3D People Counting Demo Software Implementation Guide

Rev 1.0

February 2021

Texas Instruments, Incorporated
12500 TI Boulevard
Dallas, Texas 75243 USA

Table of Contents

References:

1. MMWAVE SDK User Guide, Product Release 3.5.x.x, https://www.ti.com/tool/MMWAVE-SDK
2. Aravindh Krishnamoorthy, Deepak Menon, “Matrix Inversion Using Cholesky Decomposition”, https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6710599.
3. Optimum Array Processing: Part IV of Detection, Estimation, and Modulation Theory, Harry L. Van Trees, 2002 John Wiley & Sons, Inc

1 Purpose and Scope

The purpose of this document is to provide detailed description of 3D People Counting demo implementation using mmWave SDK software development components. The document provides description of system execution flow, memory usage, task organization and execution and benchmark results. The document also provides the implementation details of the low level signal processing chain.

2 High Level Demo Setup

The 3D People Counting demo is implemented on IWR6843. It consists of two major subsystems Detection Layer and Tracker Layer. The detection layer is implemented on a Hardware Accelerator (HWA) driven by R4F, and DSP (C674x). The Tracker Layer is implemented on R4F. The detection layer produces a point cloud with each point containing spherical coordinates, radial velocity, and signal to noise ratio (SNR). The Tracker Layer finds and tracks clusters in the point cloud. The point cloud and the tracking data are sent out via UART port to the GUI running on the Host PC.

The 3D People Counting demo setup is shown in Figure 1.

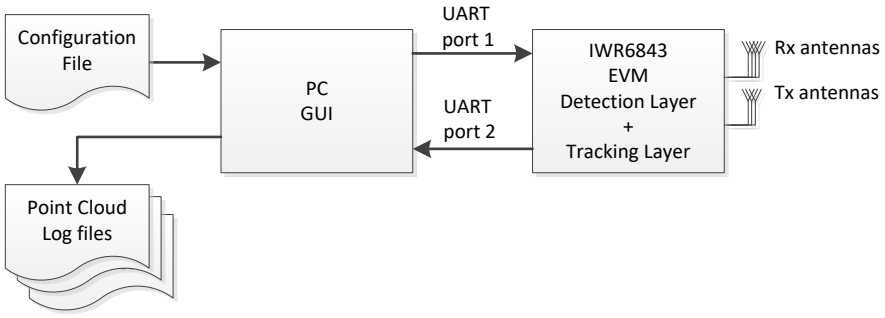


Figure 1 – 3D People Counting demo setup

Sensor EVM board is connected to PC via USB. PC GUI reads the configuration file and sends configuration commands to radar via UART port 1. After the initialization sequence the radar starts sending detection data to PC GUI via UART port 2. Optionally point cloud and tracking data is saved to log files.

3 Signal processing chain

The implementation of the 3D People Counting demo on the IWR6843 consists of a low level signal chain running on the C674x DSP, and the tracking module running on R4F processor.

The demo does two basic functions:

1 Low level signal processing: use the radar data to produce a point cloud with each point containing spherical coordinates, radial velocity, and SNR
2 Group tracking: finds and tracks clusters in the point cloud.

There are two separate signal processing chains for the 3D people counting demo, one optimized for wall-mount, shown in Figure 2, and the other for ceil-mount applications, shown in Figure 3. For the wall mount application the IWR6843 ISK/ODS/AOP EVM boards are used, and for the ceil-mount the IWR6843 ODS/AOP EVM board is used. The processing chain is selected through the configuration. Also the build images for these two applications differ since the code and data placement is optimized for seach application. In this document the detection algorithm for the wall-mount processing chain is called method 1 and for the ceil-mount method 2.

The low level signal processing includes:

- Range processing
    - For each antenna, 1D windowing, and 1D FFT,
- Static clutter removal,
- Capon Beamforming (BF):
    - Covariance matrix generation, angle spectrum generation:
    - Range-azimuth heatmap (method 1)
    - Range-azimuth-elevation heatmap , coarse azimuth/elevation estimation (method 2),
- CFAR detection algorithm:
    - Two-pass, CFAR detection: first pass CFAR-CASO in the range domain, confirmed by second pass CFAR-CASO in the angle domain, to find detection points,
- Elevation Estimation (method 1)
    - Capon BF algorithm is applied again for each point detected in Range-Azimuth heatmap
    - 1-D Elevation heatmap is calculated for the elevation estimation,
- Fine Azimuth/Elevation Estimation (method 2)
    - 2D zoom-in is performed for the detected azimuth-elevation bin for fine angle estimation,
- Radial Velocity Estimation:
    - For each detected [range, azimuth] pair from the detection module, Doppler is estimated by filtering the range bin using Capon beam-weights, followed by a maximum peak search in the FFT of the filtered range bin.

Group tracking processing:

- Operates on point cloud,
- Searches for clusters in Cartesian and Doppler Space,
- Predicts movement of clusters to maintain a track of unique objects such as people

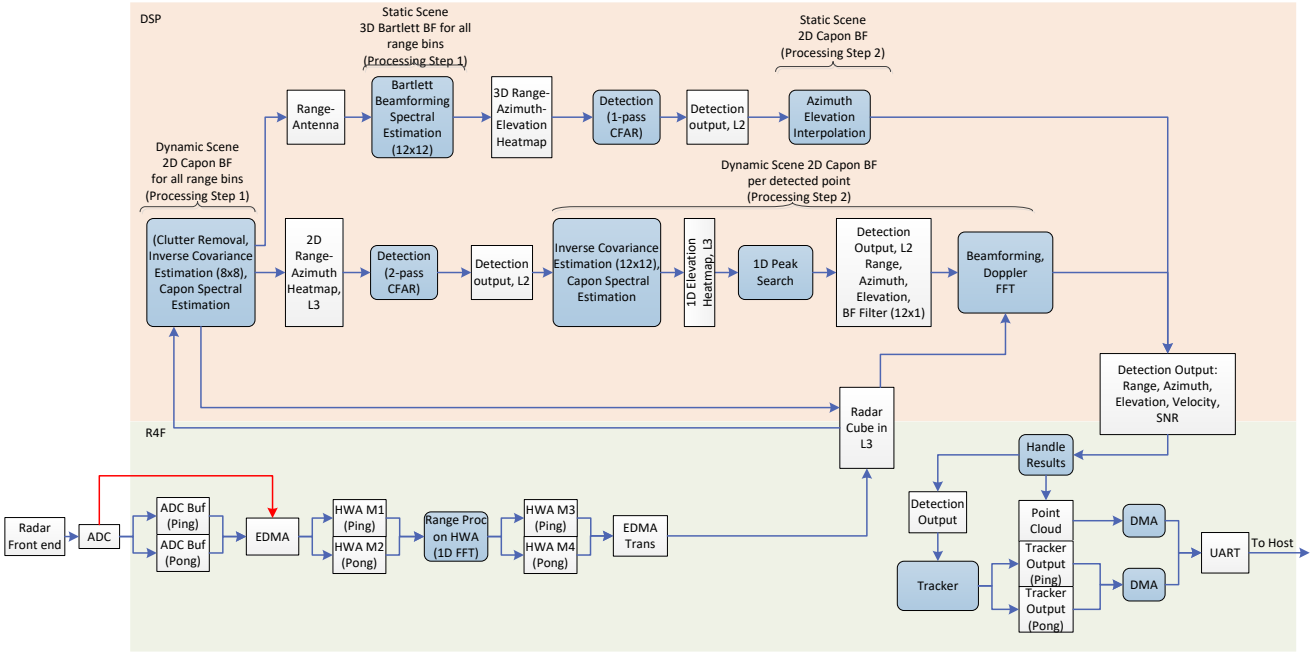


Figure 2 - The signal processing chain of the 3D wall-mount demo.

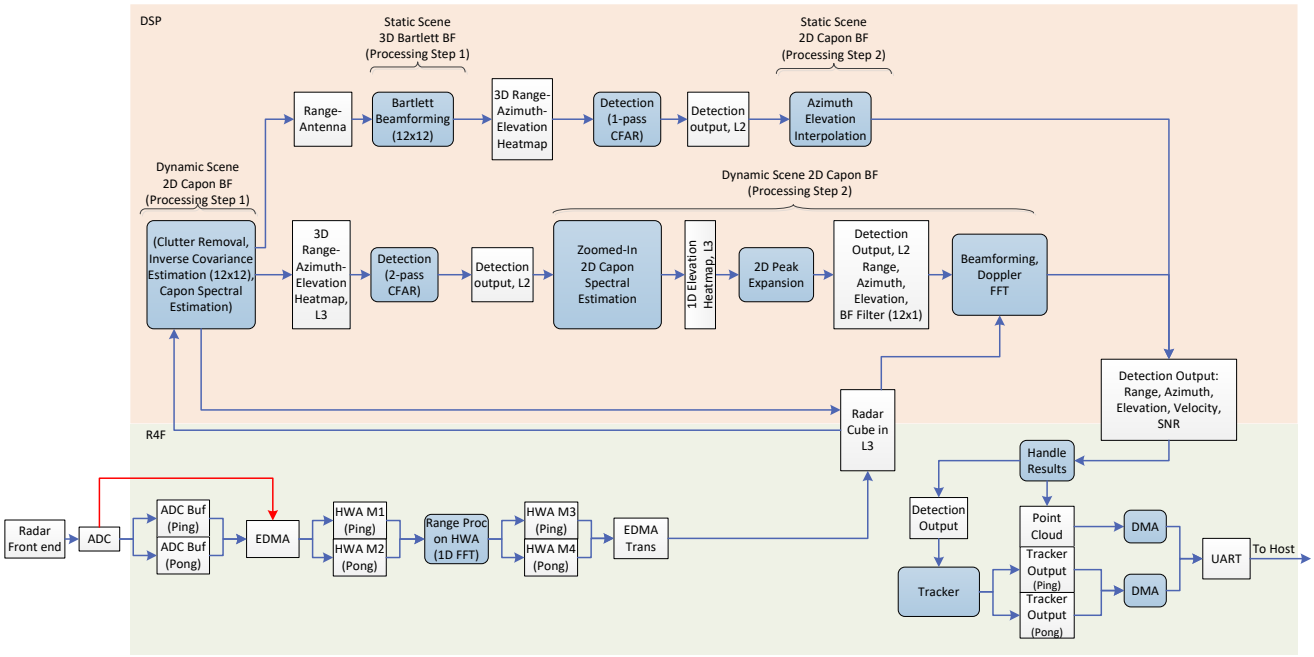


Figure 3 - The signal processing chain of the 3D ceil-mount demo.

4 3D People Counting Implementation using SDK Components

The 3D People Counting demo is implemented using mmWave SDK components and other custom components developed following the SDK architecture. The SDK provides a basic structure for developing radar processing software. This structure includes mmWaveAPI, a Data-Path Manager (DPM) which handles execution of the Signal Processing – Data Path Chain (DPC). The DPC is made of Data Path Units (DPUs). For detailed explanation of these components see [1].

The 3D People Counting demo partition between R4F (MSS) and DSP (DSS) is shown in Figure 4. The components in green are executed on MSS while components in pink are executed on DSS. The demo application is split between MSS and DSS.

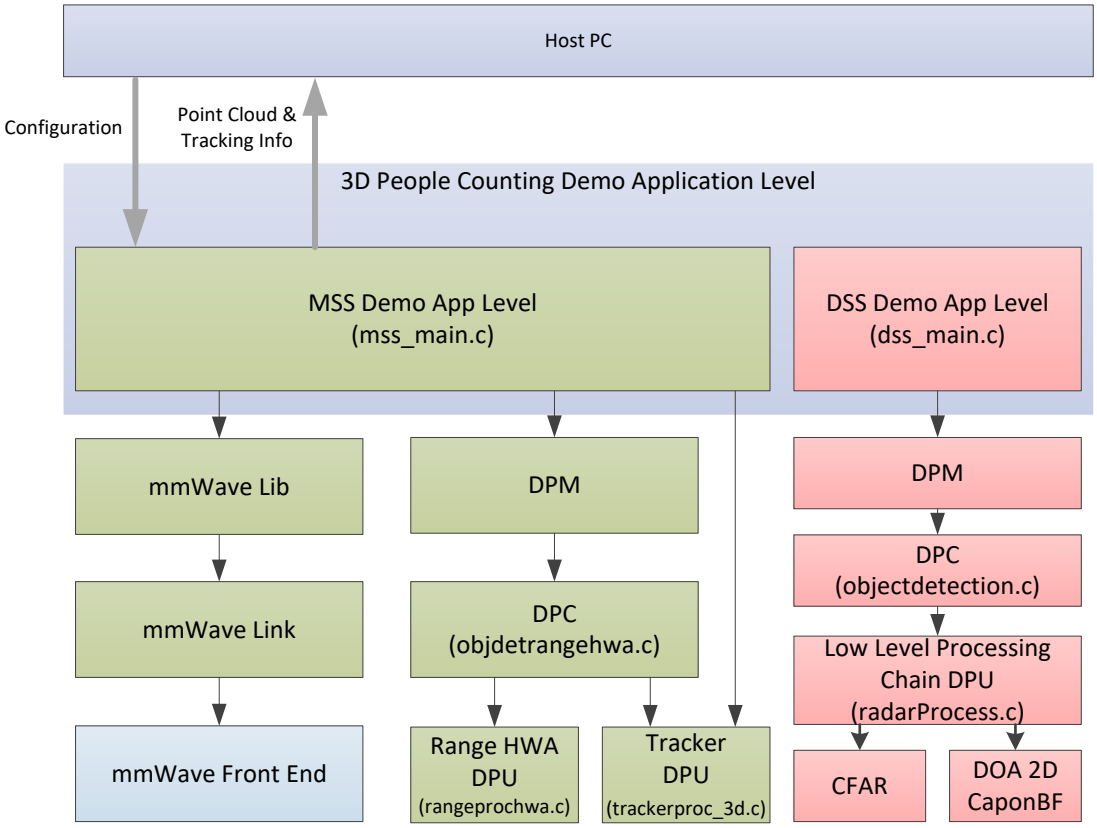


Figure 4 – 3D People Counting Demo implementation using mmWave SDK components

4.1 3D People Counting Application level

At the top level the demo application is split between MSS and DSS. The application layers on both domains DSS and MSS call the DPM APIs through which they control the configuration and execution of DPCs. The application layer on MSS also controls the radar front end, and communicates with the Host.

4.2 mmWave Lib

The mmWave Lib module is a higher layer control running on top of mmWaveLink and LLD API (drivers API). It provides simpler and fewer set of APIs for application to control the radar front end. In the demo the mmWave module runs only on R4F (MSS), in “Isolation” execution mode, and it is configured in “Full” configuration mode. The “Full” configuration mode implements the basic chirp/frame sequence of the radar front end.

4.3 mmWave Link

mmWaveLink is a control layer and primarily implements the protocol that is used to communicate between the Radar Subsystem (RADARSS) and the controlling entity which can be either MSS and/or DSS. It provides a suite of low level APIs that the application (or the software layer on top of it) can call to enable/configure/control the RADARSS. In 3D People Counting demo the mmWave Link layer is only accessed by mmWave Lib layer running on MSS.

4.4 Data-Path Manager (DPM)

DPM is the foundation layer that enables the "scalability" aspect of the architecture. It encapsulates the overall software execution on a core. This layer absorbs all the messaging complexities (cross core and intra core) and provides standard APIs for integration at the application level and also for integrating any "data processing chain". In 3D People Counting demo DPM runs in “Distributed domain” configuration mode, where the data path control is on MSS, while data path execution is split between MSS/HWA and DSS. The DPM APIs exposed to application layer and DPC are shown in Table 1.

Function name	Description
DPM_execute	The function executes the DPM Module. This involves the following: a) Handling of the reception of the IPC Messages exchanged between the DPM Peers. b) Execution & processing of the input data which has either been injected or received via the chirp available.
DPM_ioctl	The function is used to configure the processing chain.
DPM_start	The function is used to start the processing chain.
DPM_sendResult	The function is used to send the processing chain results to the remote DPM entities. The flag "isAckNeeded" can be set and this would cause the DPM framework to send to a report once the peer domain has been notified about the result availability. This can be used to ensure that the result buffer is not being reused.
DPM_relayResult	The function is used to relay the partial processing chain results from one domain to another. It does not require an acknowledgment back for the relayed results.
DPM_stop	The function is used to stop the processing chain.

DPM_notifyExecute	The function notifies the DPM module that the processing chain is ready to be executed. This function is invoked by the DPC which in turn will allow the DPM framework to invoke the profile registered execute method.
DPM_synch	The function is used to synchronize the execution of the framework between the DPM domains.
DPM_init	The function is used to initialize the data path manager and initialize and load the processing chain.
DPM_deinit	The function is used to deinitialize and shutdown the processing chain.

