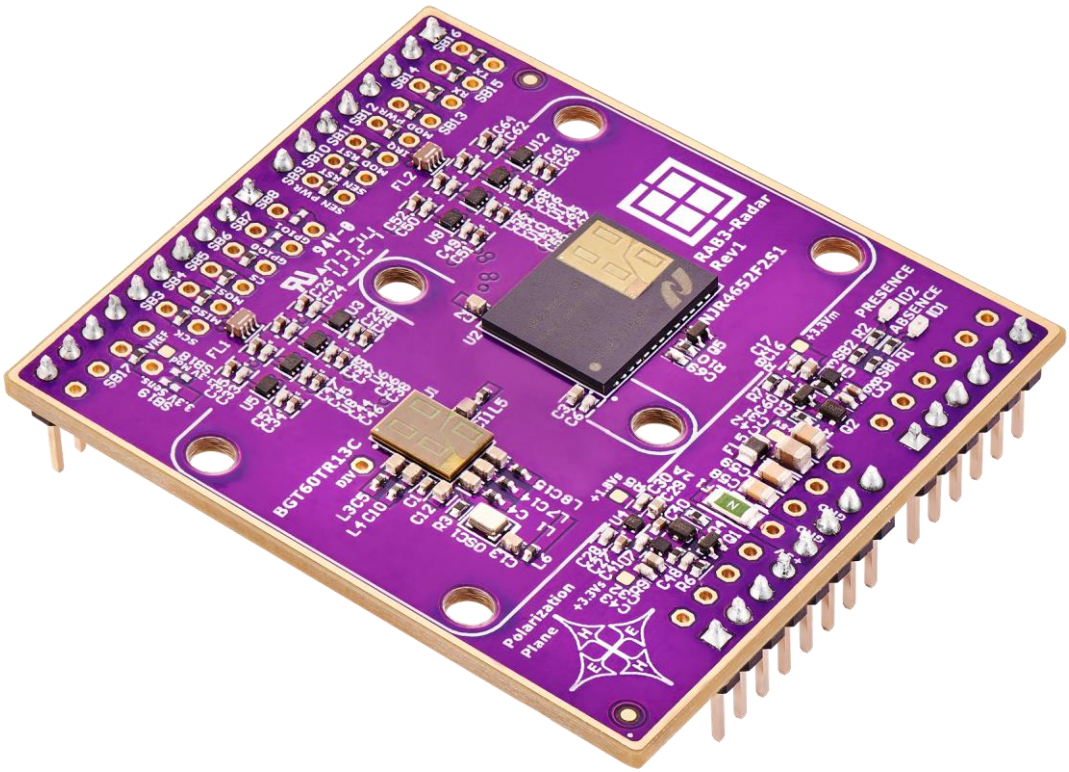
Signal processing
in presence detection demostrator

Application note



Versions

Version	Date	Rationale
1.0	August 22, 2024	First release. Author: ROJ, KOA

Legal disclaimer

The evaluation board is for testing purposes only and, because it has limited functions and limited resilience, is not suitable for permanent use under real conditions. If the evaluation board is nevertheless used under real conditions, this is done at one's responsibility;
any liability of Rutronik is insofar excluded.

Table of contents

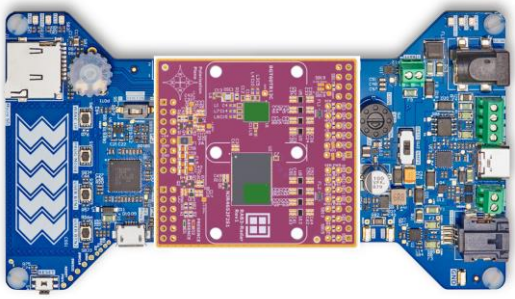
Introducing the demonstrator

This application note describes how the signal is processed “inside” a presence detection demonstrator by Rutronik. This document focuses on the processing algorithm and its C-code implementation.

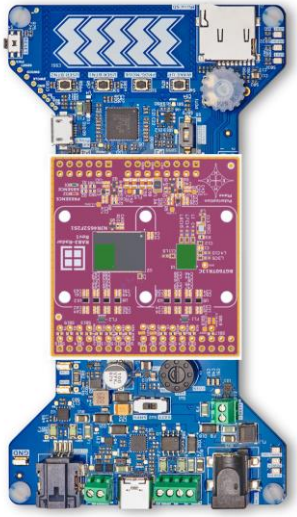
The demonstrator includes RAB3-Radar and RDK2 boards provided by Rutronik. It is intended to detect a motion in front of the radar sensor within the range of 10 meters (human walking). When detecting the motion, the sensor will deliver the distance and the azimuth (angle) of the motion.

For the digital signal processing part, the demonstrator is using a BGT60TR13C sensor of RAB3-Radar and the RDK2.

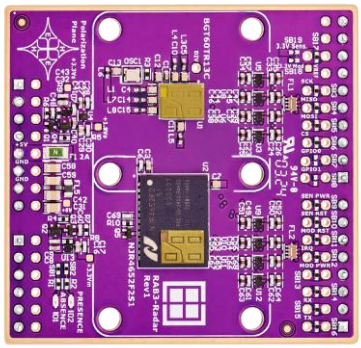
The demonstrator should be mounted horizontally since it uses antennas 1 and 3 of the BGT60TR13C to detect the azimuth.

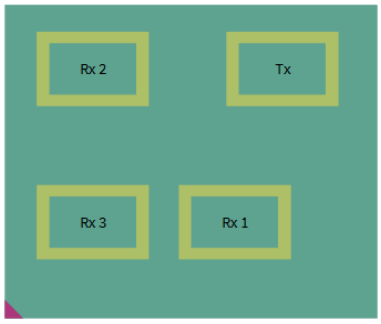


RDK2/Radar is mounted correctly



RDK2/Radar is mounted incorrectly

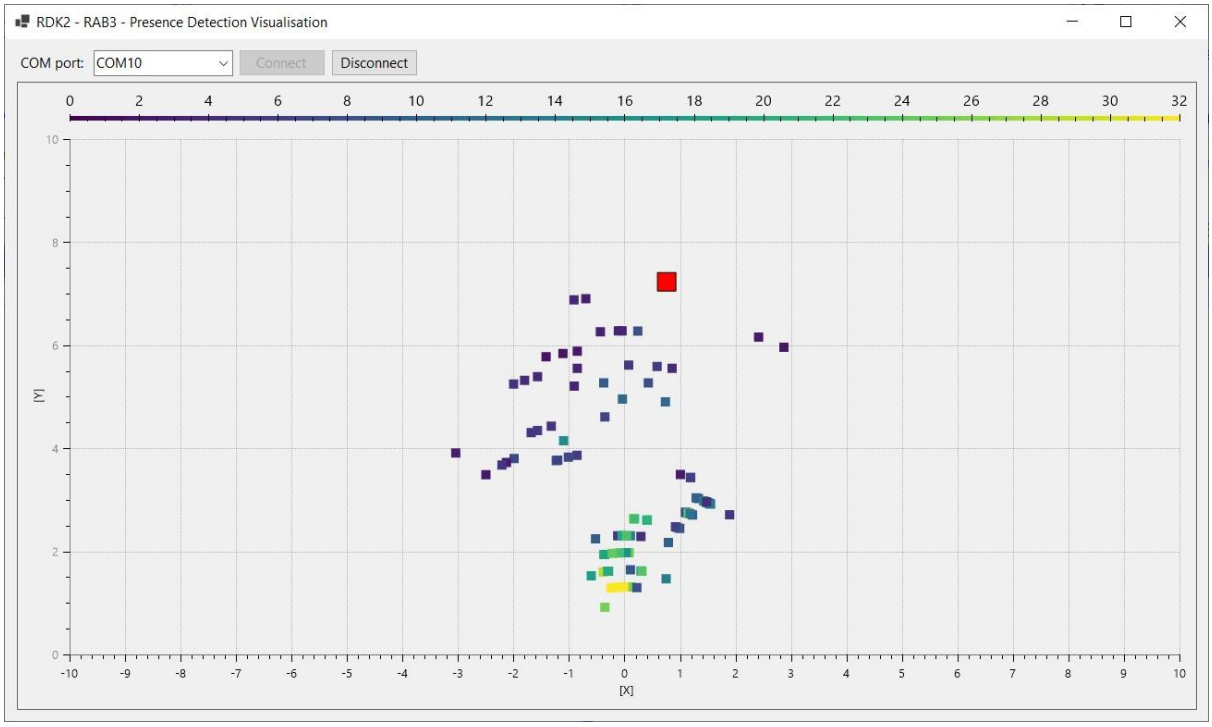




Algorithm is using Rx3 and Rx1 to determine the azimuth of the detected motion

It´s recommended to mount the demonstrator at a minimum height of 1.5 meters to avoid reflection from ground/ceiling which might generate ghost motions.

