Sparse Service

User Guide

Sparse Service

User Guide

Author: Acconeer AB

Version:v2.0.0

Acconeer AB December 2, 2019

Contents

1 Sparse Service

The Sparse Service is one of four services that provides an interface for reading out the radar signal from the Acconeer A111 sensor. The data returned from the Sparse service can be used in different types of algorithms such as motion detection algorithms and presence detection. For basic distance measurements, Envelope service is a good starting point and the advanced users that needs phase information for detecting very small variations in distance will probably prefer the IQ data service instead of the Sparse service.

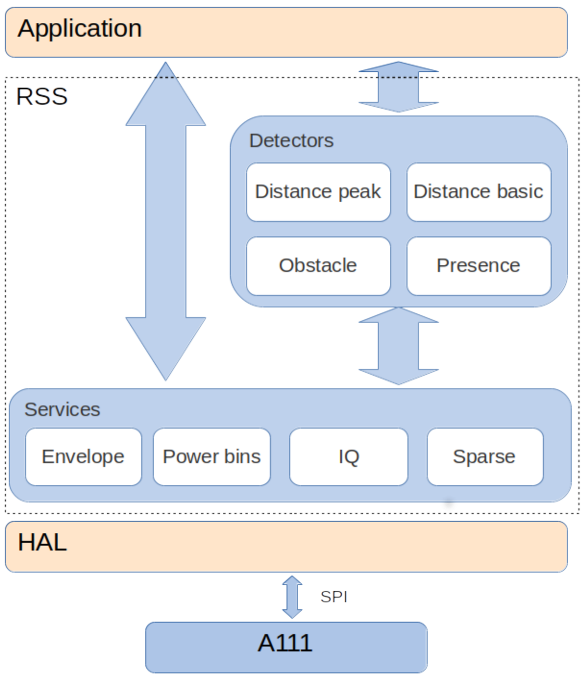

Figure 1:

Acconeer also provide several easy to use detectors that are implemented on top of the basic data services. The detectors provide APIs for higher level tasks. A detector utilizing Sparse service could be used for presence detection.

Acconeer provide an example of how to use the Sparse service: example_service_sparse.c

The Sparse service is fundamentally different from the other services. Instead of sampling the reflected pulse several times per wavelength (which is ˜5 mm), the pulse is sampled roughly every 60 mm. As such, the Sparse service should not be used to measure the reflections of static objects. Instead, the Sparse service produces a sequence of measurements of the sparsely located sampling points. This allows for detection of any movement, small or large, occurring in front of the sensor at the configured sampling points.

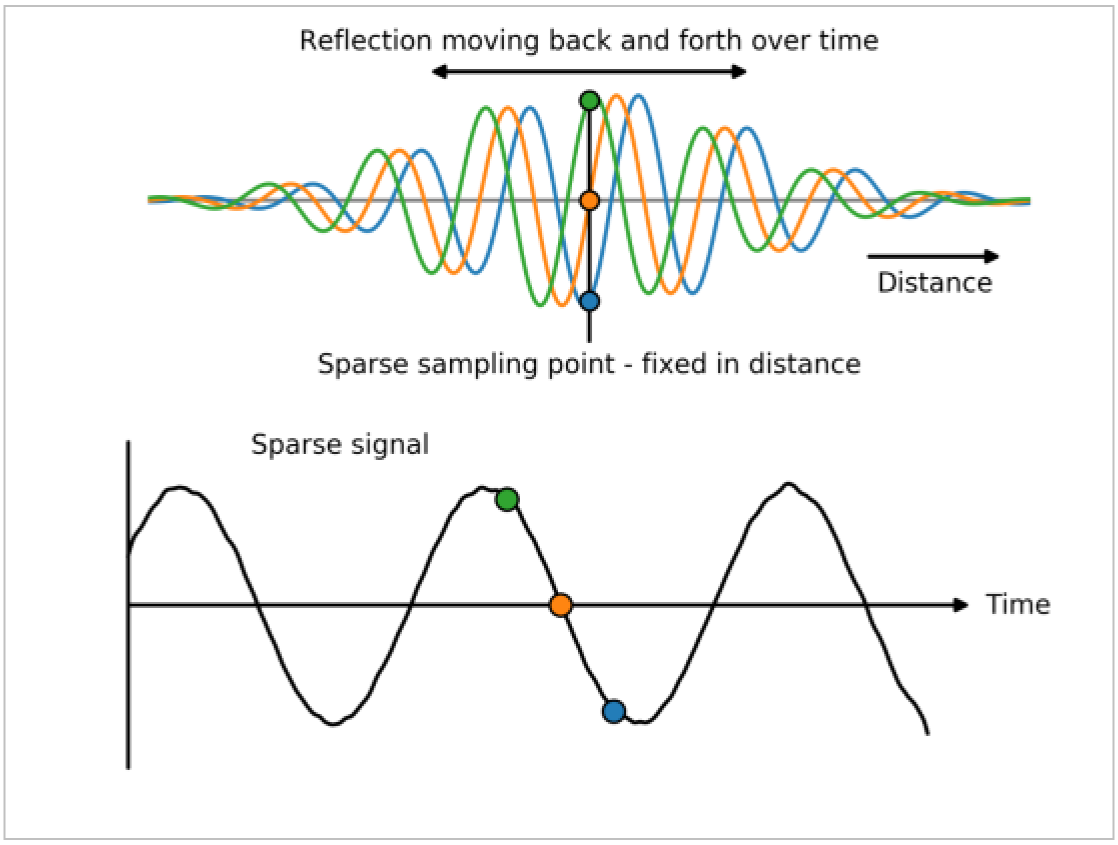
Reflection moving back and forth over time
Distance
Sparse sampling point - fixed in distance
Sparse signal
Time