Table 1 – DPM APIs

4.5 Data processing chain (DPC)

DPC is a separate layer within the data-path that encapsulates all the data processing needs of an mmWave application and provides a well-defined interface for integration with the application. Internally this layer uses the functionality exposed by Data processing units (DPUs) and DPM to realize the data flow needed for the data processing chain. As the data path processing in 3D People Counting demo is split between MSS and DSS, the DPC layer exists on both DSS and MSS domains.

4.5.1 DPC running on MSS

The DPC functions registered and called by DPM running on MSS are shown in Table 2. The functions are located in objdetrangehwa.c.

Function Name	Description
DPC_ObjectDetection_init	DPC Initialization function. It calls its DPU initialization functions DPU_RangeProcHWA_init() and DPU_TrackerProc_init().
DPC_ObjectDetection_start	Executed upon sensor start request. It sends control message *DPU_RangeProcHWA_Cmd_triggerProc* to the Range DPU to configure and enable HWA to be ready for chirp events.
DPC_ObjectDetection_execute	This function calls data processing chain DPU. It only controls DPU_RangeProcHWA_process().
DPC_ObjectDetection_ioctl	IO control function used to configure and control DPC. The commands are: - Frame start, - Pre start common configuration, - Pre start configuration – invokes DPC_ObjDetRangeHwa_preStartConfig () which further calls DPU_RangeProcHWA_config ()

DPC_ObjectDetection_stop	Executed upon sensor stop request
DPC_ObjectDetection_deinit	De-initialization function
DPC_ObjectDetection_dataInjection	Invoked on reception of the data injection from DPM.
NULL	Chirp Available Function – not used
DPC_ObjectDetection_frameStart	This function is invoked upon reception of frame start event generated from RF front end. The function calls DPM DPM_notifyExecute() API, that will cause DPM framework to invoke the registered DPC execute method on MSS, DPC_ObjectDetection_execute().

Table 2 – DPC Functions running on MSS

4.5.2 DPC running on DSS

The DPC functions registered and called by DPM running on DSS are shown in Table 3. The source code of these functions is located in objectdetection.c. Note that although the functions have the same name as those running on MSS they have different contents and are part of two different builds.

Function Name	Description
DPC_ObjectDetection_init	DPC Initialization function. It is invoked by the application through DPM_init API. (See more details on this function are in Section 1).
DPC_ObjectDetection_start	Executed upon sensor start request. It only resets the frame start event counter.
DPC_ObjectDetection_execute	The low level signal processing chain is executed within this function. (See more details on this function are in Section 4.5.2.2)
DPC_ObjectDetection_ioctl	IO control function is used to configure and control DPC. The commands are: - Frame start, - Result exported, - Pre start common configuration, - Pre start configuration – invokes DPC_ObjDetDSP_preStartConfig() which further calls DPU_radarProcess_init()
DPC_ObjectDetection_stop	Executed upon sensor stop request
DPC_ObjectDetection_deinit	De-initialization function
DPC_ObjectDetection_dataInjection	Invoked on reception of the data injection from DPM.

NULL	Chirp Available Function – not used
DPC_ObjectDetection_frameStart	Executed upon frame start ISR from the RF front-end. It records the start time of the frame, and checks if the previous frame is completed.

Table 3 – DPC functions running on DSS

4.5.2.1 DPC_ObjectDetection_init()

The function allocates DPC instance from the system heap. It also allocates memory for the detection results, the structure DPC_ObjectDetection_ExecuteResult, and for the statistics information, the structure DPC_ObjectDetection_Stats. The structure DPC_ObjectDetection_ExecuteResult contains the point cloud list sized for maximum 750 points (defined by DOA_OUTPUT_MAXPOINTS in radarProcess.h). The DPC instance and it's main elements are shown in Figure 5. The Radar processing DPU instance allocation and initialization is called later, upon receiving the pre-start configuration function from MSS.

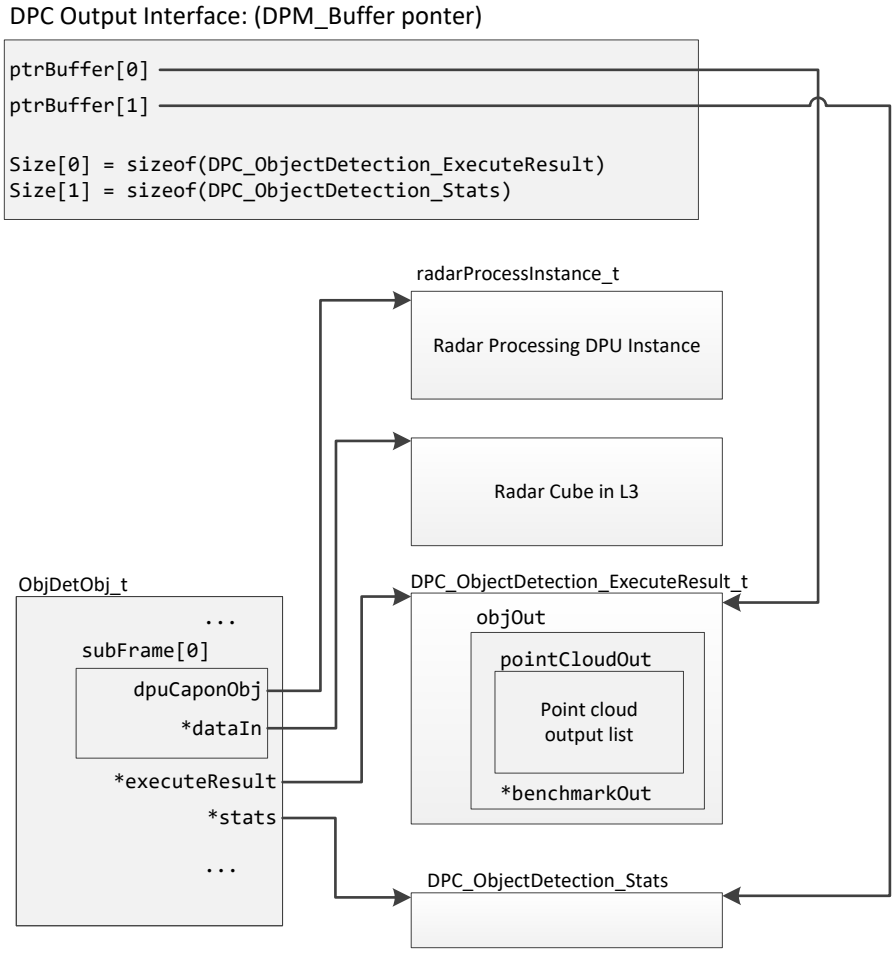


Figure 5 – Object Detection DPC instance on DSS and its output interface.

4.5.2.2 DPC_ObjectDetection_execute()

The function is executed per frame, after the range processing. The function calls the data processing chain DPU DPU_radarProcess_process(). It passes the pointers of the radar cube matrix, and the pointer to the point cloud output list pointCloudOut. After the DPU processing completion, the function updates the stats structure and sets the pointers of its output interface, DPM_buffer type pointer, and exits.

4.6 Data processing unit (DPU)

The signal processing chain is split into three mmWave DPU components:

1. Range processing DPU
2. Low Level Radar Processing DPU
3. Tracker DPU

4.6.1 Range Processing DPU

This processing unit performs 1D FFT processing on the chirp RF data during the active frame time and produces the output data in the L3 memory. This DPU is implemented on MSS and HWA. The actual processing, 1D windowing and 1D FFT, is performed by HWA, and it is interleaved with the active chirp time of the frame. During this time DPU task on MSS is in a pending state, allowing other tasks to run on MSS. Figure 10 provides more details on the task timing. The APIs are shown in Table 4.

Function Name	Description
DPU_RangeProcHWA_init	This is DPU initialization function. It allocates memory to store its internal data object and returns its handle.
DPU_RangeProcHWA_config	This is DPU configuration function. It saves buffer pointer and configurations including system resources and configures HWA and EDMA for runtime range processing.
DPU_RangeProcHWA_process	It executes FFT operation. It is invoked at a frame start time. The function is pending on a semaphore during the chirping period, and it exits after the last chirp processing of the frame is completed.
DPU_RangeProcHWA_control	It is DPU control function. The main command performed by this function is to configure and trigger HWA to be ready for the incoming chirp sequence.
DPU_RangeProcHWA_deinit	It frees the resources used for the DPU.

Table 4 – Range DPU APIs

The range DPU source code is located in rangeprochwa.c.

The major configuration parameters related to ADCBuff driver and Range DPU are shown in Table 5.

Configuration Parameters	Setting

HWA Input Mode	DPU_RangeProcHWA_InputMode_ISOLATED (ADC samples transferred from ADC buffer to internal HWA memory using EDMA)
Output Radar Cube format	DPIF_RADARCUBE_FORMAT_2, cmplx16ImRe_t, (X[numRangeBins][numDopplerChirps][numTxPatterns][numRxChan])
Interleave mode	Non-interleaved
ADC samples	16-bit ADC samples, Complex, Imaginary in LSB Real in MSB

Table 5 – ADC buffer and RANGE DPU related configuration parameters

The hardware resources related to HWA and EDMA configuration for the range DPU are stored in file pcount3D_hwres.h. This file is passed as a compiler command line define

--define=APP_RESOURCE_FILE="…\pcount3D_hwres.h"

4.6.2 Radar Processing DPU

The main part of the low level signal processing chain is implemented in this DPU which runs on DSS. Two different signal processing chains are implemented for wall mount and ceiling mount and are selected at a build time. The APIs are shown in Table 6. The DPU source code is located in radarProcess.c.

Function Name	Description
DPU_radarProcess_init	This is the initialization and the configuration function for the low level signal processing chain. It is called upon the pre start configuration command coming from MSS. It is called from DPC_ObjectDetection_ioctl() function within DPC_ObjDetDSP_preStartConfig() (see Section 5.1 for detailed information).
DPU_radarProcess_config	This function is empty.
DPU_radarProcess_process	This is the main body of the low level radar processing chain. It is called per frame. Section 5.2 provides detailed information on the algorithm implementation.
DPU_radarProcess_control	This function is empty.
DPU_radarProcess_deinit	It releases resources used for the DPU.

Table 6 – Low Level Radar Processing DPU APIs

4.6.3 Tracker DPU

This DPU runs on MSS. It is an unconventional DPU since it is accessed both from DPC and MSS application layer.

Funciton Name	Description
DPU_TrackerProc_init	It allocates memory to store its internal data object.

DPU_TrackerProc_config	The function is configuration function.
DPU_TrackerProc_process	The function is trackerProc DPU process function.
DPU_TrackerProc_control	This function is empty.
DPU_TrackerProc_deinit	It frees up the resources allocated during initialization.

Table 7 – Tracker DPU APIs.

4.7 System execution flow

4.7.1 Execution flow - Initialization

The system execution flow at the initialization time is shown in Figure 6. The code images on both DSS and MSS domains perform various driver initializations, synchronize to each other, create various tasks, and go to the pending state, waiting for the sequence of CLI commands to start processing.

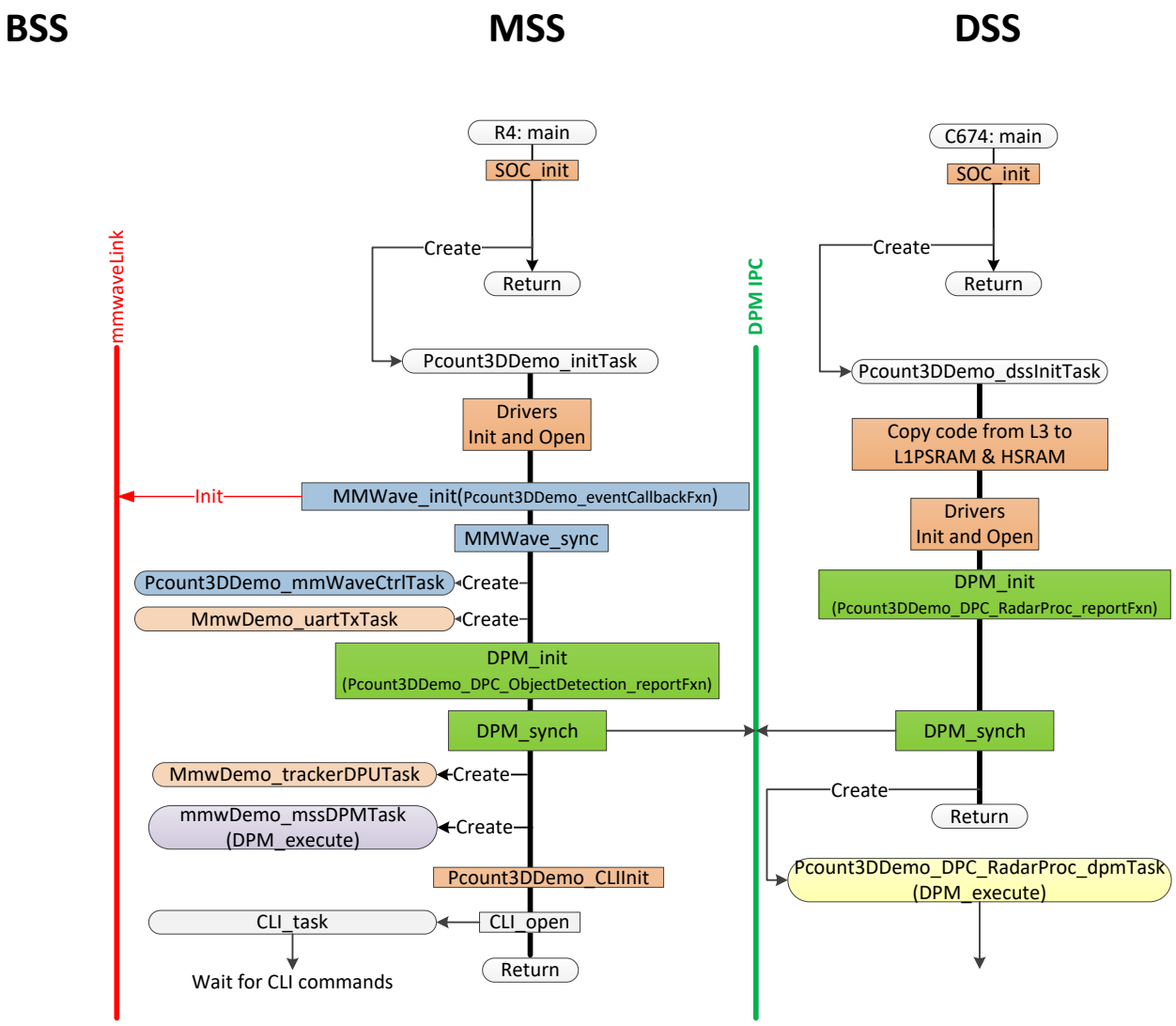


Figure 6 – System execution flow – Initialization sequence.

4.7.2 Execution flow – Configuration

The system execution flow during the configuration time is shown in Figure 7.