The RDK2 communicates over USB with a Windows GUI and displays in real time the position of the detected motion (big red square is the last detected motion).



Problems that need to be solved (enhancements for the next version):

- In case of the fixed object with big cross section (walls, …), when a person moves between the sensor and the wall and then leaves the field of view, the wall will be detected as a moving target object. This can be solved in the next phase by tracking the fixed targets using low pass filter for example.
- The angle of the detected motion can be inaccurate at a bigger distance (around 10 meters). Filtering the value can be a solution. Using the variation of the phase within a frame as a quality indicator can also be a good option.
- Display the velocity of the detected motion in the GUI (since we compute the doppler FFT, we know the velocity).

Signal processing

Radar basics

BGT60TR13C radar includes one transmitting antenna Tx and three receiving antennas Rx (for simplicity, we will sometimes show and mention the receiving antennas as a single element). The transmitter sends the electromagnetic waves, they are reflected by the target object and the receiver measures the difference between the transmitted and reflected signal frequencies.

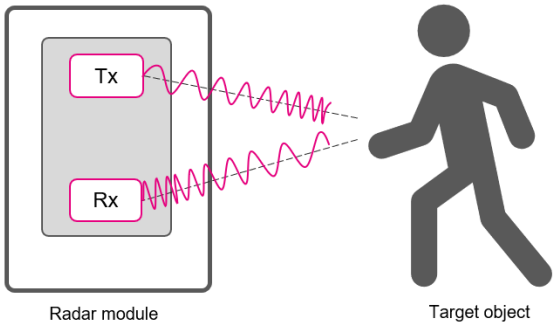


Figure 1. Radar principle

Using the radar data, we can “extract” the motion parameters:

- range d between radar and target,
- velocity v of movement of an object,
- angle α determining the position of an object relative to the radar.

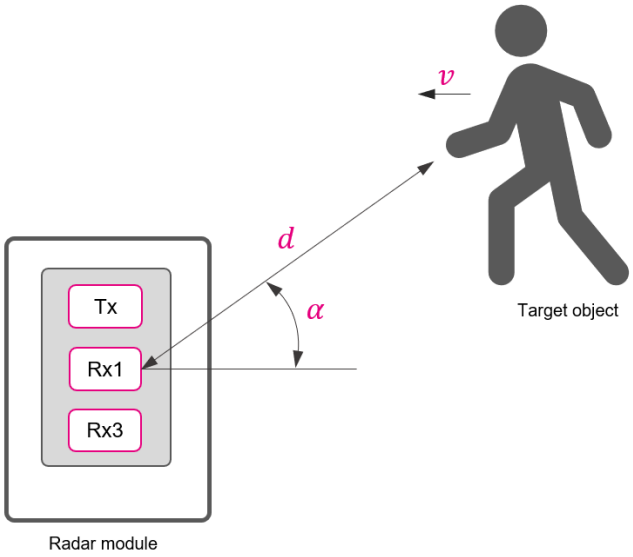


Figure 2. Motion parameters

BGT60TR13C works in a frequency modulated continuous wave mode. In an FMCW mode, the frequency of the electromagnetic wave is increased linearly with time, and the amplitude has a sinusoidal form. This kind of signal is called a chirp.

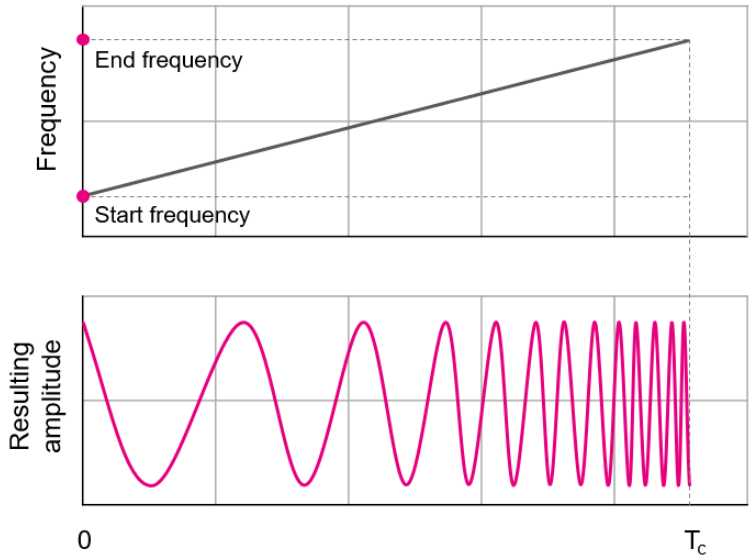


Figure 3. Frequency and amlitude in FMCW mode

N chirps create the frame, i.e. a defined period of time when the chirps are transmitted. A time interval between the chirps is called chirp repetition time (Tc).

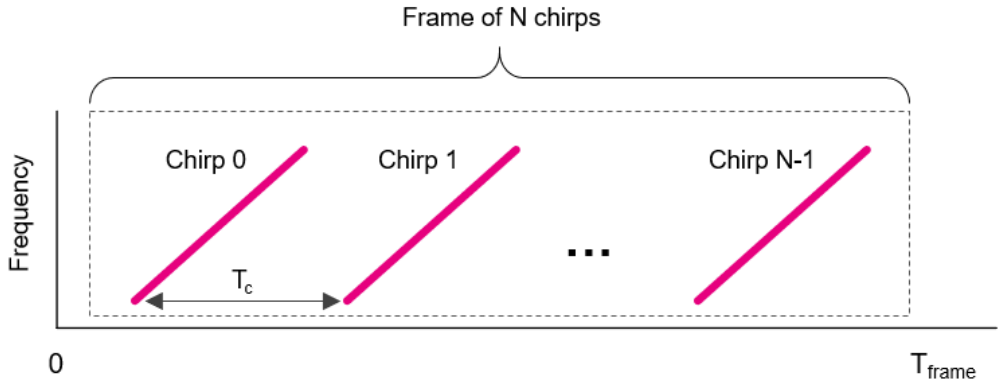


Figure 4. A frame of N chirps

Mixer is another important component of the radar module. It produces an intermediate frequency signal by mixing the transmitted signal (Tx antenna) with the reflected signal (Rx antenna).

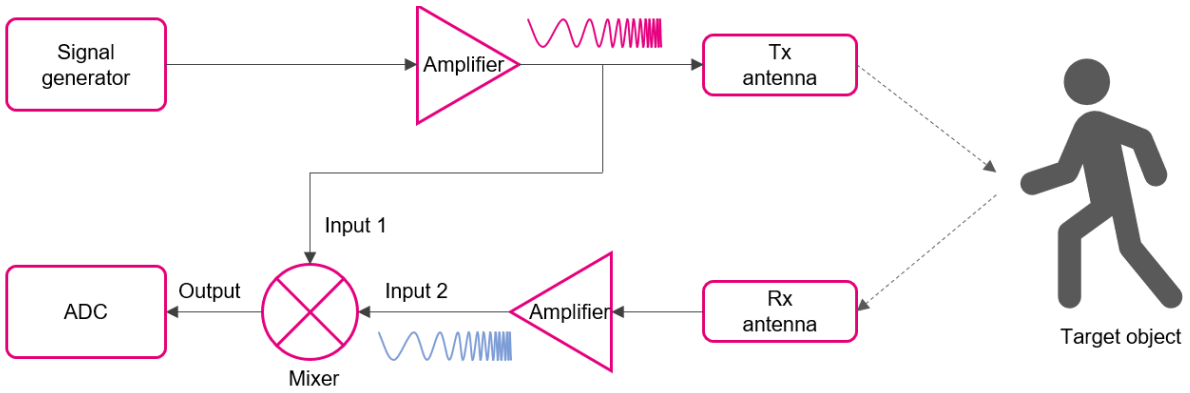


Figure 5. Radar block diagram

The mixer enables, when two delayed chirps are mixed, to generate a constant frequency at the output. At the end, it takes the inputs and highlights the differences between them.

