downsampling factor of 1, 2, or 4.

```
void acc_service_power_bins_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_power_bins_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 Power Bins 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 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_power_bins_get_next()` / `acc_service_power_bins_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_power_bins_get_next()` / `acc_service_power_bins_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_power_bins_get_next()` / `acc_service_power_bins_get_next_by_reference()` will actually acquire the data from the sweep that was started by the previous call to `acc_service_power_bins_get_next()` / `acc_service_power_bins_get_next_by_reference()`.

In synchronous mode the sensor is guaranteed to be idle outside of the `acc_service_power_bins_get_next()` / `acc_service_power_bins_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);
```

### 3.4.8 Maximum Unambiguous Range (mur)

Sets the maximum unambiguous range (MUR), which in turn sets the maximum measurable distance (MMD).

The MMD is the maximum value for the range end, i.e., the range start + length. The MMD is smaller than the MUR due to hardware limitations.

The MUR is the maximum distance at which an object can be located to guarantee that its reflection corresponds to the most recent transmitted pulse. Objects farther away than the MUR may fold into the measured range. For example, with a MUR of 10 m, an object at 12 m could become visible at 2 m.

A higher setting gives a larger MUR/MMD, but comes at a cost of increasing the measurement time for a sweep. The measurement time is approximately proportional to the MUR.

This setting changes the pulse repetition frequency (PRF) of the radar system. The relation between PRF and MUR is  $MUR = c / (2 * PRF)$  where  $c$  is the speed of light.

Setting	MUR	MMD	PRF
ACC_SERVICE_MUR_6	11.5 m	7.0 m	13.0 MHz
ACC_SERVICE_MUR_9	17.3 m	12.7 m	8.7 MHz

This is an experimental feature.

```
void acc_service_mur_set(acc_service_configuration_t service_configuration,
      acc_service_mur_t max_unambiguous_range);
```

## 3.5 Creating Service

After the Power Bins configuration has been prepared and populated with desired configuration parameters, the actual Power Bins 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(power_bins_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 Power Bins data.

```
acc_service_power_bins_metadata_t power_bins_metadata;
acc_service_power_bins_get_metadata(handle, &power_bins_metadata);

uint16_t data[power_bins_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 Power Bins Data from the Sensor

Power Bins data is read from the sensor by a call to the function `acc_service_power_bins_get_next`. This function blocks until the next sweep arrives from the sensor and the result is available in `data`. When using this function, it is up to the application to allocate memory for the result, as can be seen below.

```
uint16_t data[power_bins_metadata.data_length];
acc_service_power_bins_result_info_t result_info;

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

Another way to get the data is to call the function `acc_service_power_bins_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 `power_bins_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_power_bins_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.

```
uint16_t *data;
acc_service_power_bins_result_info_t result_info;

for (int i = 0; i < 10; i++)
{
    if (!acc_service_power_bins_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.



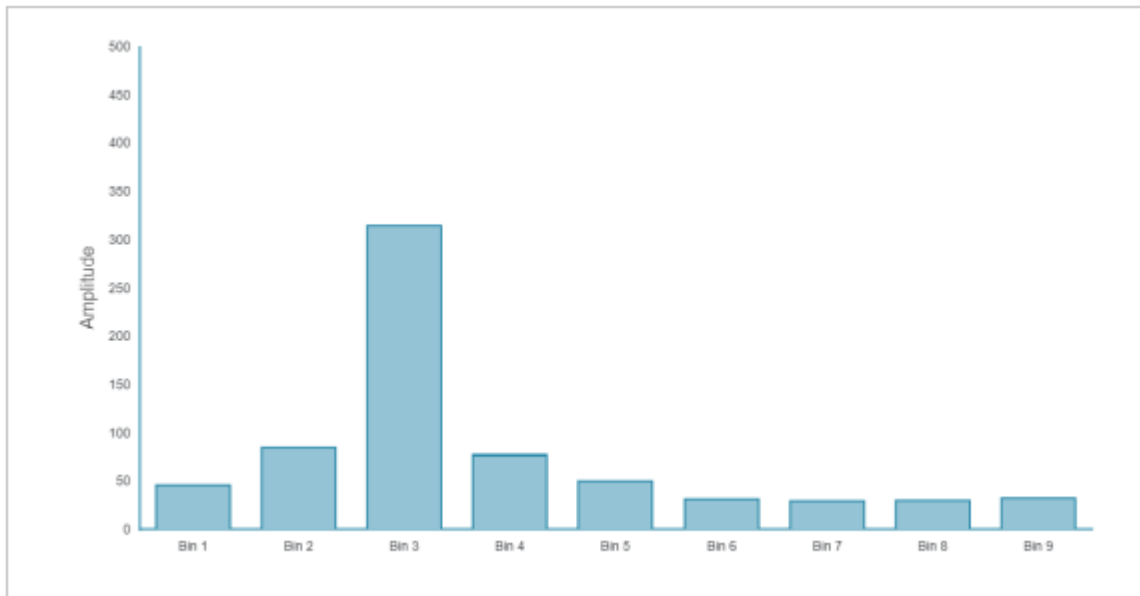
```
acc_service_destroy(&handle);
```

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 Power Bins Data

Each data point in a Power Bins data sweep contains the average amplitude in a short interval. In the example below, we are measuring the interval 0.15 m to 0.50 m using 9 bins. The interval is divided into 9 sub intervals and for each subinterval the average amplitude is calculated. The first subinterval (bin 1) contains the average amplitude for the interval 0.15 m to approximately 0.19 m and the last subinterval (bin 9) contains the average amplitude for the interval from approximately 0.46m to 0.5 m. The object is placed at roughly 24 cm and gives the highest amplitude in bin 3.



### 4.1 Power Bins Metadata

In addition to the array with Power Bins data samples, a metadata data structure provides side information that can be useful when interpreting the Power Bins 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_power_bins_metadata_t power_bins_metadata;  
acc_service_power_bins_get_metadata(handle, &power_bins_metadata);
```