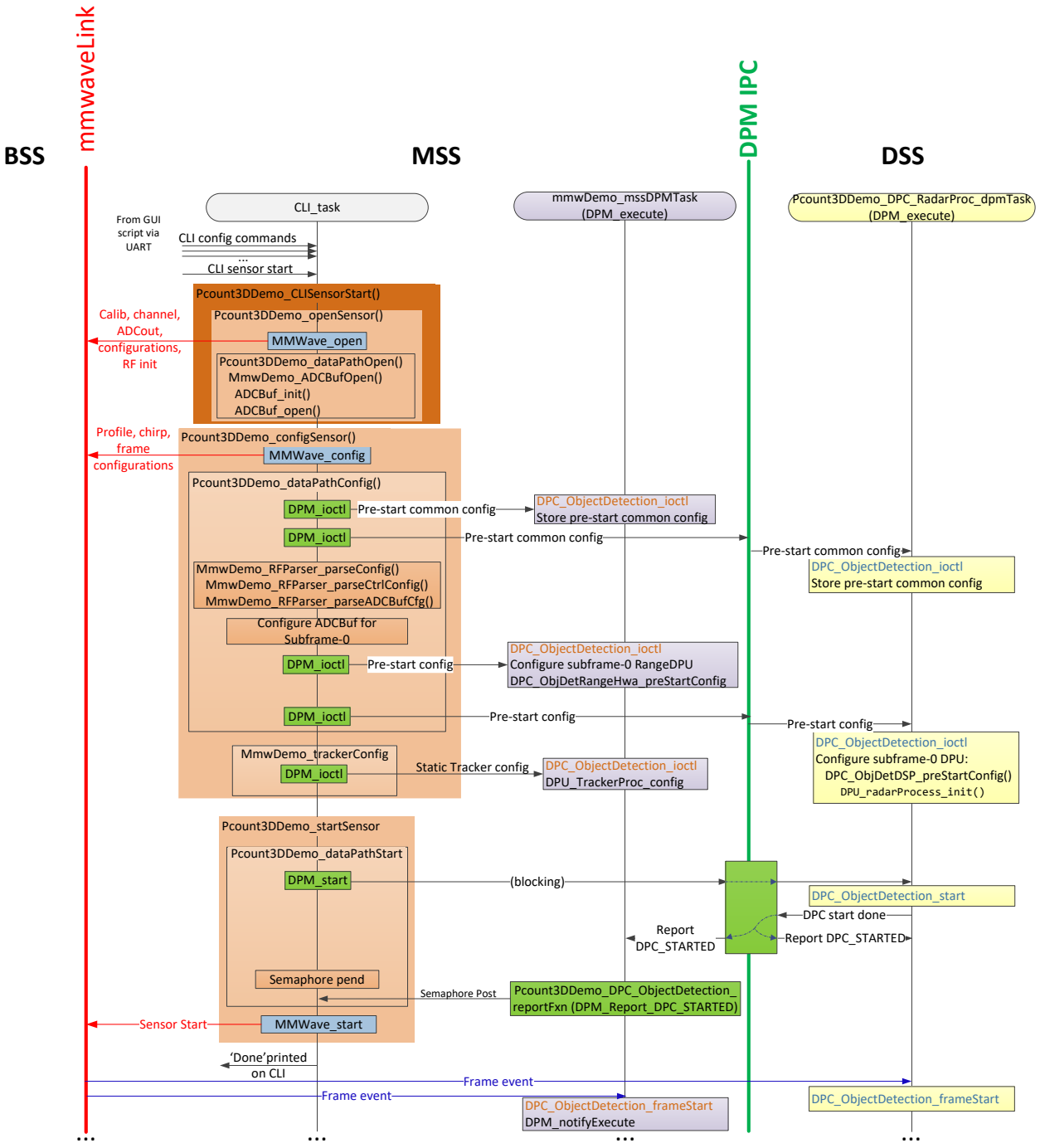


Figure 7 – System execution flow – configuration sequence.

After receiving all required CLI commands, followed by "sensorStart" command, the CLI task performs the configuration sequence. This includes sensor configuration, (MMWave_open, MMWave_config, ADCbuff_init, ADCbuff_open), parsing of the CLI commands and sending the configuration parameters to DPCs on both domains MSS and BSS. Note that the pre-start common configuration message is currently irrelevant since it sends only one common parameter, the number of sub-frames set to one. The configuration parameters are sent in the pre-start configuration message. The configuration

parameters are sent using the DPM_ioctl() API which is invoked from Pcount3DDemo_DPM_ioctl_blocking() which is waiting on a semaphore until the response to the configuration message is reported.

On the MSS side the DPM registered function DPC_ObjectDetection_ioctl() is called. It calls DPC_ObjDetRangeHwa_preStartConfig() which allocates the radar cube matrix in L3 memory, and further calls DPC_ObjDetRangeHwa_rangeConfig() , which further calls the Range DPU configuration function DPU_RangeProcHWA_config().

On the DSS side the DPM registered function DPC_ObjectDetection_ioctl() is called. It calls DPC_ObjDetDSP_preStartConfig() which further calls the Radar Processing DPU initialization function DPU_radarProcess_init() .

After the configuration has been confirmed from both domains, the CLI task calls DPM_Start(). This results in the execution of DPM registered DPC functions DPC_ObjectDetection_start() on both domains. On MSS domain this function sends the trigger command to the Range DPU resulting in the execution of rangeProcHWA_TriggerHWA(). This function configures and triggers the HWA accelerator and sets it ready for chirp procesing. On DSS domain the function with the same name only resets the frame start counter.

After the DPM_start () is confirmed (CLI task waiting on the semaphore until the response is received), the CLI task issues “Sensor start” command to BSS.

4.7.3 Execution flow - Per Frame

The system execution flow during one frame period is shown in Figure 8. At the beginning of the frame BSS will send frame start event triggering DPM which sends the start event to both DPCs on MSS and DSS.

- On the MSS side the DPM registered function DPC_ObjectDetection_frameStart() is called which calls the DPM_NotifyExecute(), which causes the DPM to invoke the registered execute function DPC_ObjectDetection_execute(). This function further calls the DPU_RangeProcHWA_process() in which the DPM task stays pending on a semaphore until the last chirp of the frame has been processed by the HWA accelerator. At this point DPC_ObjectDetection_execute() calls DPM_relayResult() with the argument containing the address of the Radar Cube matrix. This function sends the message to DSS side that the range processing is completed, with the address and the size of the Radar Cube matrix. This is a non-blocking call, and DPC_ObjectDetection_execute() further sends the trigger command to the Range DPU resulting in the execution of rangeProcHWA_TriggerHWA() and setting it ready for the next frame.
- On the DSS side DPM_relayResult message causes DPM to call the registered function DPC_ObjectDetection_dataInjection() , which calls DPM_notifyExecute(), causing the execution of DPC_ObjectDetection_execute(), which in turn calls DPU_radarProcess_process() that runs the low level signal processing chain.

Once the low level signal processing chain is completed the DSS DPM Task sends the processing results, (point cloud list, and statistics information) to the MSS side by calling DPM_sendResult(). The function is called with the flag isAckNeeded set indicating that an acknowledgment is needed after the results have been passed to the MSS side.

Just after the results have been sent, all shared data with R4F in L3 have to be written-back and cache prepared for the next frame for new radar cube data from HWA. The whole cache is written back and invalidated by calling cache_wbInvAllL2Wait().

On the MSS side the DPM report function Pcount3DDemo_DPC_ObjectDetection_reportFxn() is called with report type DPM_Report_NOTIFY_DPC_RESULT , which calls the function Pcount3DDemo_handleObjectDetResult(). This function performs the following:

- Translates the received addresses of the point cloud data and the statistics information,
- Compresses and copies point cloud data to the R4F local memory, to gMmwMssMCB.pointCloudToUart,
- Copies point cloud data to the group tracker input, to gMmwMssMCB.pointCloudFromDSP,
- Sends the notification to DSS that all the data are local now and the shared memory is released to DSS,
- Posts the group tracker semaphore to start processing,
- Posts the UART Task semaphore to start exporting data to the Host.

On DSS side the DPC marks the end of frame.

The UART task, as a higher priority task than the tracker task, initiates data transfer to the Host via UART using UART_write() blocking API. The transfer includes the point cloud list of the current frame and the tracker data of the previous frame from its output ping/pong buffer. The task waits on the semaphore until the DMA transfer is completed, letting the tracker task to process the current frame data. At the end of the UART data transfer the transfer time is recorded for the next frame.

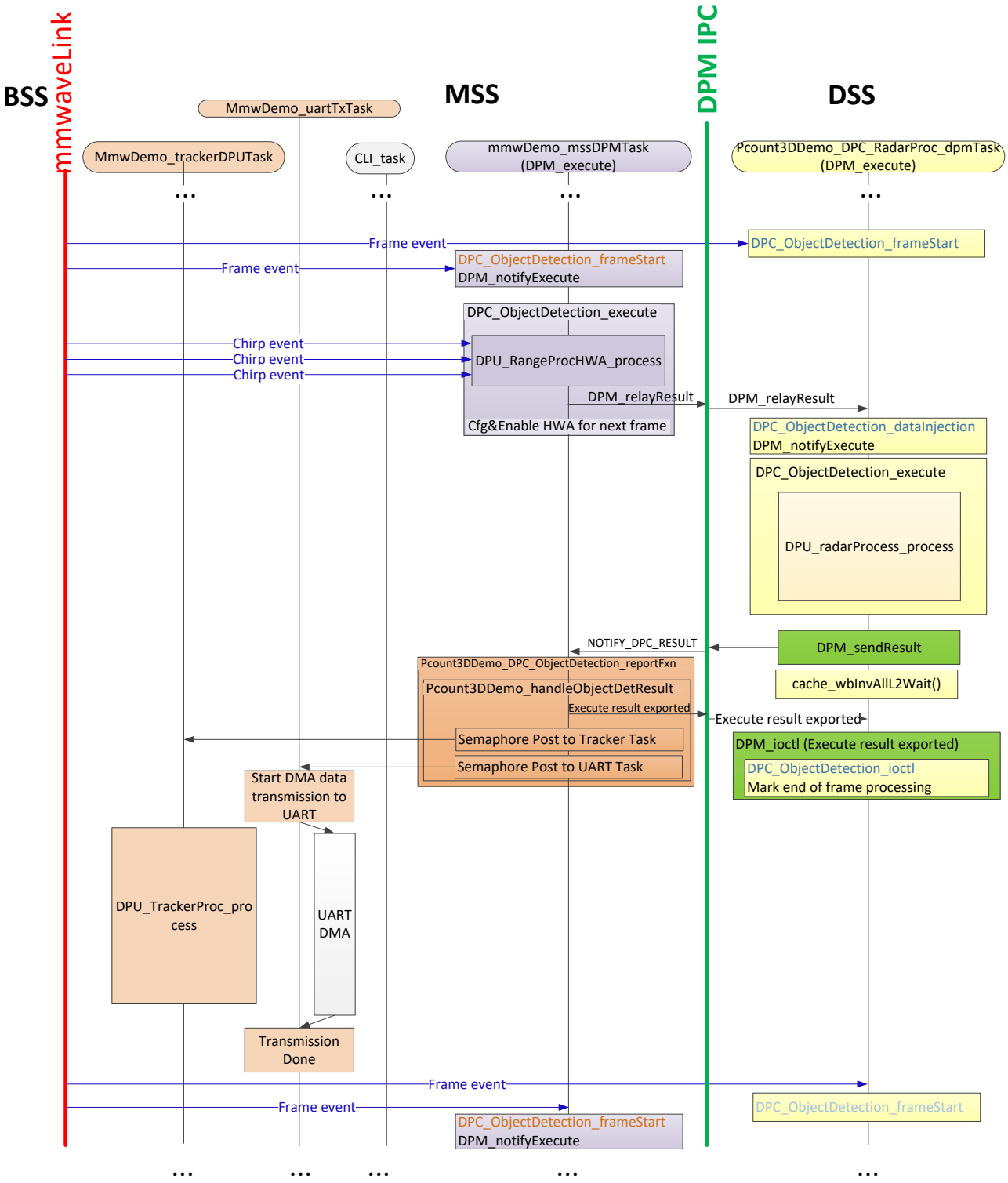


Figure 8 – System execution flow during one frame period.

4.7.4 Execution Flow - Sensor Stop and Restart

The execution flow upon receiving a sensor stop CLI command followed by sensor restart command is shown in Figure 9.

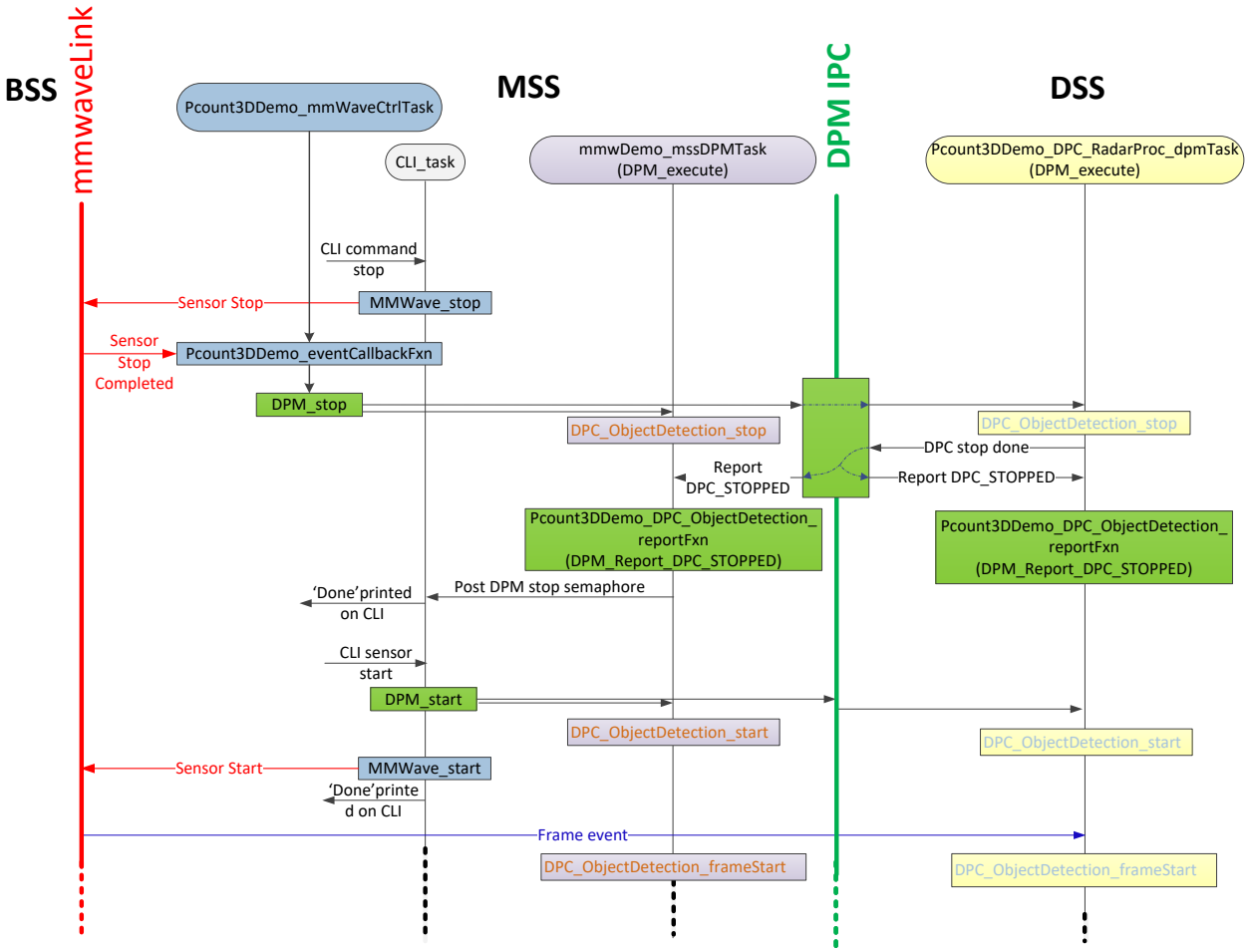


Figure 9 – Sensor stop and restart sequence

4.8 Task Model

The 3D People Counting Demo is implemented using multiple tasks running in the system. Table 8 lists all the tasks running in the system.

Task	Name in the code	Priority	Domain
DSS Initialization Task	Pcount3DDemo_dssInitTask()	1	DSS
DSS DPM Task	Pcount3DDemo_DPC_RadarProc_dpmTask()	5	DSS
MSS Initialization Task	Pcount3DDemo_initTask()	1	MSS
MMS Demo Task	mmwDemo_mssDPMTask()	5	MSS

RFE Control Task	Pcount3DDemo_mmWaveCtrlTask()	6	MSS
Tracker Task	MmwDemo_trackerDPUTask()	3	MSS
UART Task	MmwDemo_uartTxTask()	4	MSS
CLI Task	CLI_task()	2	MSS

Table 8 – Tasks running in the system

4.8.1 DSS Initialization Task

This task is created by main function and is a one-time active task whose main functionality is to initialize drivers, DPM module, and launch DSS DPM Task which provides an execution context for the DPM/DPC to run. After the initialization this task becomes terminated.

4.8.2 DSS DPM Task

This task provides the execution context for the DPM/DPC to run on DSS. In the infinite loop It invokes DPM_execute() API followed by DPM_sendResult() API. The DPM_execute() function executes the DPM module which involves handling of the reception of the IPC Messages received from its peer running on MSS, and execution of the data processing chain (DPC). The DPM_sendResult() function sends the processing chain results to the DPM running on MSS.

4.8.3 MSS Initialization Task

This task is created by main function and is a one-time active task whose main functionality is to initialize drivers, MMWave module, DPM module, open UART and data path related drivers (ADCBUF), and launch other tasks running on MSS.

4.8.4 MSS DPM Task

This task provides the execution context for the DPM/DPC to run on MSS. It calls in an endless loop DPM_execute() API. This executes the DPM module which involves handling of the reception of the IPC Messages received from its peer running on DSS, and execution of the data processing chain (DPC).

4.8.5 mmWave Control Task

This is mmWave control execution task. It provides execution context for the mmWave control. It calls in an endless loop the MMWave_execute() API. It should have priority higher than any other task which uses the mmWave control API.

4.8.6 Tracker Task

This task provides the execution context for the Tracker DPU to run on MSS. This task should have lower priority than the UART Task.