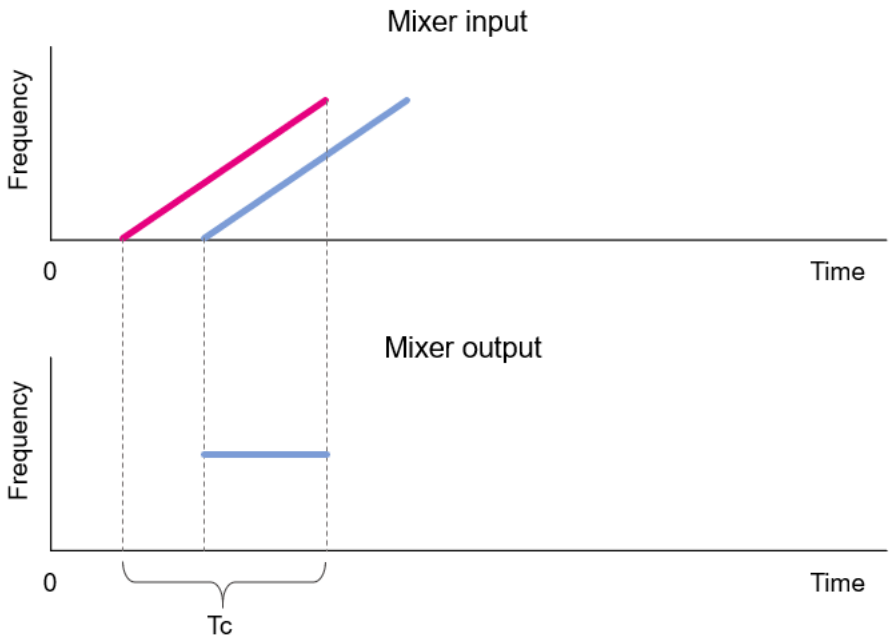


Figure 6. Mixer input and output

Summary

A time delay between the transmitted and received signals allows us to calculate the range d.

The frame (made of multiple chirps at defined time interval) is used to compute the velocity v of a target motion.

The phase difference between the measurements of multiple receiving antennas allows us to calculate the angle α of the target.

Check out next chapters to learn more about the calculation of motion parameters.

Raw data from the radar

Infineon´s Radar Fusion GUI gives you an overview of how the raw data generated by BGT60TR13C look like (**Time Domain** tab).

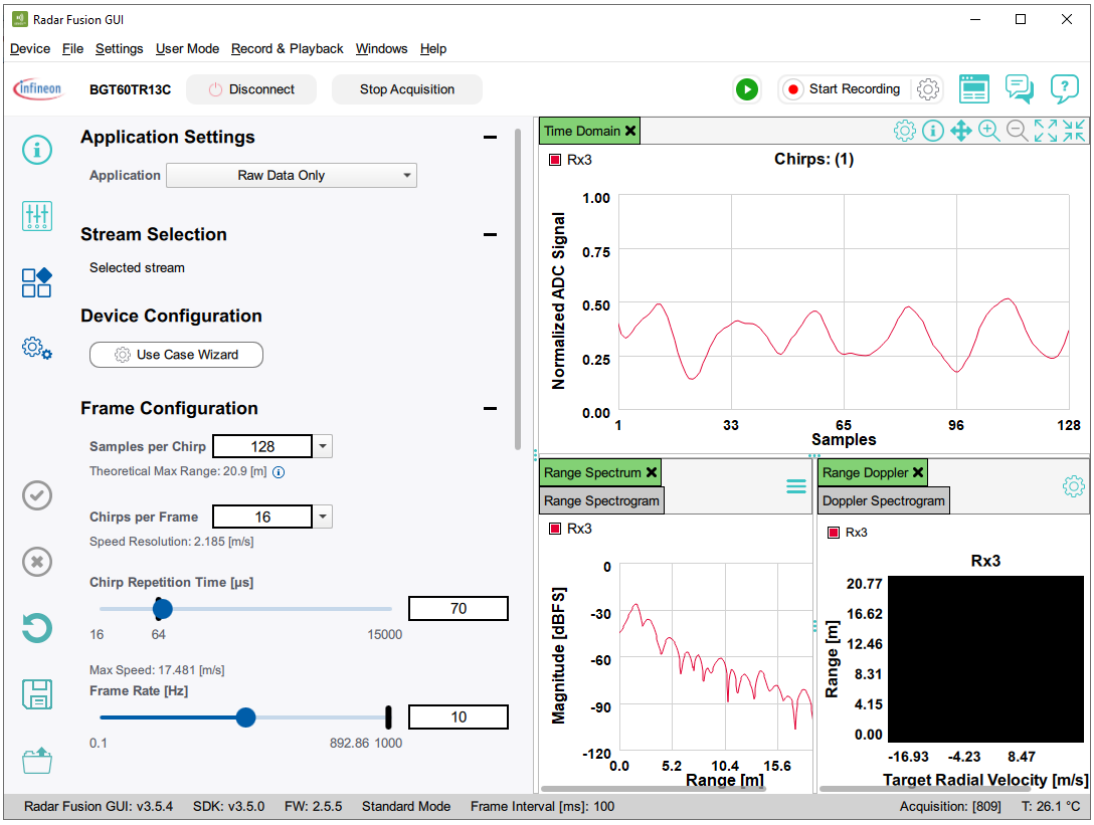


Figure 7. Infineon´s Radar Fusion GUI

By default, the tab only displays the ADC data for the first chirp of the frame. You can switch to display all the “chirps” of the current frame.

In C code, you can read those data by calling the function (the “data” buffer will contain all the ADC data for one frame):

```
xensiv_bgt60trxx_get_fifo_data(&sensor.dev, data, NUM_SAMPLES_PER_FRAME);
```

Range (distance)

In reality, in response to one transmitted signal we receive not one, but several reflected signals, since there are other objects in the space around the target object. We call these "additional" reflected signals echoes.

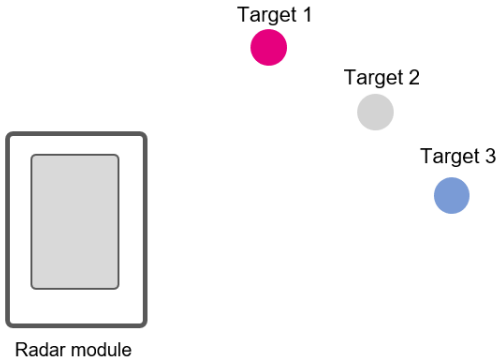


Figure 8. Radar and target objects

(1) The returning echo is a delayed version of the transmitted signal. The figure shows a graph of the frequency versus time.