The most important member variable in the meta data struct is `bin_count`. It contains the number of entries in the Power Bins data array. For other member variables see `acc_service_power_bins_metadata_t`.

### 4.2 Power Bins 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_power_bins_result_info_t result_info;  
acc_service_power_bins_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_power_bins_result_info_t`.



## 5 Object Detection Example

In the following example we will show how the Power Bins API can be used for detecting if an object is present in front of the sensor. The basic idea is to detect if the signal amplitude in any of the Power Bin data values are exceeding a predefined threshold. The measured amplitude can differ between different sensor samples. This can be problematic in applications like this where we compare the amplitude to a predefined threshold. To reduce the impact of the problem we first record the noise level. We then use the average noise level to normalize the measured values when transmission is enabled. By using this trick, we can compensate for gain variations in the receiver chain of the radar sensor.

First code example show how to measure the noise level for the sensor.

```
acc_service_configuration_t configuration;
acc_service_handle_t handle;

configuration = acc_service_power_bins_configuration_create();

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

acc_service_power_bins_requested_bin_count_set(configuration, 10);

// Update configuration according to project needs
// Disable radio transmitter to only capture noise
acc_service_tx_disable_set(configuration, true);

handle = acc_service_create(configuration);

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

acc_service_power_bins_configuration_destroy(&configuration);

handle = setup_power_bins(disable_transmission);
acc_service_power_bins_get_metadata(handle, &power_bins_metadata);

float power_bins_data[power_bins_metadata.bin_count];
float sum = 0;
const int n = 10;

bool status = acc_service_activate(handle);

for (int i=0 ; i<n ; i++) {
    status = acc_service_power_bins_get_next(handle,
                                             power_bins_data,
                                             power_bins_metadata.bin_count,
                                             &result_info);

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

    for (int j=0 ; j < power_bins_metadata.bin_count ; j++) {
        sum += power_bins_data[j];
    }
}

status = acc_service_deactivate(handle);
```



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

acc_service_destroy(&handle);

float noise_level = sum / (n * power_bins_metadata.bin_count);
```

The second part detect if an object is present in front of the sensor. It uses the noise level from previous code example as a normalization factor

```
acc_service_configuration_t configuration;
acc_service_handle_t handle;
bool object_present = false;

configuration = acc_service_power_bins_configuration_create();

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

acc_service_power_bins_requested_bin_count_set(configuration, 10);

// Update configuration according to project needs

handle = acc_service_create(configuration);

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

acc_service_power_bins_configuration_destroy(&configuration);

handle = setup_power_bins(disable_transmission);
acc_service_power_bins_get_metadata(handle, &power_bins_metadata);

float power_bins_data[power_bins_metadata.bin_count];
float sum = 0;
const int n = 10;

bool status = acc_service_activate(handle);

for (int i=0 ; i<n ; i++) {
    status = acc_service_power_bins_get_next(handle,
                                             power_bins_data,
                                             power_bins_metadata.bin_count,
                                             &result_info);

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

    for (int j=0 ; j < power_bins_metadata.bin_count ; j++) {
        if (radar_data[i] / noise_level > threshold){
            object_present = true;
        }
    }
}

status = acc_service_deactivate(handle);
```



```
if (!status) {  
    /* Handle error */  
}  
  
acc_service_destroy(&handle);
```



## 6 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.