4.8.7 UART Task

This task controls the transfer of point the cloud data along with the tracker data to the host. Data transfers are done using UART_write() blocking API that uses DMA for data transfer. During the transfer the task is in a pending state allowing other tasks such as a Tracker Task to be executed in parallel with the data transfer to the Host. This task has to be higher priority than the Tracker Task.

4.8.8 CLI Task

The CLI task provides the execution context for the command line interface. The CLI utility library is a part of mmWave SDK. The library provides a simple CLI console over the specified serial port. It includes simple command parser. In addition to mmWave control commands, the additional custom commands are registered for this demo. The related code is located in pcount3D_cli.c .

4.8.9 Timing Diagram

The typical task activity timing diagram during one frame period is illustrated in Figure 10.

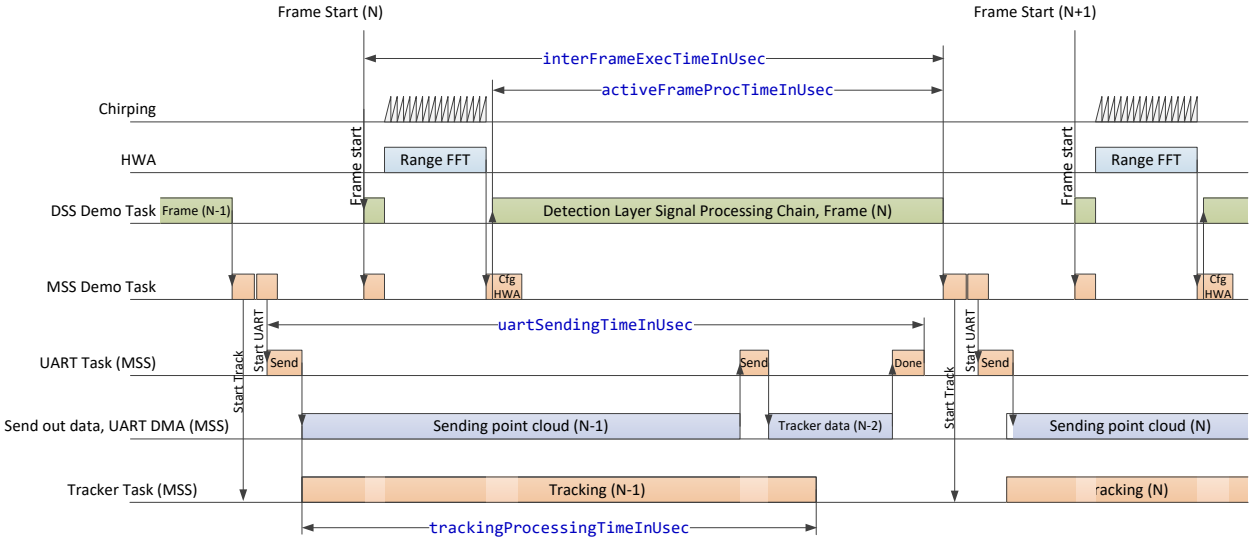


Figure 10 – Task activity timing diagram

5 Radar Processing DPU – Details

This section provide more details on the two main APIs: DPU_radarProcess_init() and DPU_radarProcess_process().

5.1 DPU_radarProcess_init()

This function is called upon pre start configuration command from DPC configuration function DPC_ObjDetDSP_preStartConfig(). It allocates memory to store its internal data object. Also It allocates buffers used the processing chain. It also calls the initialization APIs of CFAR and 2D Capon BF modules RADARDEMO_aoaEst2DCaponBF_create() and RADARDEMO_detectionCFAR_create(). Figure 11 shows its instance structure with links to the main elements.

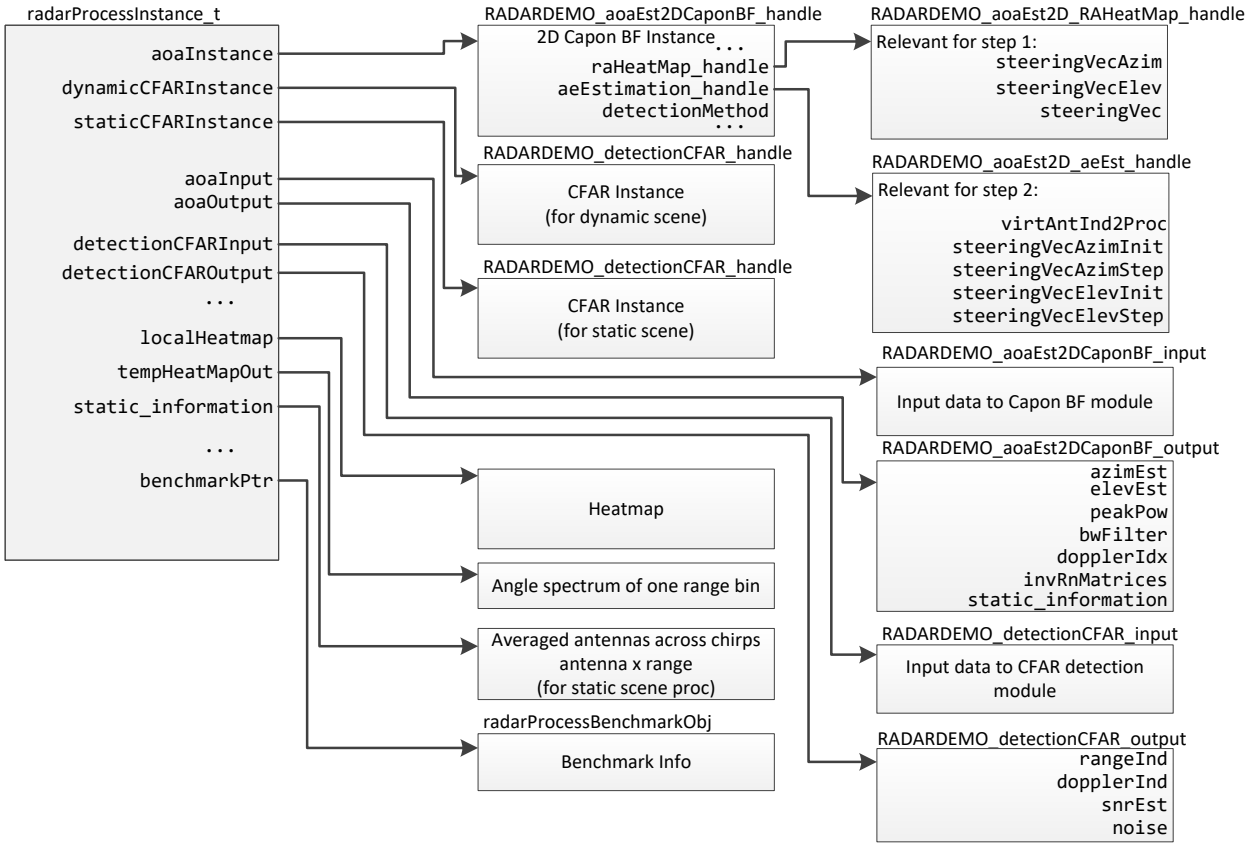


Figure 11 - Radar Processing DPU Instance

At the end of the initialization, the function prints to the CCS console output window the memory addresses of the allocated objects, Figure 12. By pasting these addresses (along with the cast text) into the CCS expression window all the object fields will be easily explored, Figure 13. This may be helpful for debugging.

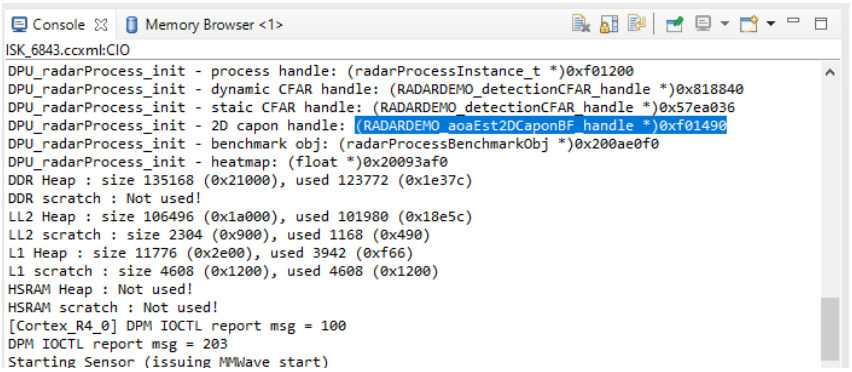


Figure 12 – CCS logs addresses of dynamically allocated radar DPU objects

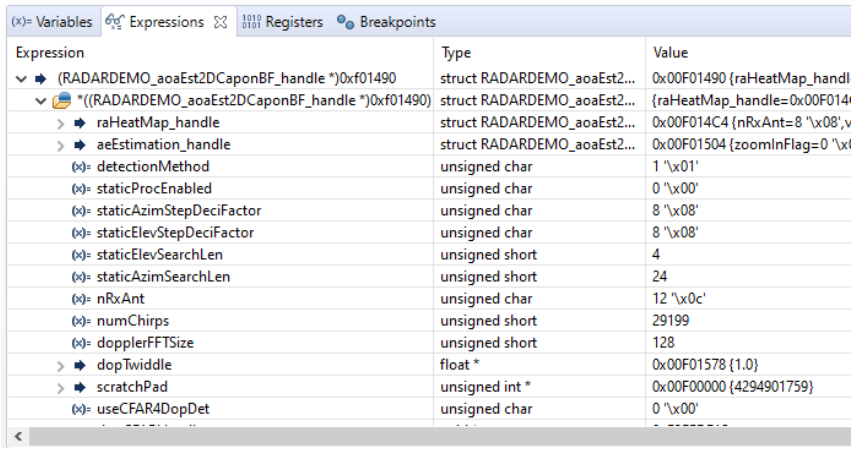


Figure 13 – Example: Capon beam forming object in CCS expression window

5.2 DPU_radarProcess_process()

This function is called per frame and all the detection layer processing on DSS is executed within this function call. The high level processing flow diagram is shown in Figure 14. The processing consists of two parts, the dynamic scene processing and if enabled, the static scene processing. It executed per frame. The input is the radar cube matrix, and the output is a single list of detected points. The function call two signal processing modules 2D Capon beamforming module (blocks in green) and CFAR detection module (blocks in blue).

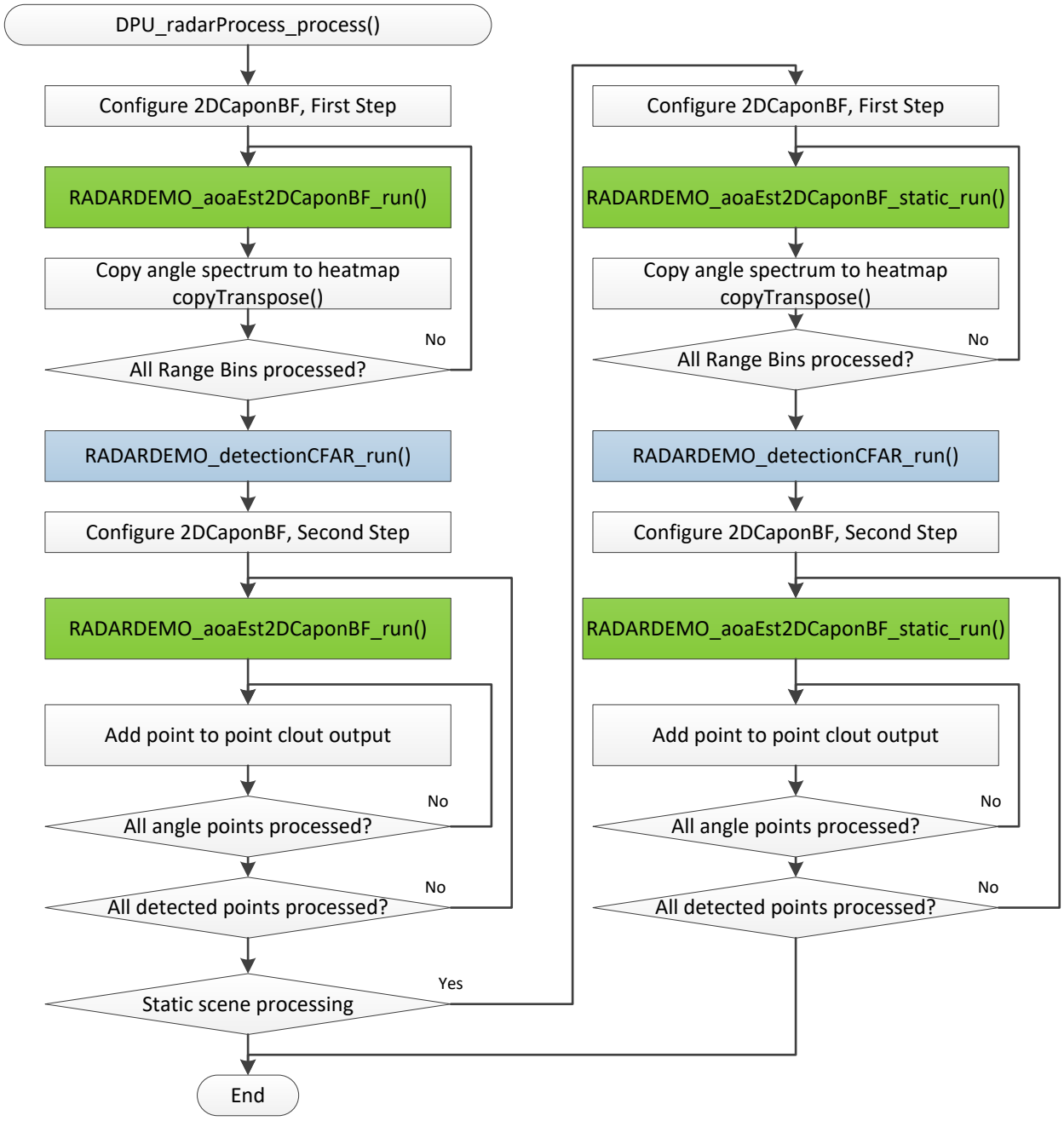


Figure 14 – DPU_radarProcess_process() - flow diagram

6 Detection Layer Signal Processing Modules

The low level signal processing on DSS is implemented using two signal processing modules: 2D Capon Beamforming module, and CFAR detection module.

6.1 2D Capon Beamforming module

In the processing chain this module is executed before the CFAR detection, (processing step 1), and also after the CFAR detection, (processing step 2). Before detection, it generates 2D or 3D heatmap used for the point cloud detection and estimation. After detection it completes the angle estimation and estimates the velocity of detected points. It is used in both signal processing chains, wall-mount and ceil-mount.

The APIs are

- RADARDEMO_aoaEst2DCaponBF_create()
- RADARDEMO_aoaEst2DCaponBF_delete()
- RADARDEMO_aoaEst2DCaponBF_run()
- RADARDEMO_aoaEst2DCaponBF_static_run()

6.1.1 RADARDEMO_aoaEst2DCaponBF_create()

This function initializes and configures the module. It allocates memory for its internal object and data. It calculates pre-calculates steering vectors.

6.1.1.1 Steering vector design

In the 3D people counting demo, the Capon beamforming algorithm, a.k.a. the minimum variance distortionless response (MVDR) spectral estimator, is used for high-resolution angle of arrival estimation step. In this algorithm, the beamforming weights steered to any azimuth (φ) and elevation (θ) angle for an N-element array with element spatial locations $\mathbf{p_n} = \left[p_{x_n}\ p_{y_n}\ p_{z_n}\right]^{\boldsymbol{T}}, n = 1,2,\ldots,N$ are defined as

$$\boldsymbol{a}(\varphi,\theta) = \left[e^{jk\mathbf{a}_{\varphi,\theta}^{T}\mathbf{p}_1}, e^{jk\mathbf{a}_{\varphi,\theta}^{T}\mathbf{p}_2}, \ldots, e^{jk\mathbf{a}_{\varphi,\theta}^{T}\mathbf{p}_N}\right], \tag{1}$$

where $k = 2\pi/\lambda$ is the wavenumber at wavelength λ, and $\mathbf{a}_{\varphi,\theta}^{T}$ is the unit vector pointing in the assumed direction of field propagation, which can be expressed as (refer to the book in [3] for details)

$$\mathbf{a}_{\varphi,\theta} = \begin{bmatrix} \cos(\theta)\sin(\varphi) \\ \cos(\theta)\cos(\varphi) \\ \sin(\theta) \end{bmatrix}. \tag{2}$$

Figure 15 illustrates the system geometry of a single reflection point *n*. The azimuth (φ) is defined as the angle from the y-axis to the orthogonal projection of the position vector onto the xy-plane. The angle is positive, going from the y-axis toward the x-axis. The elevation (θ) is defined as the angle from the projection onto the xy-plane to the vector. The angle is positive, going from the xy-plane to the z-axis.