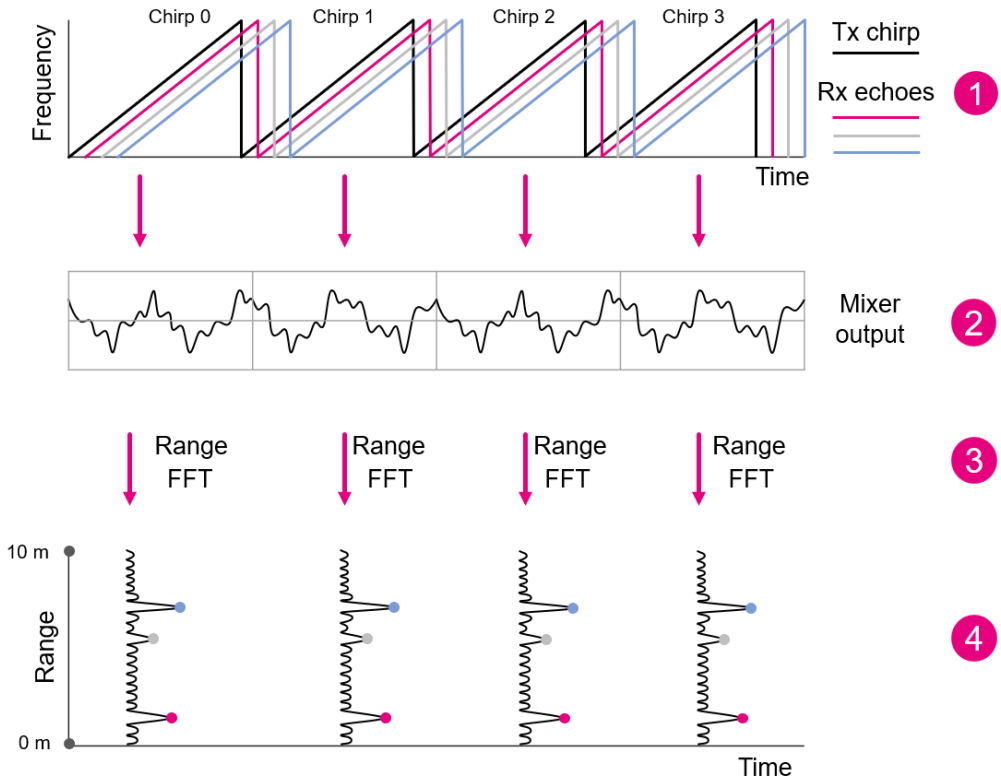


Figure 9. Range calculation

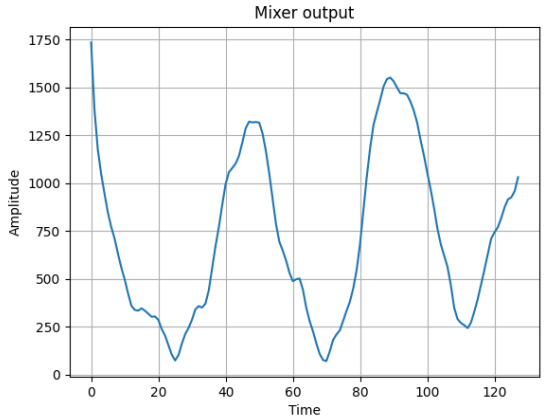
A time delay of the echo signal introduces a difference in frequency thanks to the mixer. The range can be found by measuring the frequency of the signal generated by the mixer.

(2) Three returning signals are coming to the mixer output of the radar.

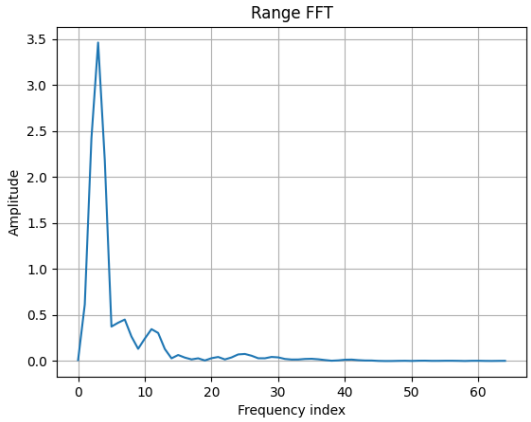
(3) FFT checks the participation of every returning signal in the construction of the examined signal. It finds the contribution from sinusoids of different frequencies above a certain threshold.

(4) Here are shown the amplitudes of the Rx signals for a given range. The peaks indicate that a signal has been reflected at the given range. FFT reveals the ranges of mixer outputs through detection of peak frequencies. This gives us the beat frequencies of all three echoes and hence the target ranges.

The screenshots here and further were generated using a Python script which can be found at GitHub.

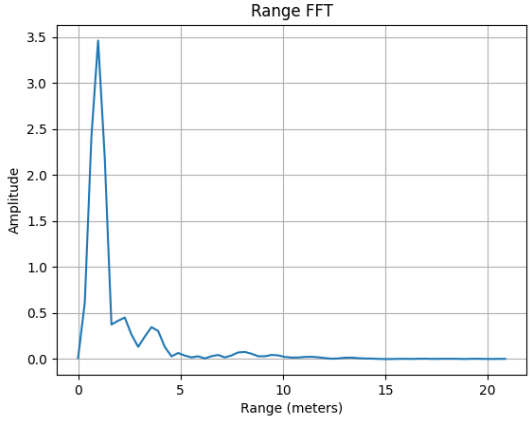


A mixer output when placing an object (a wall) in front of the sensor at ~1 meter.



When computing the range FFT of the mixer output, and displaying the magnitude of it, we can see a “peak” at the frequency index 3.

The frequency index can be converted into meters (see screenshot below).



The “frequency indexes” have been converted into meters. We see a big echo at around 1 meter.

Velocity

If the target object is moving with velocity v, after a time Tc or chirp repetition time, equal to the interval between two chirps, the object will be moved by a distance Δd.

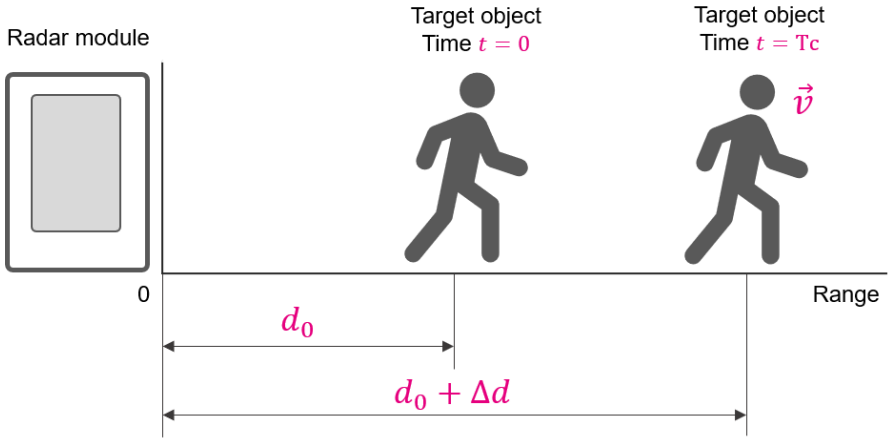


Figure 10. Parameters of moving target object

That also causes a shift in the phase of the reflected signals. The amplitude in this case remains practically unchanged. Figure 11 shows the reflected 1st and 2nd chirps (mixer output) to highlight the phase shift. In the reality they don´t start at the same moment of time.

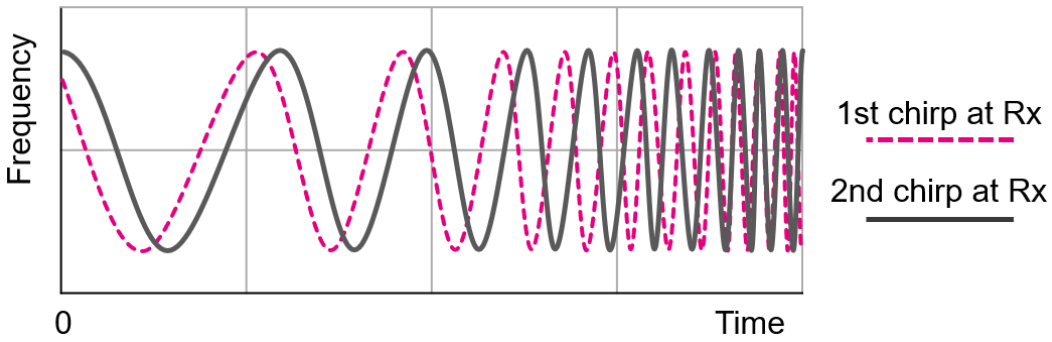


Figure 11. Phase shift of the reflected signals

The velocity of a single target can be calculated by a phase evolution over a period of time.

Returning signal (1) (see Figure 12) have a phase shift caused by a movement of the target objects. Longer frame length gives a better resolution because it´s easier to distinguish the velocities.