Figure 2:

The above image illustrates how a single sparse point samples a reflected pulse from a moving target. Due to the movements of the target, the signal from the Sparse service varies over time. If the object is static, the signal too will be static, but not necessarily zero.

Every data frame from the Sparse service consists of a configurable number of sweeps, which are sampled immediately after each other. Every sweep consists of one or several points sampled in space as configured. The sweeps are, for many applications, sampled closely enough in time that they can be regarded as being sampled simultaneously. In such applications, the sweeps can be averaged for optimal SNR. Note that unlike the other services, there is no pre-processing applied to the data. As pre-processing is not needed, this service is relatively computationally inexpensive.

The Sparse service is ideal for motion-sensing applications requiring low power consumption. Often, less processing is needed as the Sparse service outputs a lot less data than the Envelope or IQ service.

For more details on the Sparse data it is recommended to use our exploration tool. Check it out on github, https://github.com/acconeer/acconeer-python-exploration.

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 Setting up the Service

2.1 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 to 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.

```
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 driver layer is first initialized by calling acc_driver_hal_init. The hal struct is then obtained with the function acc_driver_hal_get_implementation.

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

hal = acc_driver_hal_get_implementation();

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

2.2 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.

2.3 Sparse Service Configuration

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

```
acc_service_configuration_t sparse_configuration =
   acc_service_sparse_configuration_create();

if (sparse_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_sparse.h and acc_base_configuration.h for a complete description of configuration parameters.

2.3.1 Profiles

The services and detectors support profiles with different configuration of emission in the sensor. The different profiles provide an option to configure the wavelet length and optimize on either depth resolution or radar loop gain. More information regarding profiles can be read in the sensor introduction document, https://acconeer-python-exploration.readthedocs.io/en/latest/sensor_introduction.html.

The Sparse service supports 5 different profiles which are defined in acc_service.h. Profile 1 has the shortest wavelet and should be used in applications which aim to see multiple objects or with short distance to the object. Profiles with higher numbers hve longer wavelet 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);
```

2.3.2 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_base_configuration_repetition_mode_on_demand_set(
   acc_base_configuration_t 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 a fft. This mode can not be used it the application set a length which requires stitching.

```
void acc_base_configuration_repetition_mode_streaming_set(
   acc_base_configuration_t configuration, float update_rate);
```

2.4 Creating Service

After the Sparse configuration has been prepared and populated with desired configuration parameters, the actual Sparse 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(sparse_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 frame. This value can be useful when allocating buffers for storing the Sparse data.