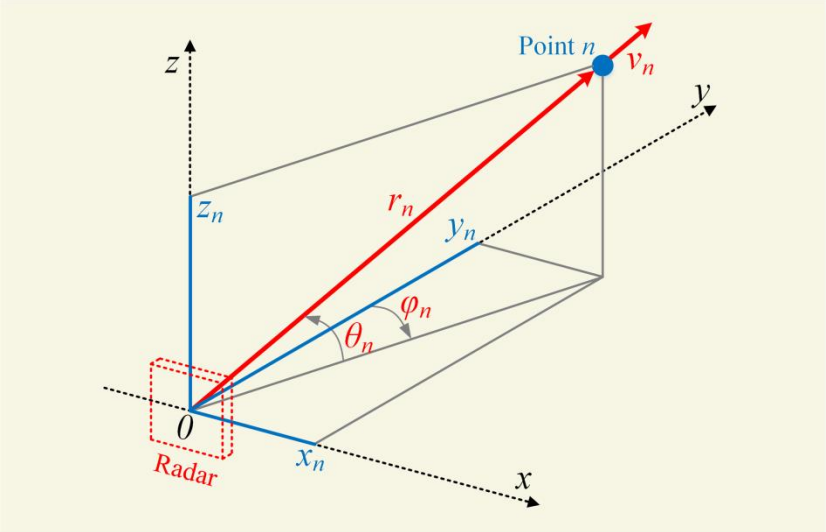


Figure 15 - The system geometry of the point cloud in the 3D coordinate system.

The numbering of the physical transmit and receive antennas is shown in Figure 16.

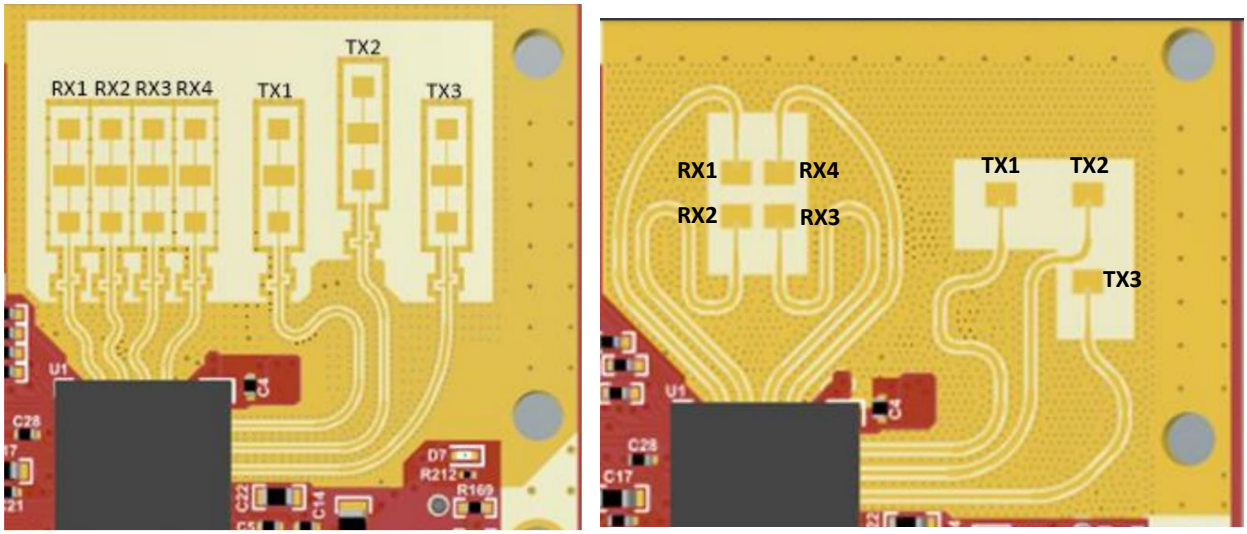


Figure 16 – Physical antenna ordering for ISK EVM (left) and ODS EVM (right)

The virtual antenna coordinates and the order of antennas in Radar Cube memory generated by the Range DPU are shown in Figure 17. Numbers in the circles represent the antenna order. Note that this order holds if the transmit antenna order in TDM-MIMO scheme matches their physical order. On the ODS EVM the Rx antennas Rx1 and, Rx4 are fed from the opposite side compared to Rx2 and Rx3, therefore the phase rotation of 180 degrees has to be applied either to Rx1 and Rx4 or to Rx2 and Rx3.

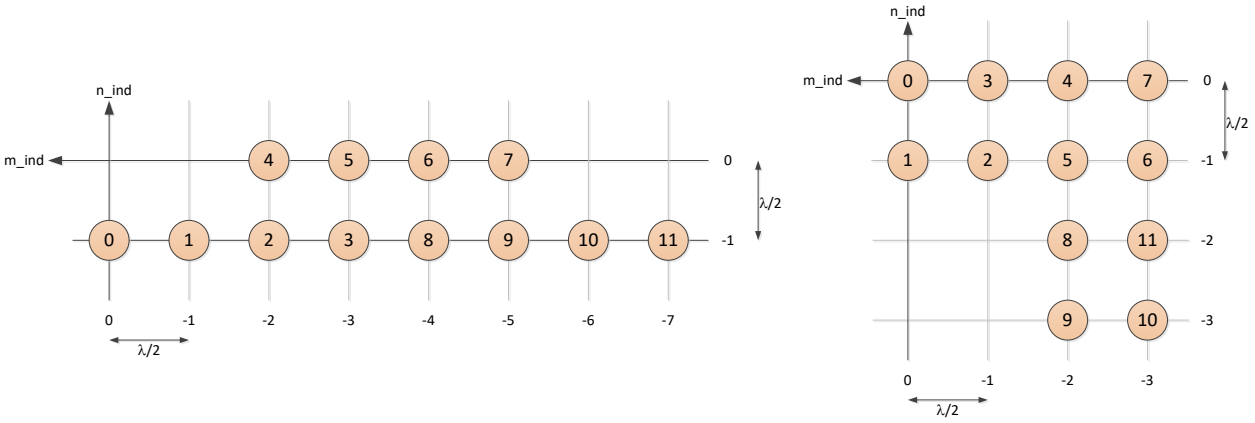


Figure 17 – Virtual antenna m and n coordinates and the order in Radar Cube matrix for ISK (left) and ODS (right).

As illustrated in Figure 17, both EVMs have 2D planar arrays with uniformly spaced antenna elements in xz-domain (according to the reference coordinate system in Figure 15). For these EVMs, the virtual antenna element locations in the y-domain is zero and the inter-element spacing of the antennas in the xz-plane is $\lambda/2$. Therefore, the element position vectors in (1) can be defined as

$$\mathbf{p_n} = (\lambda/2)[x_n\ 0\ z_n]^{\boldsymbol{T}}, \quad x_n = 0,1,\dots(N_x - 1)\ \ and\ y_n = 0,1,\dots(N_z - 1) \tag{3}$$

where N_x and N_z are the number of virtual receive antenna elements in the x and z direction, respectively. If we define direction cosines ν (nu) and μ (mu) with respect to x and z axes using (2) as

$$\begin{matrix} \nu = \cos(\theta)\sin(\varphi) \\ \mu = \sin(\theta) \end{matrix}, \tag{4}$$

and take (2),(3), and (4) into (1), the steering vector can be written in the nu-mu domain as

$$\boldsymbol{a}(\nu,\mu) = \left[1, e^{j\pi\nu}, e^{j\pi 2\nu}, e^{j\pi\mu}, e^{j\pi 2\mu}, \dots, e^{j\pi(N_x-1)\nu} e^{j\pi(N_z-1)\mu}\right]. \tag{5}$$

Therefore, the steering vector in (1) can be calculated in any (ν,μ) direction and for any antenna element n as the multiplication of the corresponding elements in the following 1D steering vectors in ν and μ domain

$$\begin{matrix} \boldsymbol{a}(\nu) = \left[1, e^{j\pi\nu}, e^{j\pi 2\nu}, \dots, e^{j\pi(N_x-1)\nu}\right] \\ \boldsymbol{a}(\mu) = \left[1, e^{j\pi\mu}, e^{j\pi 2\mu}, \dots, e^{j\pi(N_z-1)\mu}\right] \end{matrix}. \tag{6}$$

The antenna coordinates m_ind (azimuth dimension) and n_ind (elevation dimension), and the phase rotation are

For ISK EVM:

m_ind = [0 -1 -2 -3 -2 -3 -4 -5 -4 -5 -6 -7]

n_ind = [0 -1 -1 0 0 -1 -1 0 -2 -3 -3 -2]
Phase rotation = [1 1 1 1 1 1 1 1 1 1 1 1]

For ODS EVM:

m_ind = [0 0 -1 -1 -2 -2 -3 -3 -2 -2 -3 -3]
n_ind = [0 -1 -1 0 0 -1 -1 0 -2 -3 -3 -2]
Phase rotation = [-1 1 1 -1 -1 1 1 -1 -1 1 1 -1]

The indices m_ind and n_ind are specified by CLI configuration commands antGeometry0 and antGeometry1 respectively. The phase rotation is specified by CLI command antPhaseRot.

For the first processing step, for range-angle heatmap generation, a full set of steering vectors is generated including all virtual antennas to save cycles. In the second processing step, after the CFAR detection, the steering vectors are generated on the fly. To be able to generate them on-the fly, the steering vectors are handled in mu-nu domain, instead of angle domain (theta-phi domain). After the completion of the angle estimation, the asin() and acos() functions (from 674x MATHLIB library) are called to convert to the final spherical point cloud format. The antenna geometry, antenna phase rotation, and board related phase bias are all taken care of in the steering vector generation. No run-time calculation is involved.

6.1.1.1.1 Steering vectors calculation for wall-mount scenario

For wall-mount scenario two sets of steering vectors are pre-calculated and stored at the initialization time: azimuth steering vectors, and elevation steering vectors. The steering vectors are calculated for full set of virtual antennas, (12 antennas).

Figure 18 and Figure 19 illustrate the steering vector grid in mu-nu domain and theta-phi domain respectively. The pre-calculated azimuth steering vectors are shown as blue dots and the pre-calculated elevation steering vectors are shown as red dots. The other dots, shown in gray, are generated on the fly in the second processing step for the elevation estimation. The vectors are evenly spaces in the mu-nu domain, while in theta-phi domain the step size is not constant and increases towards the edges of FOV. The number of steering vectors depends on the configuration parameters: the field of view and the angle step size. These parameters are specified in the CLI configuration file for both azimuth and elevation. For example, as shown in Figure 18 and Figure 19, for the following input parameters: azimuth FOV = +/- 70°, azimuth step size = 0.75°, elevation FOV = +/- 20°, and elevation step size = 0.75°, the number of steering vectors in azimuth direction is 187, and the number in elevation direction is 54.

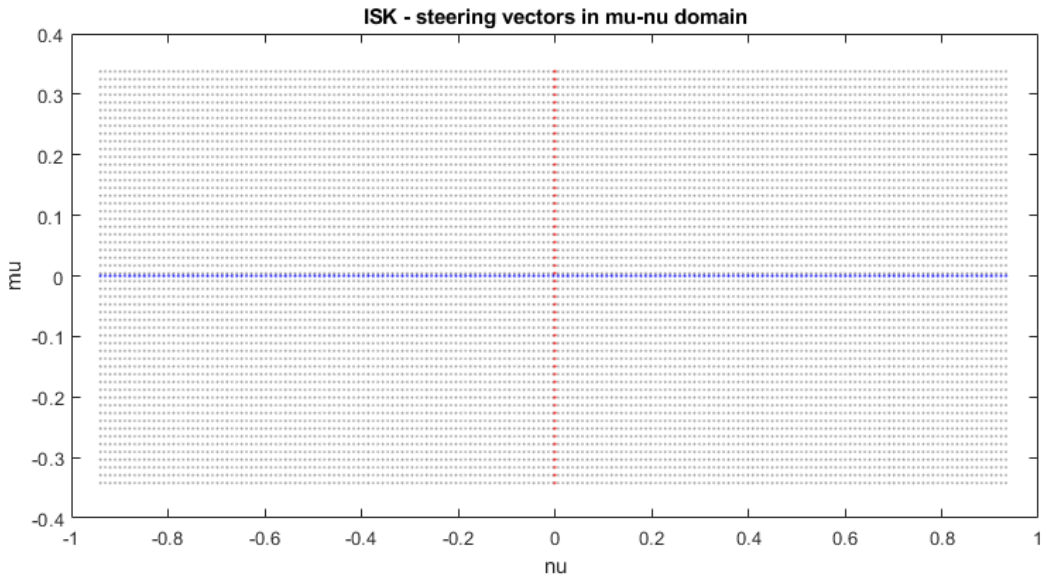


Figure 18 – Steering vector grid in mu-nu domain (wall-mount scenario)

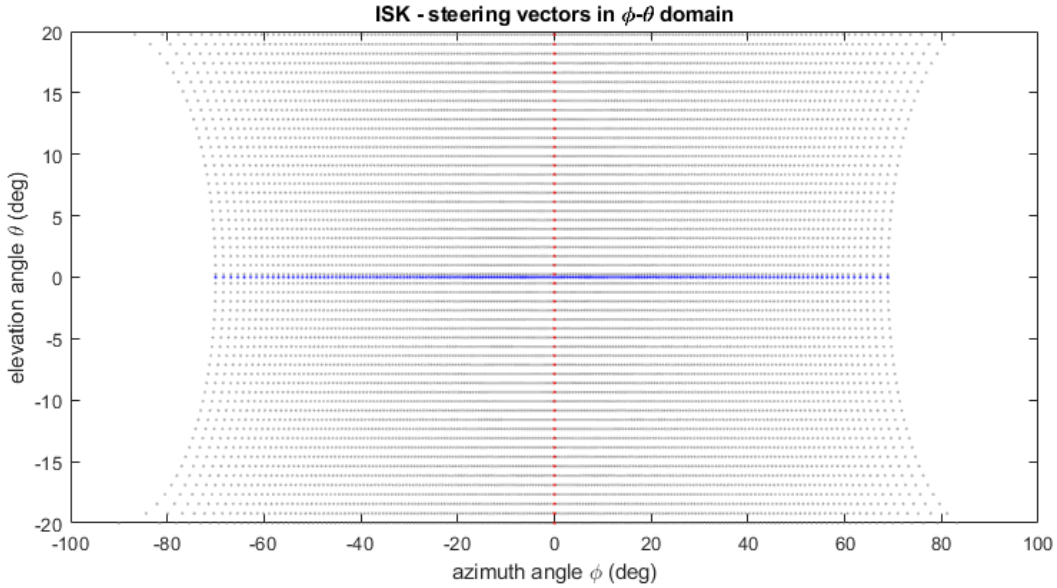


Figure 19 – Steering vector grid in theta-phi domain (wall-mount scenario)

In the first processing step, range-angle heatmap generation, only azimuth-antennas, 8 antennas of the pre-calculated azimuth steering vectors are used. In the second processing step all 12 antennas are used. The elevation steering vectors are calculated by multiplying pre-calculated elevation steering vectors with the azimuth steering vector corresponding to the detected point.

Pre-calculated steering vectors are stored in raHeatMap_handle->steeringVecAzim and raHeatMap_handle->steeringVecElev, allocated in L2 memory heap.

6.1.1.1.1.1 Azimuth steering vectors calculation

Base on the configuration parameters azimuth angle FOV φ_{FOV} and angle step size φ_{Step}, the azimuth steering vectors are calculated as

Number of azimuth steering vectors

$$N_A = \frac{2\varphi_{FOV}}{\varphi_{Step}}$$

Initial value of nu and nu step

$$\upsilon_{init} = -\sin\varphi_{FOV}, \quad \upsilon_{step} = \frac{-2\upsilon_{init}}{N_A}$$

nu grid

$$\upsilon_i = \upsilon_{init} + i \cdot \upsilon_{step}\ ,\ i = 0, \dots, N_A - 1$$

Azimuth steering vectors

$$\mathbf{a}(\upsilon_i) = \left[p_0 e^{j\pi m_0 \upsilon_i}, \dots, p_{N_r-1} e^{j\pi m_{N_r-1} \upsilon_i},\right]^T\ ,\ i = 0, \dots, N_A - 1$$

were

N_r is number of virtual antennas ($N_r = 12$),

$m_k, k = 0, \dots, N_r - 1$ are antenna geometry indices, m_ind.

$p_k, k = 0, \dots, N_r - 1$ are 180° phase rotation coefficients.

Note that azimuth steering vectors include phase rotation coefficients.