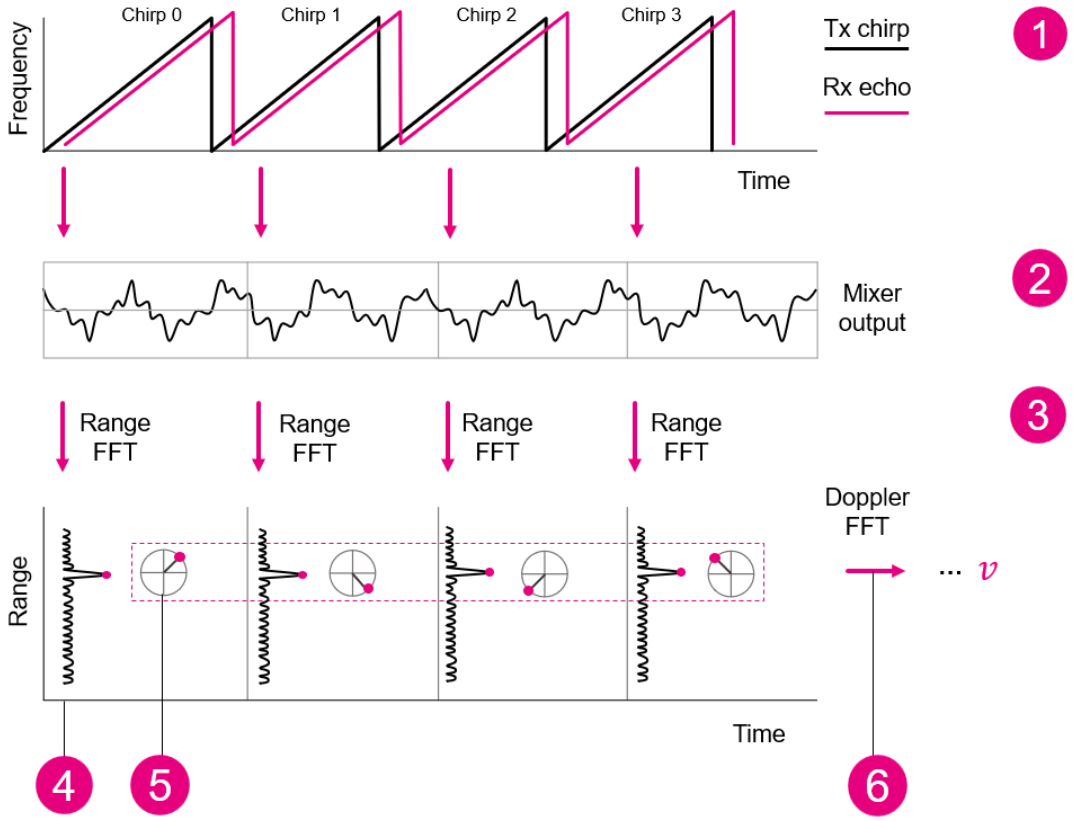


Figure 12. Velocity calculation

Steps (2), (3) and (4) are equal to the range computation.

(4) Displays the amplitude of the complex signal (result of the FFT computation) and (5) displays the phase of the complex signal. If we consider the phase for a given range over the time (where we observe a peak), we can see that it rotates with a certain speed. This speed can be used to estimate the velocity of the target.

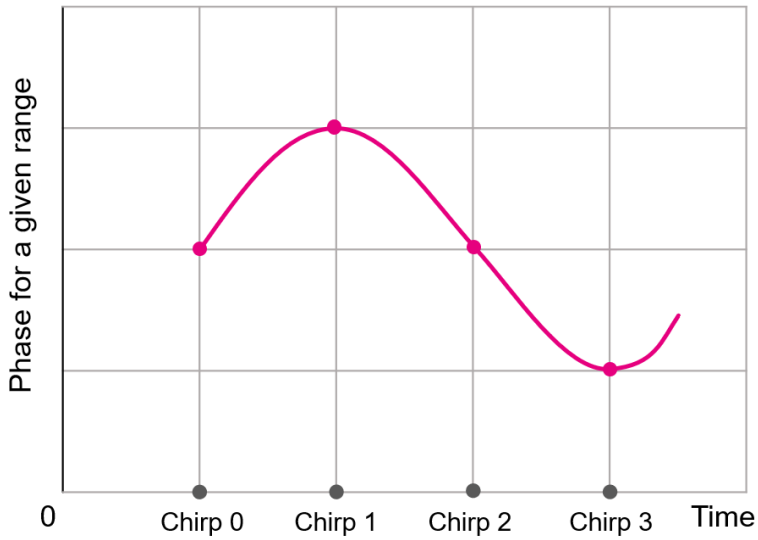
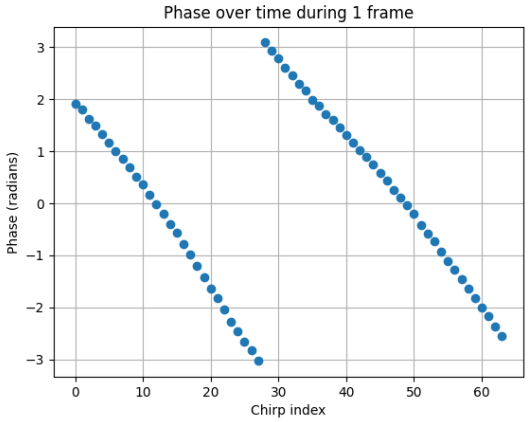


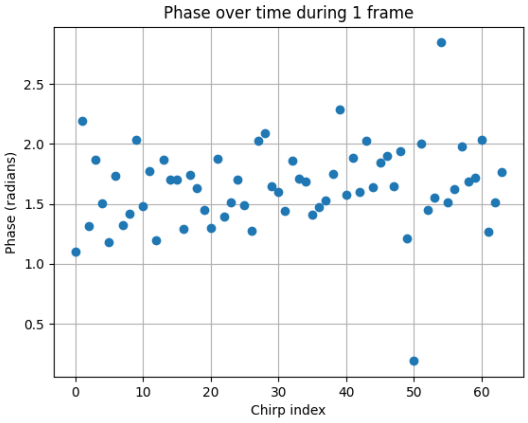
Figure 13. Phase evolution over time

Doppler FFT (6) computed on the phase signal (5) helps us to estimate the frequency of the phase rotation which leads to the velocity of the target.



The screenshot displays the phase over time (during 1 frame) when detecting a target with a given velocity – low velocity in that case.

The jump at index 29 occurs because the phase always remains between -Pi and Pi.



In case there is no motion, a DC value will be measured.

Angle of arrival (azimuth)

The angle of the target object relative to the radar is calculated by using multiple receiving antennas (two in our case).

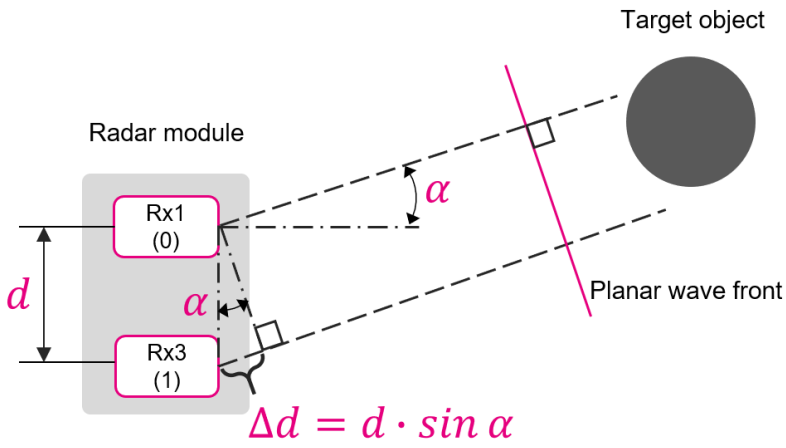


Figure 14. Geometry of angle of arrival

d is the distance between the antennas. Let's choose one of the antennas – Rx1 - as a reference point (0). The angle α is an angle of "incidence" of the planar wave front coming from a target object. The angle is determined relative to the imaginary horizontal axis of the antennas. To reach the second antenna Rx3, the wave will have to travel an additional distance Δd equal to $d \cdot \sin\alpha$. Additional distance supposes phase difference. Using it we compute the angle α.

In our case, there are two antennas, therefore the accuracy might be low because of the noise. When using more antennas (> 4), we could apply an angle FFT to extract the phase evolution over the time and compute a more accurate angle of arrival.

Digital beam forming is used for angle estimation. This method allows to focus a radar transmitter or receiver in a certain direction in 3D space. A left to right “scanning” direction is commonly referred to as azimuth.