```
acc_service_sparse_metadata_t sparse_metadata;
acc_service_sparse_get_metadata(handle, &sparse_metadata);

uint16_t data[sparse_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 */
}
```

2.5 Reading Sparse Data from the Sensor

Sparse data is read from the sensor by a call to the function acc_service_sparse_get_next. This function blocks until the next sweep arrives from the sensor and the Sparse data is then copied to the Sparse data array.

```
uint16_t                           data[sparse_metadata.data_length];
acc_service_sparse_result_info_t result_info;

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

2.6 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();
```

3 How to Interpret the Sparse Data

The data frame from the Sparse Service should be interpreted as a one-dimensional array with the number of sweeps stored consecutively. The first datapoint in each sweep corresponds radar signal at distance start_m (see Sect. 6.1) and the last data point to distance start_m + length_m. The data points between correspond to the radar signal between these to distances, sampled approximately 60 mm apart and exact number is given by metadata, step_length.

The data format is unsigned 16 bits integer and the data is centered around approximately 32000, but the center level can differ slightly between sensors.

The sweeps are acquired during a relatively short time, so for objects moving with low speed, such as humans sitting or standing, the sweeps should be very similar, deviating from each other only by sensor noise.

In the figure below the data in a data frame from the Sparse service is plotted. The scanned range is 1 meter resulting in 17 data points (1 meter divided by 60 mm is rounded to 17). The range is scanned 16 times resulting in 272 data points. So, index 0, 17, 34, 51, etc. corresponds to measurement of the closest distance, i.e. "start", and index 16, 33, 50, 67, etc. corresponds to measurements of the farthest distance, i.e. "start + length". The points in between correspond to measurements of the point in between in range, similar to the IQ and Envelope service.

In the center of the range a reflector is present which give rise to the deviation from the mean data level at the center of each sweep. Also, the difference between each sweep in the data frame is very low, suggesting a stationary reflector.

Below is an example Sparse data frame. Here, each sweep consists of 17 measured distances and there is 16 sweeps in the data frame. The first data point in each sweep is marked in light blue.

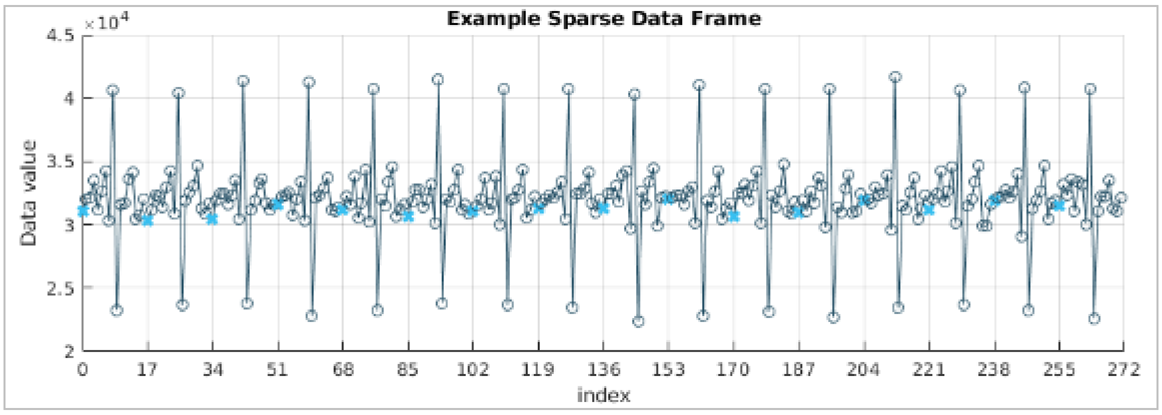

Figure 3:

3.1 Sparse Metadata

In addition to the array with Sparse data samples, a metadata data structure provides side information that can be useful when interpreting the Sparse 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_sparse_metadata_t sparse_metadata;
acc_service_sparse_get_metadata(handle, &sparse_metadata);
```

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

3.2 Sparse 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_sparse_result_info_t result_info;
acc_service_sparse_get_next(handle, data, data_length, &result_info);
```

The most important member variable is the sensor_communication_error which indicates whether a sensor communication has occurred. For other member variables see acc_service_sparse_result_info_t

4 Disclaimer