6.1.1.1.1.2 Elevation steering vectors calculation

Based on the configuration parameters, the elevation angle FOV θ_{FOV} and the angle step size θ_{Step} , the elevation steering vectors are calculated as

The number of elevation steering vectors

$$N_E = \frac{2\theta_{FOV}}{\theta_{Step}}$$

Initial value of mu and mu step

$$\mu_{init} = -\sin\theta_{FOV}, \quad \mu_{step} = \frac{-2\mu_{init}}{N_E}$$

nu grid:

$$\mu_i = \mu_{init} + i \cdot \mu_{step}\ ,\ i = 0, \dots, N_E - 1$$

Elevation steering vectors:

$$\mathbf{a}(\mu_i) = \left[e^{j\pi n_0 \mu_i}, \ldots, e^{j\pi n_{N_r-1}\mu_i},\right]^T,\ i = 0, \ldots, N_E - 1$$

were

N_r is number of virtual antennas ($N_r = 12$)

$n_k, k = 0, \ldots, N_r - 1$ are antenna geometry indices n_ind.

Note that in the code the values μ_{init}, μ_{step}, υ_{init}, υ_{step}, are saved normalized to π.

6.1.1.1.2 Steering vectors calculation for ceil-mount scenario

For ceil-mount scenario the 3D range-azimuth-elevation heatmap is generated using 2D coarse azimuth-elevation grid of steering vectors which are pre-calculated and stored at the initialization time. The steering vectors are generated for full set of virtual antennas, ($N_r = 12$). After the CFAR detection, a finer heatmap is generated around detected points on a denser grid of steering vectors. These steering vectors are calculated on the fly based on the four pre-calculated steering vectors and the coarse steering vector corresponding to a detected point. Figure 20 and Figure 21 illustrate the steering vector grid in mu-nu domain and theta-phi domain respectively. The pre-calculated steering vectors of the coarse grid are shown as blue dots. The four steering vectors used for zoom-in heatmap generation are shown as red dots. The dots in gray are the example of on the fly generated zoomed-in steering vectors for a detected point at nu,mu=(0.5,0.4). The configuration parameters in this example are: azimuth FOV = +/- 69°, azimuth step size = 7°, elevation FOV = +/- 62°, and elevation step size = 7°, zoom-in factor = 5.

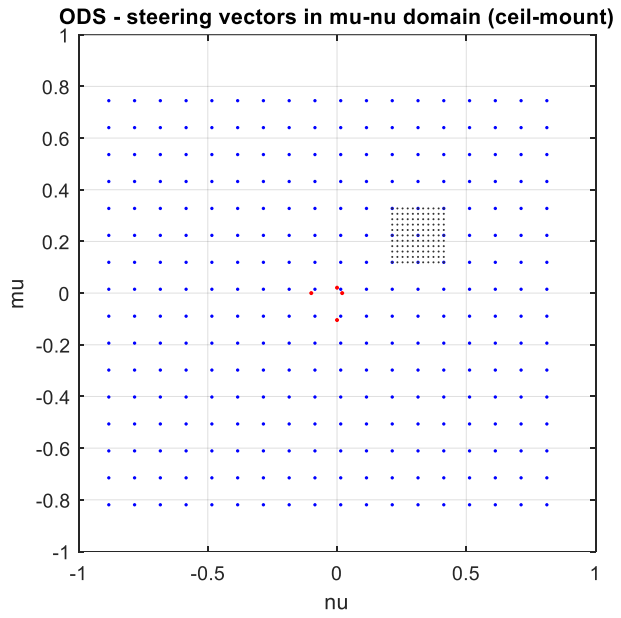


Figure 20 – Steering vector grid in mu-nu domain (ceil-mount scenario), pre-calculated steering vectors for coarse heatmap (blue dots), pre-calculated steering vectors for zoomed-in heatmap (red dots).

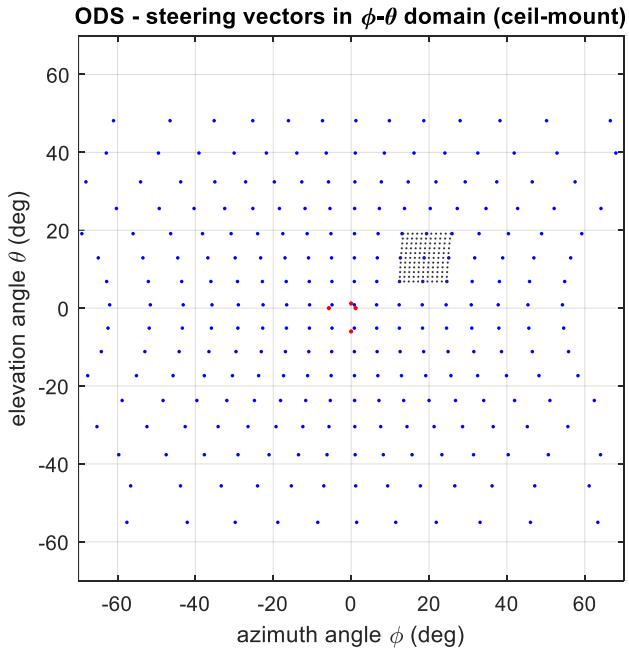


Figure 21 – Steering vector grid in theta-phi domain (ceil-mount scenario)

6.1.1.1.2.1 Coarse azimuth-elevation steering vector calculation

Base on the configuration parameters azimuth angle FOV φ_{FOV} , θ_{FOV}, step size φ_{Step} (equal in both directions) the steering vectors are calculated as

Number of steering vectors in azimuth direction

$$N_A = \frac{2\varphi_{FOV}}{\varphi_{Step}}$$

Initial value of nu and nu step

$$\upsilon_{init} = -\sin\varphi_{FOV}, \quad \upsilon_{step} = \frac{-2\upsilon_{init}}{N_A}$$

nu grid

$$\upsilon_i = \upsilon_{init} + i \cdot \upsilon_{step}\ ,\ i = 0, \dots, N_A - 1$$

The number of steering vectors in elevation direction

$$N_E = \frac{2\theta_{FOV}}{\theta_{Step}}$$

Initial value of mu and mu step

$$\mu_{init} = -\sin\theta_{FOV}, \quad \mu_{step} = \frac{-2\mu_{init}}{N_E}$$

mu grid:

$$\mu_i = \mu_{init} + i \cdot \mu_{step}\ ,\ i = 0, \dots, N_E - 1$$

Coarse azimuth-elevation steering vectors

$$\mathbf{a}(\mu_{i2}, \upsilon_{i1}) = \left[p_0 e^{j\pi(m_0\upsilon_{i1} + n_0\mu_{i2})}, \dots, p_{N_r-1} e^{j\pi(m_{N_r-1}\upsilon_{i1} + n_{N_r-1}\mu_{i2})},\right]^T, i_2 = 0, \dots, N_E - 1,\ i_1 = 0, \dots, N_A - 1$$

were

N_r is number of virtual antennas (=12),

$m_k, k = 0, \dots, N_r - 1$ are antenna geometry indices, m_ind.

$n_k, k = 0, \dots, N_r - 1$ are antenna geometry indices n_ind.

$p_k, k = 0, \dots, N_r - 1$ are 180° phase rotation coefficients.

The coarse steering vectors are stored in raHeatMap_handle->steeringVec allocated in L2 memory heap. The steering vectors are stored in order [elevation index][azimuth index][antenna index].

6.1.1.1.2.2 Zoom-in steering vectors

Zoom in steering vector grid size is $N_{zoom\_in} \times N_{zoom\_in}$ where $N_{zoom\_in} = 2M_{zoom\_in} + 1$ and

M_{zoom_in} is zoom-in factor specified in CLI configuration. As mentioned before only four steering vectors are pre-calculated, the initial value, and the step, for each direction.

$$v'_{init} = -v_{step}, \quad v'_{step} = \frac{v_{step}}{M_{zoom\_in}}$$

$$\mathbf{b}_{\boldsymbol{azim\_init}}(v'_{init}) = \left[e^{j\pi m_0 v'_{init}}, \dots, e^{j\pi m_{N_r-1} v'_{init}}, \right]^T$$

$$\mathbf{b}_{\boldsymbol{azim\_step}}(v'_{step}) = \left[e^{j\pi m_0 v'_{step}}, \dots, e^{j\pi m_{N_r-1} v'_{step}}, \right]^T$$

$$\mu'_{init} = -\mu_{step}, \quad \mu'_{step} = \frac{\mu_{step}}{M_{zoom}}$$

$$\mathbf{b}_{\boldsymbol{elev\_init}}(\mu'_{init}) = \left[e^{j\pi n_0 \mu'_{init}}, \dots, e^{j\pi n_{N_r-1} \mu'_{init}}, \right]^T$$

$$\mathbf{b}_{\boldsymbol{elev\_step}}(\mu'_{step}) = \left[e^{j\pi n_0 \mu'_{step}}, \dots, e^{j\pi n_{N_r-1} \mu'_{step}}, \right]^T$$

These four zoom-in steering vectors are stored in the following arrays:

aeEstimation_handle->steeringVecAzimInit, aeEstimation_handle->steeringVecAzimStep,

aeEstimation_handle->steeringVecElevInit, aeEstimation_handle->steeringVecElevStep, all allocated in L1 memory heap.

The steering vectors on the zoomed-in grid are generate on the fly iteratively as

$$\mathbf{v}(0) = \mathbf{a}_{coarse\_est} \circ \mathbf{b}_{azim\_init} \circ \mathbf{b}_{elev\_init}$$

$$\mathbf{v}(i) = \mathbf{v}(i-1) \circ \mathbf{b}_{azim\_step}, \quad i = 1, \dots, N_{Ezoom\_in} - 1$$

$$\mathbf{u}(i,0) = \mathbf{v}(i)$$

$$\mathbf{u}(i,j) = \mathbf{u}(i,j-1) \circ \mathbf{b}_{elev\_step}, \quad j = 1, \dots, N_{Ezoom\_in} - 1$$

The symbol ∘ denotes element-wise vector product. This is illustrated in Figure 22. Blue circles represent coarse grid, while grey dots represent zoomed-in grid around detected point at $\mathbf{a}_{coarse\_est}$.

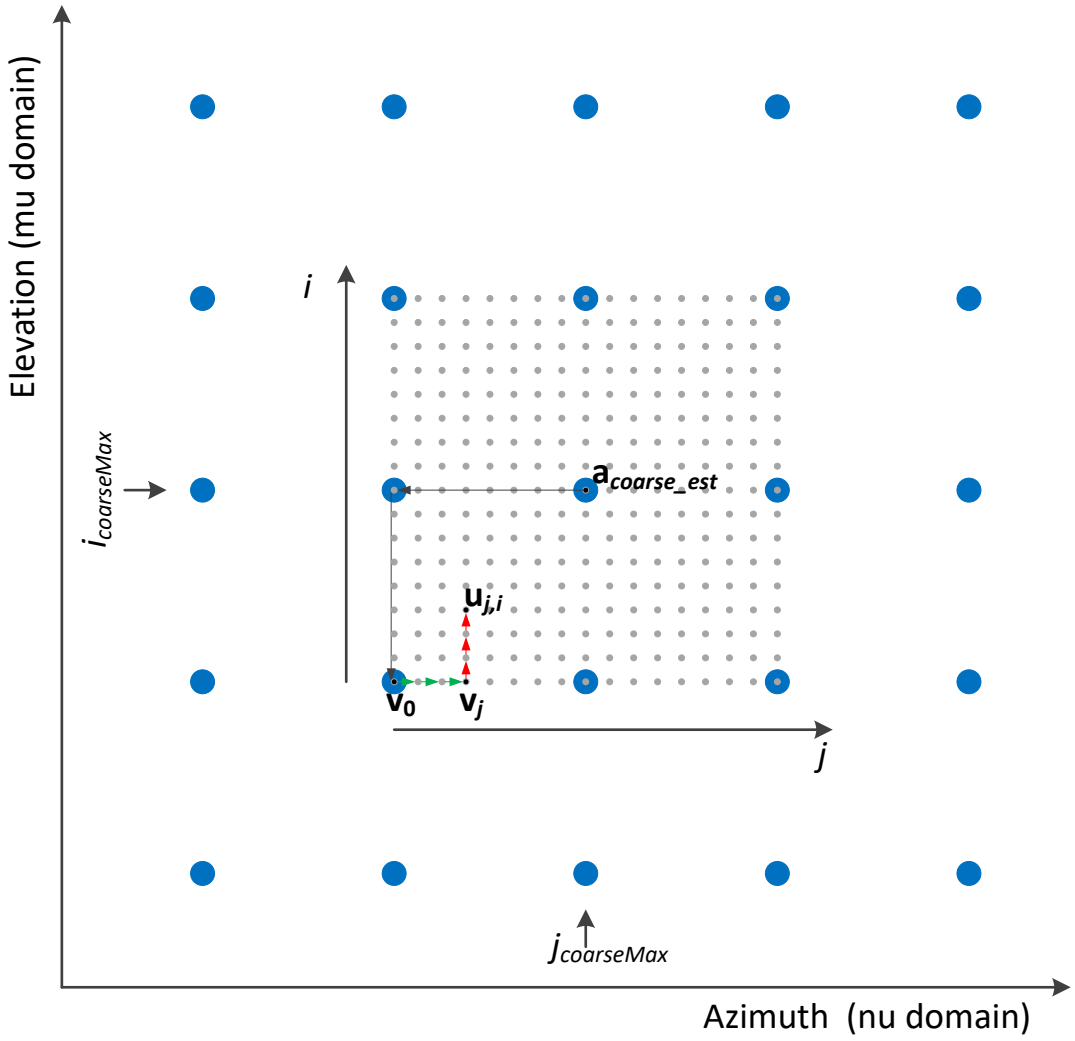


Figure 22 – Zoomed-in grid steering vector generation

6.1.2 RADARDEMO_aoaEst2DCaponBF_run()

This function generates detection heat map using 2D Capon beam forming approach. It is called per range bin. It is called in both signal processing chains, wall-mount and ceil-mount. In the processing chain it is called at two places, referred in a code as processing step 1 and processing step 2. The first processing step is before CFAR detection and the second step is called after the CFAR detection.

In step 1 it calculates per range bin

- 1D azimuth spectrum and stores it in a 2D range-azimuth heatmap (wall-mount),
- 2D azimuth-elevation spectrum and stores it in a 3D range-azimuth-elevation (ceil-mount).

In step 2 it calculates per detected point

- elevation spectrum and estimates the elevation angle (method 1),
- finer azimuth-elevation spectrum and estimates both angles (method 2),

- peak expansion (method 2),
- radial velocity of the detected points.

The function flow diagram is shown in Figure 23.

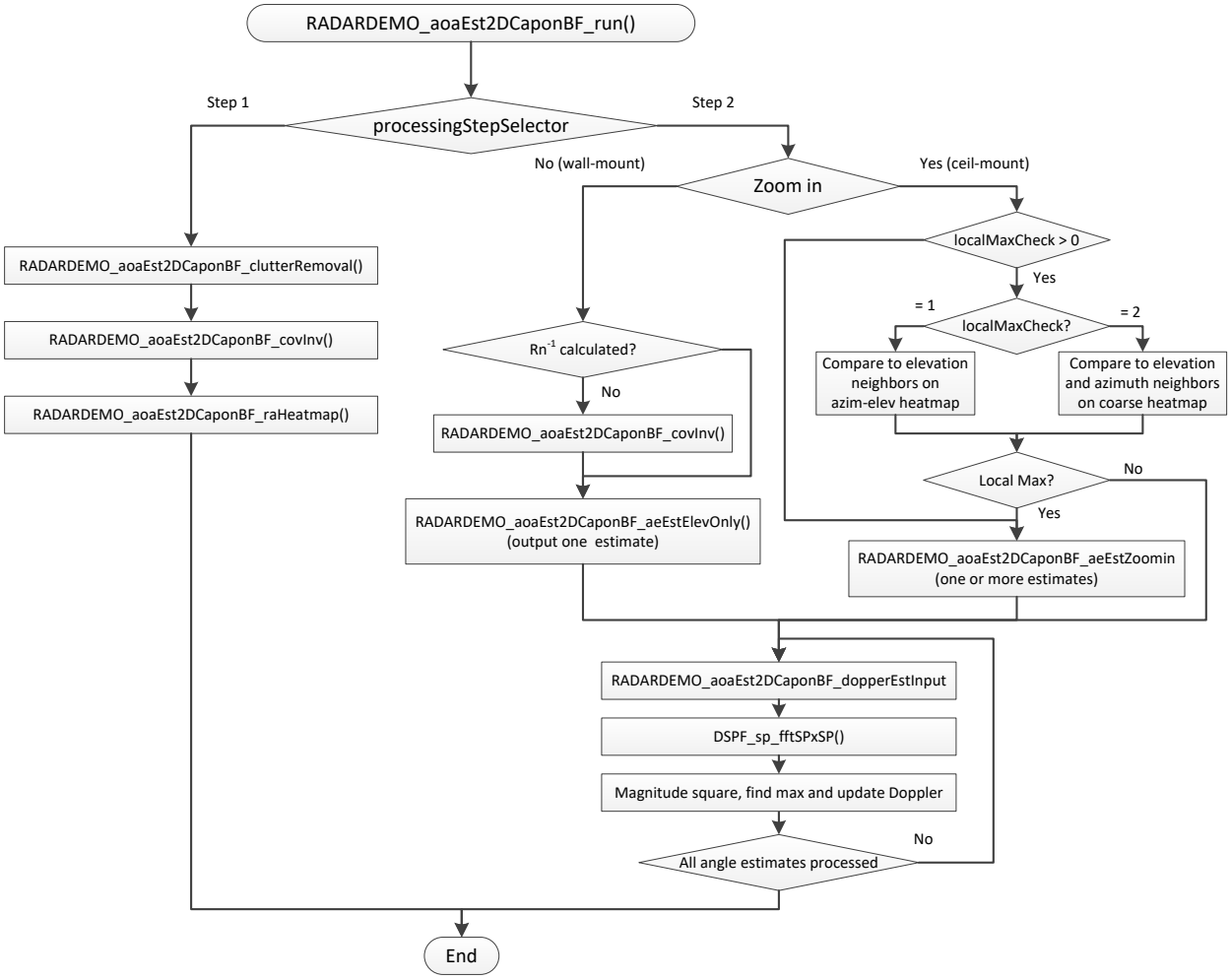


Figure 23 – 2D Capon BF run API flow diagram

6.1.2.1 Processing step 1

6.1.2.1.1 Static Clutter Removal

The static clutter removal is performed by function RADARDEMO_aoaEst2DCaponBF_clutterRemoval(). It is called within RADARDEMO_aoaEst2DCaponBF_run() in the first processing step. It is called per range bin. The static clutter removal is performed on the Radar Cube matrix located in L3. The function computes the mean values of antenna symbols across the chirps per antenna. Then it subtracts the mean values from the symbols and outputs them to the L1 scratch memory for further processing in the chain. Note that the output data are stored in the floating point format and from this point on, all the further processing is done in the floating point. It also outputs the mean values for the static scene

processing. The function writes back the updated symbols to the Radar Cube matrix in a transposed form as illustrated in Figure 24.