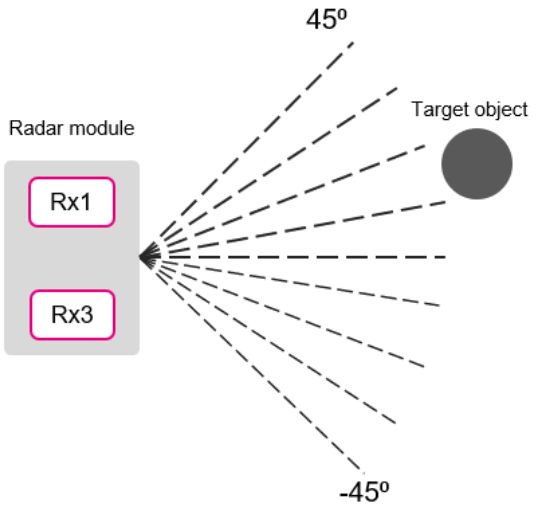
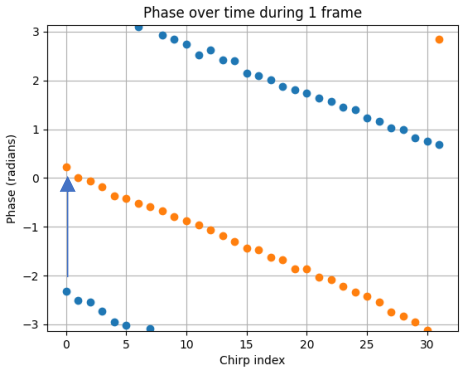


Figure 15. Digital beamforming in radar

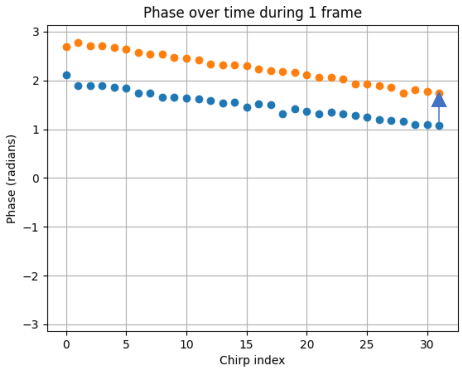
The screenshots below illustrate the following: a person comes from the side of the radar module and stops in front of it.



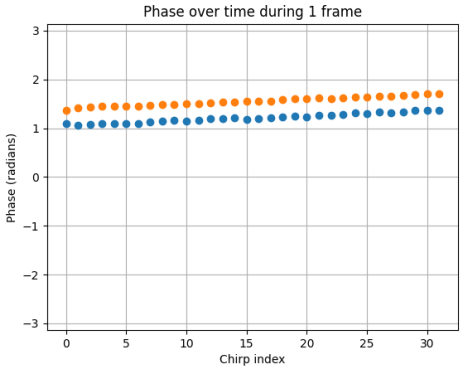
Blue: phase over time for antenna Rx1.

Orange: phase over time for antenna Rx3.

The phase difference is around 2 rad (i.e. the target is not in front of the radar module).



The phase difference is now smaller, around 1 rad (i.e. the target is more in front of the radar module).



The phase difference is now even smaller, the target is almost in front of the sensor.

Workflow overview

This workflow was followed to develop this application.

1. Use Radar Fusion GUI from Infineon to configure the radar sensor.

The configuration of the radar sensor enables to select the maximum detection range of the radar, the range resolution and the speed resolution. See this app note to learn more.

Once the configuration is completed, the Radar Fusion GUI enables to generate a file (radar_settings.h) that stores the configuration of the radar, and that can be used to initialize the measurement in C language.

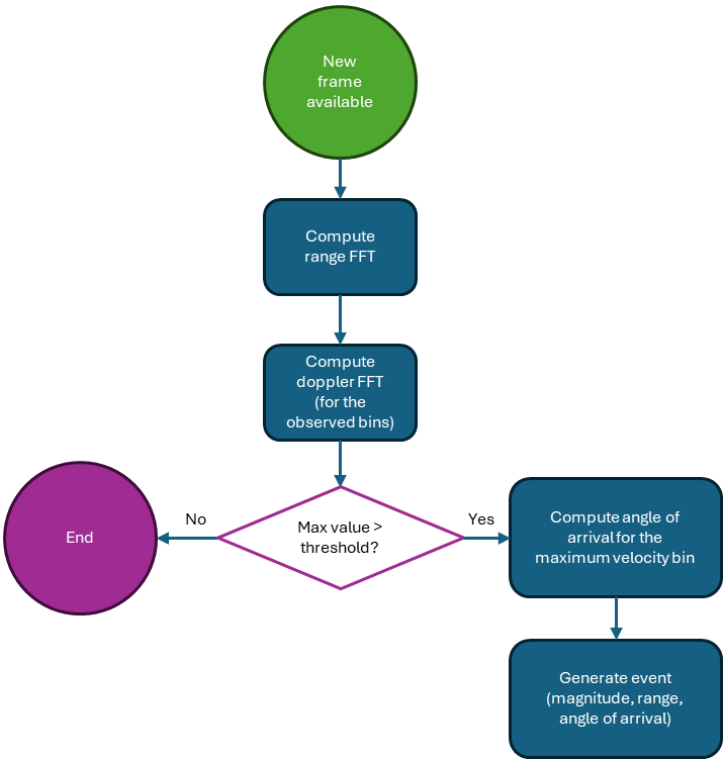
2. Gather data using Radar Fusion GUI.

Using the Radar Fusion GUI, you can store the data measured by the sensor (using the configuration you selected on the first step) into an “NPY” format that can be read easily using Python.

It’s suggested to play different scenarios with the radar sensor and the motion you want to detect (someone walking in front the sensor, someone running, …) and to store and label those data (i.e. for each data set – “NPY file” – document what happened during the scenario). You can also vary the static targets to see their effects.

3. Perform signal processing using Python to determine the right algorithm.

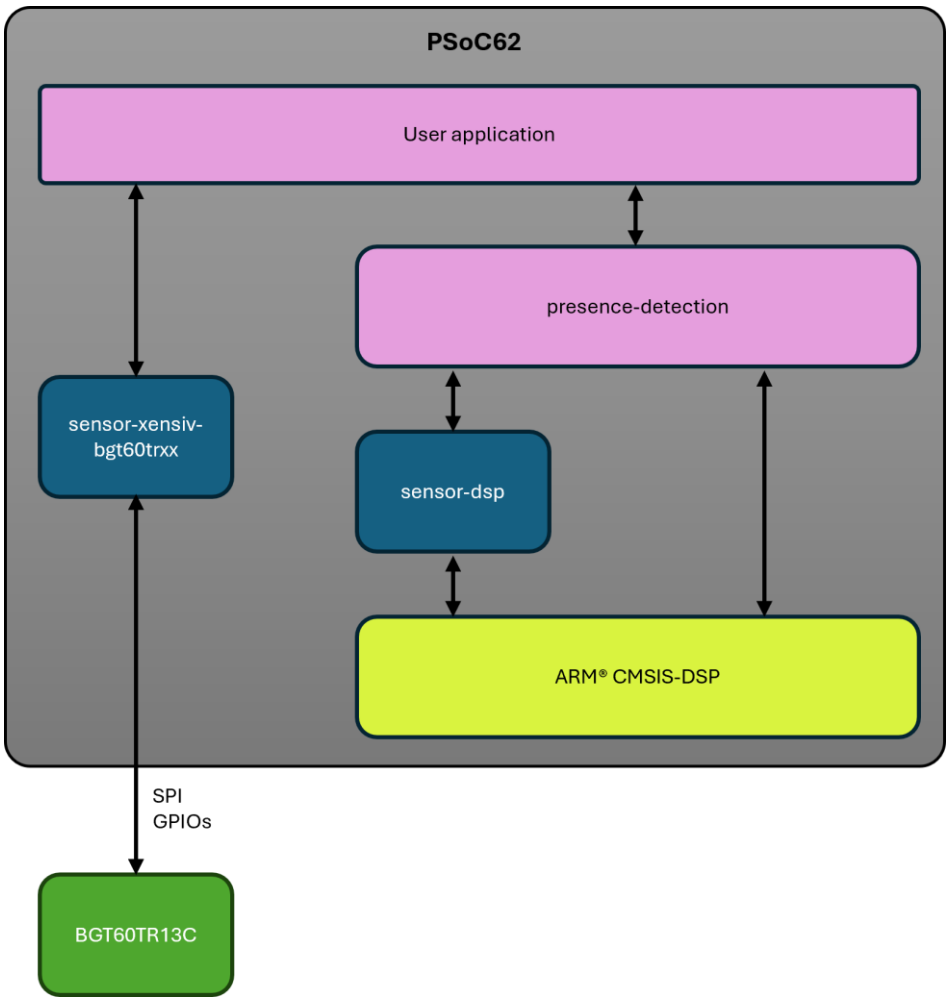
You can use Python and the “NPY” files that you generated in the previous step to develop your processing algorithm. Once the algorithm has been defined, you can port it into C to be run on the PSoC62. The algorithm that is implemented looks like below:



Computing the doppler FFT enables to compute the velocity of the reflected signals. By ignoring echoes with a velocity of 0m/s we can detect motion only (and ignore the static targets).

Another technique allowing to ignore the static targets is to compute how the background looks like using digital filter (also known as clutter) and then subtract the background from the measured signal. We chose to use the doppler FFT as the results in the field were better.

4. Implement the algorithm in C# (computation on PC side). It can be useful to make live tests and see the memory consumption.
   An example can be found here in GUI folder.
5. Implement the algorithm in C to be used with the PSoC62 on the RDK2 (no computer needed anymore). A firmware structure implemented for this application is shown below.



User application: enables to configure and start the radar measurement. Feeds the “presence-detection” lib with the gathered sensor data and listen to events generated by the “presence-detection” lib.

presence-detection lib: uses the raw radar sensor data to detect a motion (or not). When a motion is detected, generate an event.

sensor-dsp lib: contains some useful mathematic functions.

ARM (r) CMSIS-DSP lib: contains useful FFT mathematic functions.

sensor-xensiv-bgt60trxx lib: enables to configure and communicate with the BGT60TR13C.

Useful code examples

All source codes used for the application notes are available on GitHub:

RDK3 project using Infineon's library to detect a motion (to be used with Rutronik System Solutions iOS/Android app)	RDK3_Radar_Presence_Detection
Project to be used with RDK2 enabling to use the Radar Fusion GUI of Infineon (useful for configuration and to record the radar data)	RDK2_RAB3_Radar_Fusion
Project to be used with RDK2 enabling to implement data processing in C# (firmware and GUI)	RDK2_RAB3_Radar_USB_Gateway
Project to be used with RDK2 where the data processing is performed directly on the RDK2 (firmware and GUI). This project is a good start, if you want to implement the signal processing yourself on a microcontroller.	RDK2_RAB3_Radar_Presence_Detection

Deep dive in the C source code

Initialize the radar sensor and start the frame generation

The initialization (and all the calls to the sensor-xensinv-bgt60trxx library) are performed inside the file "bgt60tr13c.c" file:

`xensiv_bgt60trxx_mtb_init(&sensor,`
`&spi_obj,`
`ARDU_CS,`
`ARDU_IO4,`
`register_list,`
`XENSIV_BGT60TRXX_CONF_NUM_REGS);`
	This call enables to initialize the radar (over SPI) using the configuration defined inside the file "radar_settings.h" (previously generated using the Radar Fusion GUI software).

`xensiv_bgt60trxx_mtb_interrupt_init(&sensor,`
`NUM_SAMPLES_PER_FRAME,`
`ARDU_IO6,`
	This call enables to configure the interrupt service routine to be called,

CYHAL_ISR_PRIORITY_DEFAULT, xensiv_bgt60trxx_mtb_interrupt_handler, NULL);	when a data frame is available inside the radar FIFO memory.
xensiv_bgt60trxx_start_frame(&sensor.dev, true)	This call enables to start the frame generation. You need to read the data coming from the radar sensor fast enough, or an overflow will occur and the measurement will be stopped.

Once the measurement is started, the function "xensiv_bgt60trxx_mtb_interrupt_handler" will be called when data are available.

To read the data, you will have to call following function:

```
xensiv_bgt60trxx_get_fifo_data(&sensor.dev, data, NUM_SAMPLES_PER_FRAME);
```

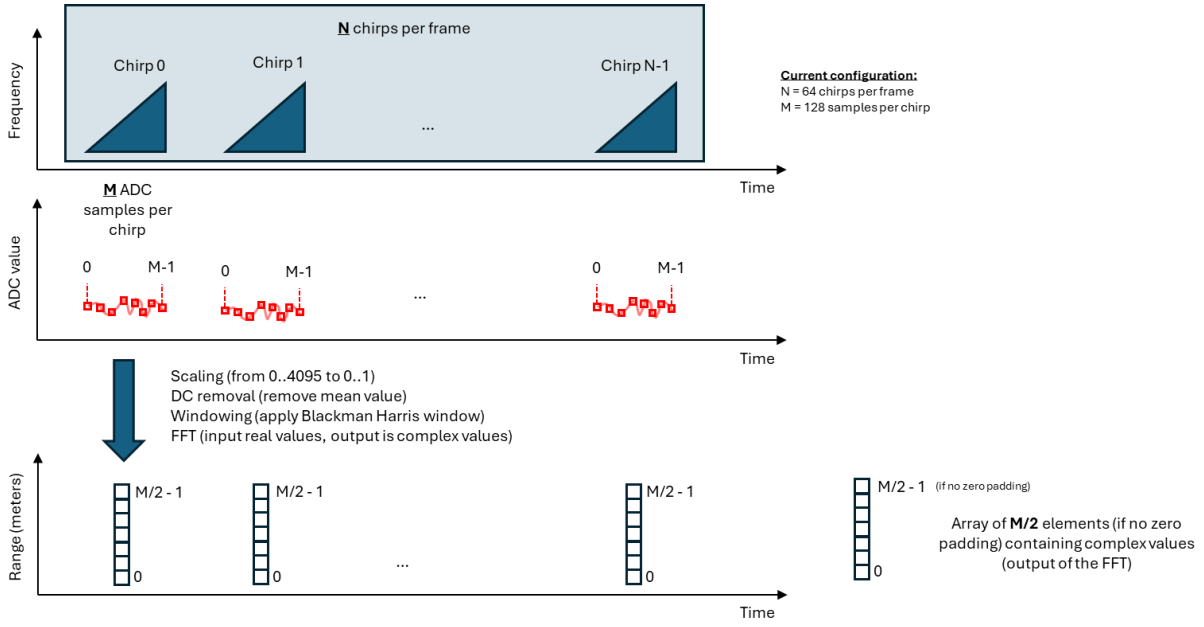
The buffer "data" will contains the raw ADC data (between 0 and 4095) coming from the radar module.

/!\ In the current configuration, we are using 2 receiving antennas (since we want to compute the angle of arrival / azimuth). The format of the "data" buffer looks like:

data[0]	Antenna 0, chirp 0, ADC sample 0
data[1]	Antenna 1, chirp 0, ADC sample 0
data[2]	Antenna 0, chirp 0, ADC sample 1
data[3]	Antenna 1, chirp 0, ADC sample 1
…	…

Compute the range FFT

At the moment, the output of the range FFT is only used as input for the doppler FFT. In a future version, we might use the output of the range FFT to track the static targets (using digital filter).



To compute the range FFT, following operations are performed:

Scaling: the ADC values (between 0 and 4095) are converted into floating point values between 0 and 1.

DC removal: For each chirp, the mean value is computed and removed from all samples

Windowing: a Blackman Harris window is applied on each chirp (enable to reduce the effect of spectral leakage)

FFT: after all the transformation on the time domain data, the FFT is computed (from real values into complex values), therefore the output of the range FFT is an array containing complex values (real and imaginary values).

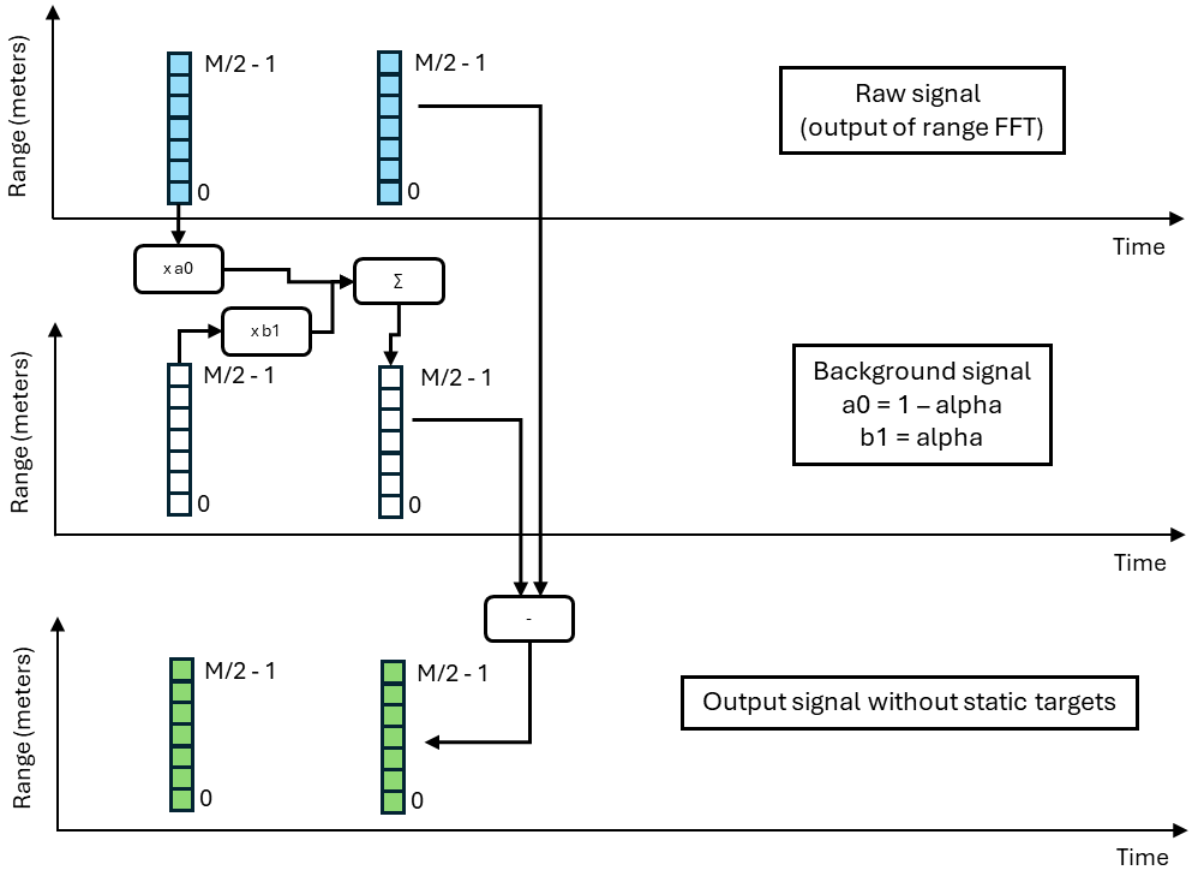
The range FFT is computed inside the file “presence_detection.c”.

`range_fft_do(frame_samples,`
`range,`
`adc_samples,`
`true, // remove mean`
`window, // Blackman Harris`
`REQUIRED_ANTENNA_COUNT,`
`internal_params.samples_per_chirp,`
`internal_params.chirps_per_frame);`
	The output of the function is the buffer “range” which as size:
	Antenna count * chirps per frame * FFT length
	
	With FFT length = samples per chirps / 2 (since we are not using zero padding at the moment)

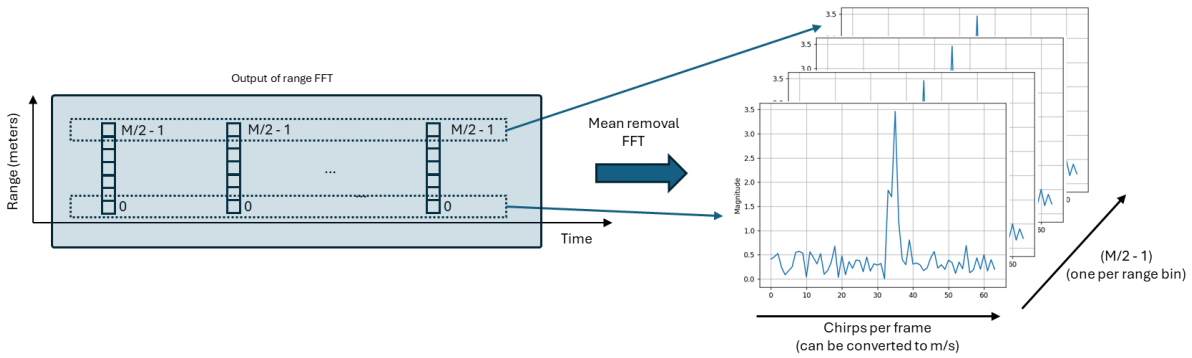
The format of the “range” buffer looks like:

range[0]	Antenna 0, Chirp 0, Range index 0
range[1]	Antenna 0, Chirp 0, Range index 1
…	…
range[samples per chirps / 2]	Antenna 0, Chirp 1, Range index 0
…	…
range[chirps per frame * samples per chirps / 2]	Antenna 1, Chirp 0, Range index 0

To keep track of the static targets (i.e. echoes coming from targets which are not moving, like walls for example) we might use a low pass filter on the output of the range FFT (for each chirp). Example:



Compute the doppler FFT

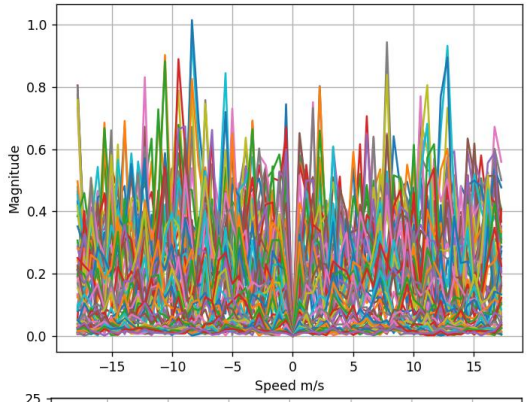


To compute the range FFT, we took the ADC samples of one chirp (fast time) and computed the FFT out of it.

To compute the doppler FFT, we will take the results of the range FFT, and compute the FFT over the chirps (slow time). Therefore, we need to store in memory the complete output of the range FFT.

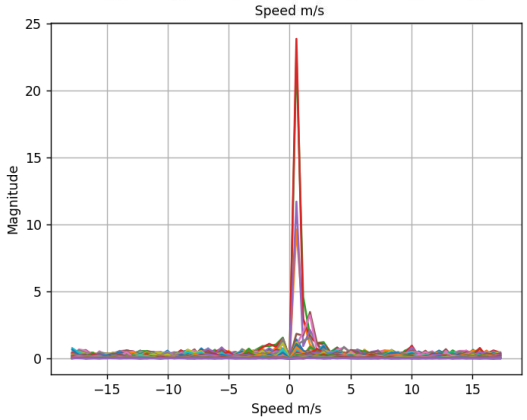
If we remove the mean value before computing the FFT, we will remove all static targets (i.e. echoes with 0/ms velocity).

That will give us such types of results:



Each curve represents the doppler FFT of one bin (we have M/2 – 1 bins).

Here we see only noise (no motion detected).



Each curve represents the doppler FFT of one bin (we have M/2 – 1 bins).

Here we see a big spike for the red curve (which represents the bin/range 3 meters) ➔ we have a motion at range 3 meters!

By setting a threshold on the magnitude of the doppler FFT results, we can identify easily if a motion happened or not.

There are several ways to define the threshold: you can use a fixed one, or use a dynamic one by using techniques likes CFAR or example.

The doppler FFT is computed inside the file “presence_detection.c”.

`doppler_fft_bin_do(range,`
`  doppler_out, // Doppler FFT output (size is chirps_per_frame)`
`  true, // Remove mean`
`  NULL, // No window`
`  bin_idx, // Bin index`
`  0, // Antenna index`
`  internal_params.chirps_per_frame,`
`  fft_len);`
	The output of the function is the buffer “doppler_out” which as size: chirps per frame
	
	It contains complex values.
	
	Remark: in that case the doppler FFT is only computed for the bin index “bin_idx”.
	
	To compute it for all ranges (distances), you need to do a loop:
	`for(uint16_t bin_idx = 0 bin_idx < fft_len; ++bin_idx){...}`
	
	With:
	`uint16_t fft_len = internal_params.samples_per_chirp / 2;`

In the current version, we only the select the biggest magnitude, so the output of the algorithm is a magnitude and the index of the bin (or range) for which we have the biggest magnitude.

Angle computation

In the previous step, we computed the doppler FFT to detect if a motion (echo with a velocity != 0m/s) occurred or not. This leaves us with the index of the bin (range) for which the biggest motion has been detected.

The next step is then to determine the angle of arrival (or azimuth) of this motion.

To do so, we will use the output of the range FFT (for the 2 antennas) and compute the phase difference for the given bin (range).

To get a better result, we use the average over all the chirps (below is the simplified source code):

```
for(uint16_t chirp_idx = 0; chirp_idx < internal_params.chirps_per_frame; ++chirp_idx)
{
 …
 float phase0 = atan2f(imag0, real0);
 float phase1 = atan2f(imag1, real1);
 …
 angle_sum += angle_diff;
}
return (angle_sum / (float) internal_params.chirps_per_frame);
```

Remark: the angle of arrival is computed using the raw range FFT. An improvement might be to first remove the static targets, and then compute the phase difference again.