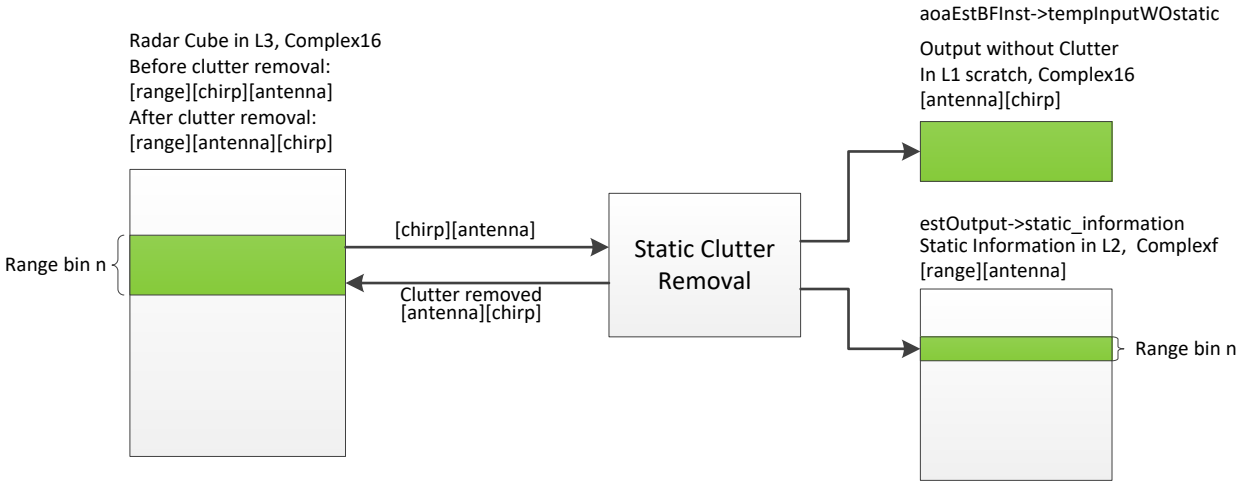


Figure 24 – Clutter removal function

6.1.2.1.2 Spatial Covariance Matrix Estimation and Inversion

The spatial covariance matrix estimation and inversion is implemented by function RADARDEMO_aoaEst2DCaponBF_covInv(). The function first calculates the covariance matrix and then depending on the input flag it computes the inverse matrix. The output matrix is stored as the upper triangular matrix, row by row in memory. This function is used in both processing steps. The covariance matrix is computed as

$$\mathbf{R}_{yy} = \frac{1}{N_c}\sum_{c=0}^{N_c-1}\mathbf{Y}_c\,\mathbf{Y}_c^H,\ \mathbf{Y}_c = \left[y_{c,0},\dots,y_{c,N_r-1}\right]^T$$

Where

N_c - is number of chirps,

$\mathbf{Y}_c$ – is array of selected antenna symbols of chirp c that are used in Capon beamforming spectrum estimation,

N_r - is number of used selected antennas.

A diagonal loading is applied to the R matrix to ensure stability

$$\mathbf{R}_{yy} = \mathbf{R}_{yy} + \gamma\frac{tr(\mathbf{R}_{yy})}{N_a}\mathbf{I}_{N_r}$$

The covariance matrix inversion is implemented using Cholesky decomposition. The method is proposed in [2]. Figure 25 illustrates the function input and output buffers.

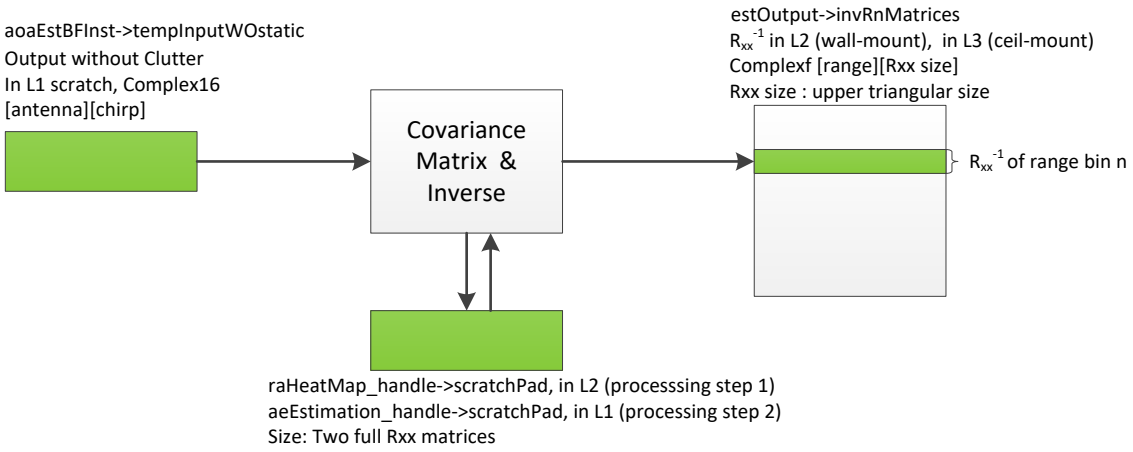


Figure 25 – Covariance matrix estimation and inverse

The full floating point implementation for matrix inversion takes 5k cycles for 8x8, or 12k cycles for 12x12.

6.1.2.1.3 Range-angle heatmap generation

This is done by function RADARDEMO_aoaEst2DCaponBF_raHeatmap(). It is called per range bin. It calculates MVDR angle spectrum according to following equations:

- for wall-mount using azimuth steering vectors, including only azimuth antennas, N_r=8

$$P_b(v_i) = \frac{1}{\mathbf{a}^H(v_i)\boldsymbol{R}_{\boldsymbol{YY}}^{-\mathbf{1}}\mathbf{a}(v_i)},\ i = 0, \dots, N_A - 1$$

- for ceil-mount using coarse azimuth-elevation steering vectors, including all antennas, N_r=12

$$P_b(\mu_{i2}, v_{i1}) = \frac{1}{\mathbf{a}^H(\mu_{i2}, v_{i1})\boldsymbol{R}_{\boldsymbol{YY}}^{-\mathbf{1}}\mathbf{a}(\mu_{i2}, v_{i1})},\ i_2 = 0, \dots, N_E - 1,\ i_1 = 0, \dots, N_A - 1$$

The calculation of one spectral point involves the calculation of vector by matrix by vector multiplication of the form $\mathbf{v}^{\boldsymbol{H}}\mathbf{A}\mathbf{v}$, where $\mathbf{v}$ is an Nx1 vector, $\mathbf{A}$ is an NxN positive semidefinite Hermitian matrix, and then taking the inverse of the product. This is the most cycle cost kernel in the implementation, because it is per range bin per angle bin, and the size of the matrix is large. It can be written into the following form:

$$\sum_{i=0}^{N-1} diag(\mathbf{A}) + \sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1} 2 * Re(a_{i,j} * v_i * conj(v_j))$$

The formula above is fully unrolled to scheduled loops to get better cycle performance at the cost of code memory usage. Currently only 4/8/12 antennas are supported fully unrolled. If there is a need to do other number of antennas, e.g., 6 or 10 antennas, additional code will need to be written in the similar fashion to get similar cycle profile at the cost of increased code memory. The final spectral point is obtained by taking the inverse of the result. The function outputs the angle spectrum to a temporary buffer that holds data only of the current range bin, as shown in Figure 26. From the temporary buffer

the function copyTranspose() copies and transposes the result into the final range-angle heatmap to be processed by CFAR. It also outputs the maximum value and stores in the array of peak values per range bin, which is used later by CFAR for side-lobe threshold calculation. The final data arrangement in the heatmap, after all the range bins have been processed, is:

- For wall-mount: [azimuth index] [range index],
- For ceil-mount: [elevation index][azimuth index][range index].

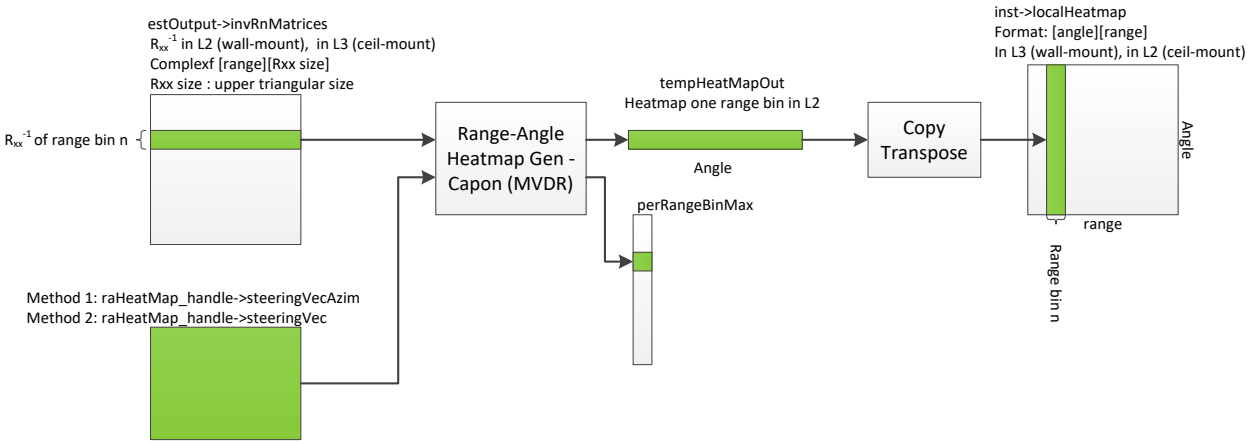


Figure 26 – Range-Angle heatmap generation

6.1.2.2 Processing step 2

The function is called after CFAR detection and it is called per detected point. It estimates the elevation (wall-mount) or refines the azimuth/elevation estimation (ceil-mount) and then it estimates radial velocity of the detected point based on a Doppler FFT.

6.1.2.2.1 Wall-mount – Elevation Estimation

In processing step 2 the inverse covariance matrix is recalculated including all virtual antennas, N_r=12, since in the first processing step N_r=8 antennas are used. In order to avoid repeated calculation, it sets the bit corresponding to the range index in the bitmask array, aeEstimation_handle->procRngBinMask, to indicate that the inverse covariance is calculated in case the following detected points have the same range index. The elevation spectrum is calculated by RADARDEMO_aoaEst2DCaponBF_aeEstElevOnly(). The relevant input/output information is shown in Figure 27.

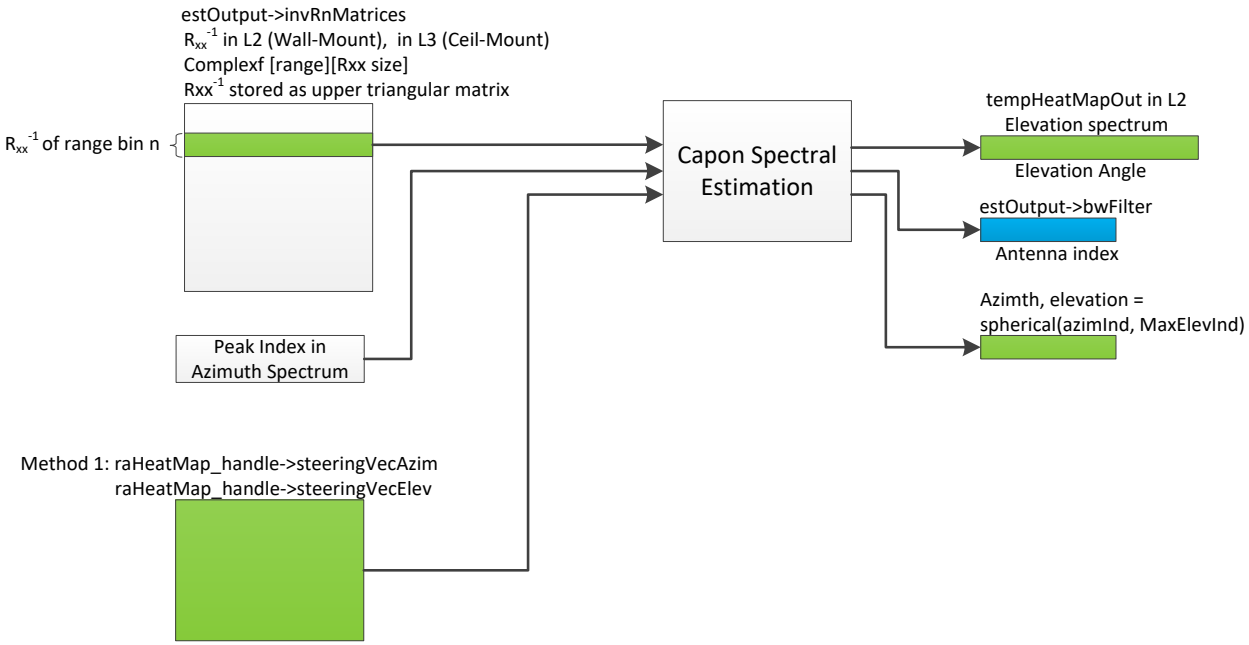


Figure 27 – Wall-mount elevation estimation

The function calculates the elevation spectrum at detected azimuth angle υ_{jMax} as

$$P_b(\mu_i, \upsilon_{jMax}) = \frac{1}{\mathbf{a}^H(\mu_i,\upsilon_{jMax})\boldsymbol{R}_{\boldsymbol{YY}}^{-\mathbf{1}}\mathbf{a}(\mu_i,\upsilon_{jMax})},\ i = 0, \dots, N_E - 1$$

Where the steering vectors are calculated on the fly as

$$\mathbf{a}(\mu_i, \upsilon_{jMax}) = \mathbf{a}(\mu_i) \circ \mathbf{a}(\upsilon_{jMax}),\ i = 0, \dots, N_E - 1$$

After finding the maximum in the elevation spectrum, the function calculates the beamforming weights

$$\boldsymbol{w} = \boldsymbol{R}_{\boldsymbol{YY}}^{-\mathbf{1}}\mathbf{a}(\mu_{iMax}, \upsilon_{jMax})$$

the coordinates $\mu_{iMax}, \upsilon_{jMax}$

$$\mu_{iMax} = \mu_{init} + i_{Max} \cdot \mu_{step}$$

$$\upsilon_{iMax} = \upsilon_{init} + j_{Max} \cdot \upsilon_{step}$$

and the spherical coordinates:

Elevation: $\theta_{est} = \sin^{-1}(\mu_{iMax})$

Azimuth: $\varphi_{est} = \sin^{-1}(\upsilon_{jMax}/\cos(\theta_{est}))$

6.1.2.2.2 Ceil-mount

In the ceil-mount scenario in processing step 2, after the CFAR detection, the refined azimuth elevation estimation is done by generating the zoomed-in heatmap around the detected point. Initially, as shown in Figure 23, if the checking for the detected peak being a local maximum on the coarse grid is enabled, (the filed localMaxCheck > 0 in CLI command dynamic2DAngleCfg), the function will compare the detected peak to its neighbors, and if it is not a local peak the inclusion of the point will be skipped. Otherwise, a zoomed-in heatmap using the Capon beamforming approach is generated. This is done by function RADARDEMO_aoaEst2DCaponBF_aeEstZoomin(). Since the covariance matrix is already calculated in the first processing step the function calculates the 2D zoomed-in heatmap around detected point. For detected point at $(i_{CoarseMax}, j_{CoarseMax})$, the 2D spectrum is calculated as

$$P_b\left(\mathbf{u}(i,j)\right) = \frac{1}{\mathbf{u}^{\mathbf{H}}(i,j)\boldsymbol{R}_{YY}^{-1}\mathbf{u}(i,j)},\ i,j = 0,\dots,N_{Ezoom_{in}} - 1$$

using the steering vectors $\mathbf{u}(i,j)$ calculated on the fly as described in Section 6.1.1.1.2.2. After finding the peak position in the zoomed-in heatmap, the function calculates the beamforming weights

$$\boldsymbol{w} = \boldsymbol{R}_{YY}^{-1}\mathbf{u}(i_{Max}, j_{Max})$$

then the coordinates $\mu_{iMax}, \upsilon_{jMax}$

$$\mu_{iMax} = \mu_{init} + \mu'_{init} + i_{Max} \cdot \mu'_{step}$$

$$\upsilon_{iMax} = \upsilon_{init} + \upsilon'_{init} + j_{Max} \cdot \upsilon'_{step}$$

and the spherical coordinates:

Elevation: $\theta_{est} = \sin^{-1}(\mu_{iMax})$

Azimuth: $\varphi_{est} = \sin^{-1}\left(\upsilon_{jMax}/\cos(\theta_{est})\right)$

If the peak expansion is enabled, (the CLI command filed <peakExpSamples> > 0), the neighbor points are included in the list of detected points if the following criteria are satisfied:

- P_{max}/P_{noise} is greater than SNR threshold, and
- The neighbor point is greater than threshold $T = P_{max} \cdot (T_{relative} - Shapness)$

Where

SNR threshold – is specified in the CLI command, the field <peakExpSNRThre>,

P_{noise} – is the noise estimated in the first pass of the CFAR detection,

$Sharpness = \frac{P_{max} - P_{min}}{P_{max} + P_{min}}$ – is the peak sharpness in the zoomed-in heatmap,

P_{max}, P_{min} – are the maximum and minimum values respectively in the zoomed-in heatmap,

$T_{relative}$ – is relative threshold specified in the CLI command, the field <peakExpRelThre>.

The above criteria for the peak expansion are heuristic with the intention to provide more points around the detected peak when the energy of the peak is higher and the detected peak shape is sharper in the observed zoomed-in area.

Based on the CLI configuration field <peakExpSamples>, the candidate neighboring points for the peak expansion are shown as red dots in the example in Figure 28 for the peak shown as a blue dot.

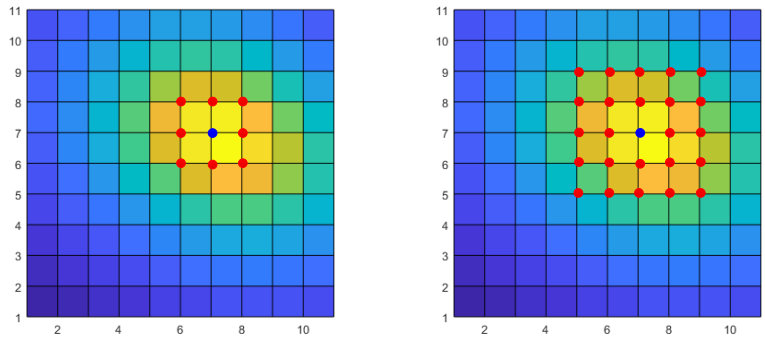


Figure 28 – Peak expansion candidates (red dots) in zoomed-in heatmap, for <peakExpSamples>=1 (left), and for <peakExpSamples>=2 (right).

6.1.2.2.3 Radial Velocity Estimation

The radial velocity estimation is performed on the detected points and neighboring points included through the peak expansion procedure. Before computing Doppler FFT the Capon beamforming is applied to all 12 antenna symbols at detected range. This is done by function RADARDEMO_aoaEst2DCaponBF_dopperEstInput(). The function calculates input to FFT as

$$y(c) = \mathbf{w}^H \mathbf{X}_c, c = 0, \dots, N_c - 1$$

Where N_c is number of chirps in a frame. The FFT is calculated using DSP library function DSPF_sp_fftSPxSP(). Note that before FFT calculation the real and imaginary are swapped since the library FFT function requires complex samples stored in order real first then imaginary, as opposed to the rest of the chain. The maximum peak is searched applied on the FFT output to find the radial velocity of detected point.

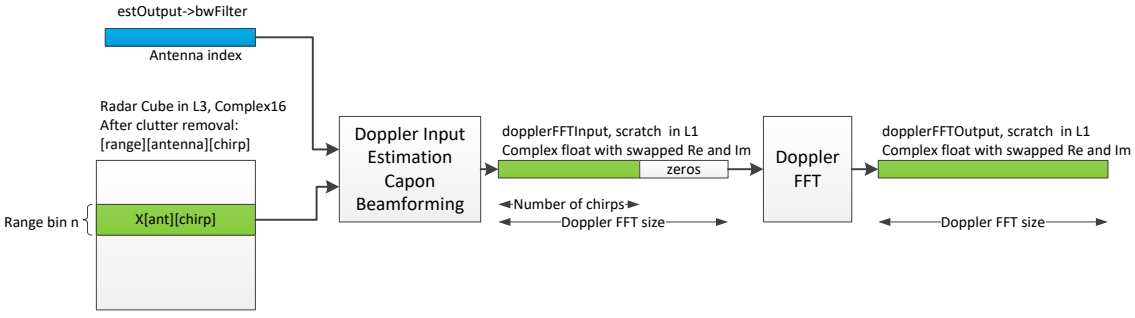


Figure 29 – Doppler Input estimation and Doppler FFT

6.1.3 RADARDEMO_aoaEst2DCaponBF_static_run()

This function is used for the static scene processing. It calculates 3D range-azimuth-elevation heatmap using a Bartlett beamforming approach. The 2D azimuth-elevation heatmap is generated at lower or equal resolution compared to dynamic scene processing. In wall-mount scenario the grid is decimated in both directions. Figure 30 illustrates decimated grid, for decimation factor equal to 8 in both directions. In ceil-mount scenario the azimuth-angle steering vector grid is same as in dynamic scene processing. This function is called per range bin. It is called at two places in the static scene processing chain, before and after the CFAR detection. It is refereed in the code as processing step 1 and processing step 2.

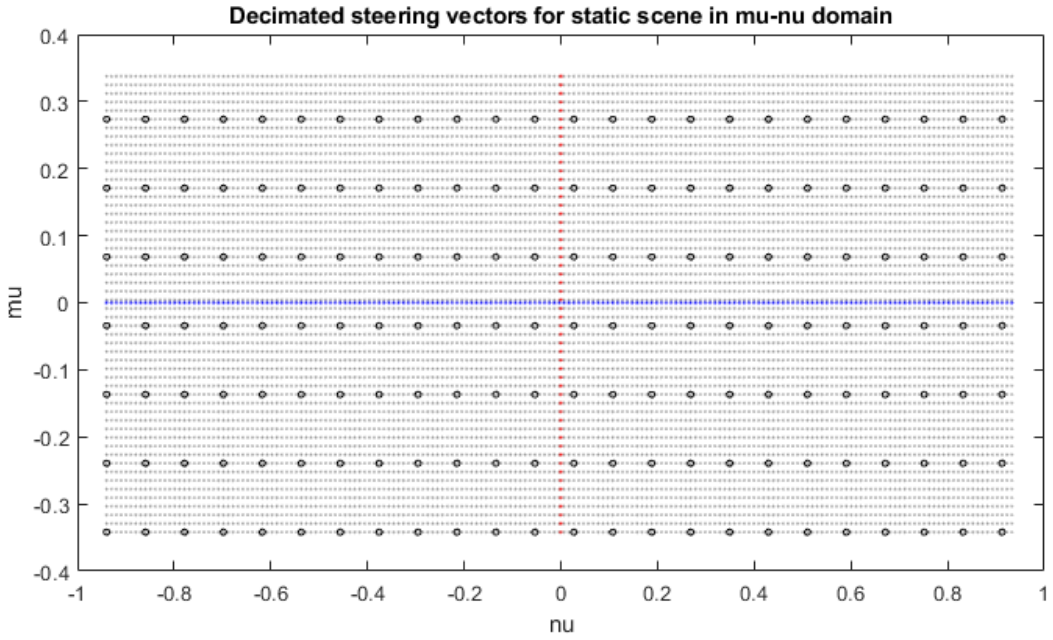


Figure 30 – Decimated steering vector grid (for method 1) with decimated factor set to 8 in both dimensions.

The function flow diagram is shown in Figure 31.

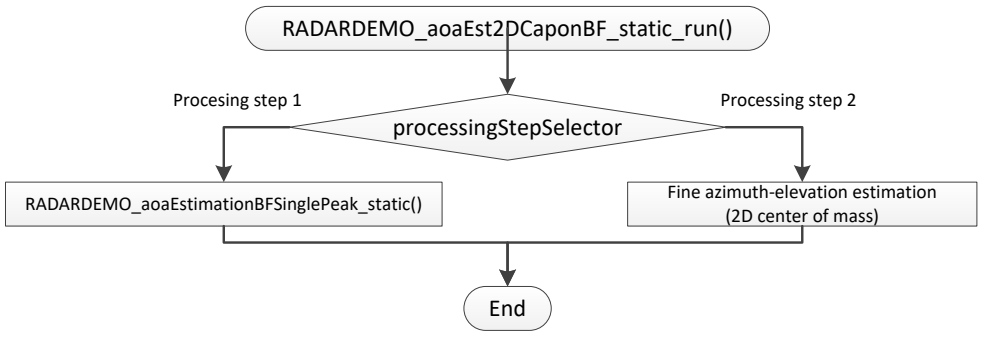


Figure 31 – Static scene processing flow diagram

6.1.3.1 Processing step 1

In step 1 the function is called for all range bins to construct the range-azimuth-elevation heatmap. The angle spectrum is calculated using Bartlett Beamforming approach. The input data are the averaged antennas across all chirps, per antenna, previously generated during the clutter removal processing:

$$\bar{\mathbf{Y}}_{\boldsymbol{r}} = \left[\bar{y}_0, \dots, \bar{y}_{N_R-1}\right]^{\boldsymbol{T}}, r = 0, \dots, N_{range\_fft} - 1$$

The spectrum is calculated as

$$P_r(\mu_{i2}, \upsilon_{i1}) = |\mathbf{a}^H(\mu_{i2}, \upsilon_{i1})\bar{\mathbf{Y}}_{\boldsymbol{r}}|^2, r = 0, \dots, N_{range\_fft} - 1, i_2 = 0, \dots, N'_E - 1, i_1 = 0, \dots, N'_A - 1$$

Where

$$N'_E = \left\lceil N_E / M_{elev\_step\_decim\_factor} \right\rceil, N'_A = \left\lceil N_A / M_{azim\_step\_decim\_factor} \right\rceil,$$

and the steering vectors calculated on the fly using pre-calculated azimuth and elevation steering vectors as

$$\mathbf{a}(\mu_{i2}, \upsilon_{i1}) = \mathbf{a}(\mu_{i2}) \circ \mathbf{a}(\upsilon_{i1})$$

The symbol ∘ denotes element-wise product. The final data arrangement in the heatmap, after all the range bins have been processed, is [elevation index][azimuth index][range index].

6.1.3.2 Processing step 2

The goal of this step is to increase the accuracy of the detected points in the static scene. Among numerous ways of doing the interpolation, the approach using a two-dimensional center of mass is implemented. The function is called per detected point. The function refines the azimuth-elevation estimation by calculating the center of mass of the detected point and its neighbors on the coarse heatmap $P(i, j)$ as

$$i_{COM} = \frac{\sum_{i=i-1}^{i+1}\sum_{j=j-1}^{j+1} i \cdot P(i,j)}{\sum_{i=i-1}^{i+1}\sum_{j=j-1}^{j+1} P(i,j)}$$

$$j_{COM} = \frac{\sum_{i=i-1}^{i+1}\sum_{j=j-1}^{j+1} j \cdot P(i,j)}{\sum_{i=i-1}^{i+1}\sum_{j=j-1}^{j+1} P(i,j)}$$

Then the coordinates $\mu_{iMax}, \upsilon_{jMax}$ are calculated

$$\mu_{iMax} = \mu_{init} + i_{COM} \cdot \mu_{step} \cdot M_{elev\_step\_decim\_factor}$$

$$\upsilon_{iMax} = \upsilon_{init} + j_{COM} \cdot \upsilon_{step} \cdot M_{elev\_step\_decim\_factor}$$

and the spherical coordinates

Elevation: $\theta_{est} = \sin^{-1}(\mu_{iMax})$

Azimuth: $\varphi_{est} = \sin^{-1}\left(\upsilon_{jMax}/\cos(\theta_{est})\right)$

This is illustrated in Figure 32.

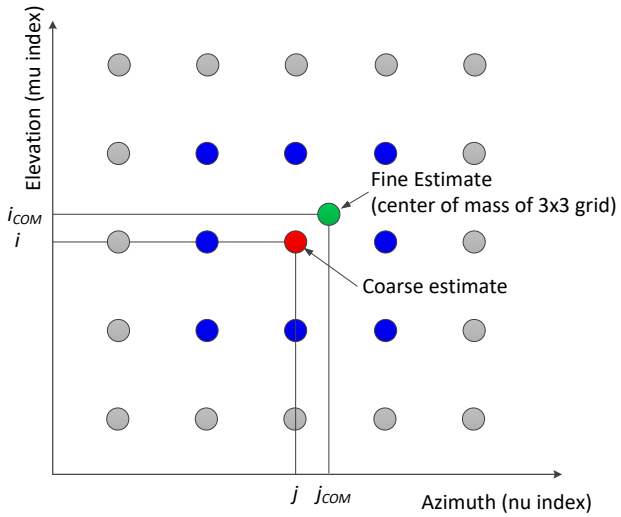


Figure 32 – Fine angle estimation as a center of mass on a coarse azimuth-elevation heatmap.

6.2 CFAR detection module

This module performs 2-pass CFAR detection algorithm in similar way in both scenarios wall-mount and ceil-mount. In wall mount scenario, the detection is applied on the 2D azimuth-range heatmap. In ceil-mount scenario the 3D elevation-azimuth-range heatmap is treated as a 2D heatmap with azimuth and elevation dimension reduced to one dimension (angleInd = azimInd + elevInd * azimDim). The CFAR detection module APIs are

- RADARDEMO_detectionCFAR_create()
- RADARDEMO_detectionCFAR_delete()
- RADARDEMO_detectionCFAR_run()

6.2.1 RADARDEMO_detectionCFAR_create()

This function initializes and configures the module. It allocates memory for its internal object and scratchpad memory for temporary detection list of the first pass CFAR detection. In the processing chain two instances are created, one for dynamic and the other for static scene processing. Note that internally the parameters for the second dimension are named using word Doppler although they are configured for the angle dimension.

6.2.2 RADARDEMO_detectionCFAR_run()

The run function calls different flavors of CFAR detection algorithms selected by CFAR type input parameter. In the 3D People counting demo the configuration parameter cfarType is hardcoded to RADARDEMO_DETECTIONCFAR_RA_CASOCFAR. In this mode the CFAR function

RADARDEMO_detectionCFAR_raCAAll() is used. Table 9 shows the hardcoded configuration parameters used in 3D People counting demo.

Parameter	Dynamic scene	Static scene
CFAR type	CFAR-CASO	CFAR-CASO
2-PASS CFAR	Enabled	Disabled
Neighbor Check	Enabled	Enabled

Table 9 – CFAR hardcoded configuration parameters

The function flow diagram is shown in Figure 34. For each angle index, the function performs CFAR along the range bins, (along the rows in the example shown in Figure 33), and stores detections in the temporary list. Further processing depends on the second pass flag:

Second pass is enabled: for each detected point from the temporary list, the CFAR across angle bins is applied. If the point is again detected, it is placed in the final list. If not, it checks for two conditions: the peak is greater than its neighbors in angle direction, and the peak is greater than the sidelobe threshold. If the conditions are satisfied, the point is added to the final list.

Second pass is disabled: for each detected point from the temporary list, two conditions are checked: the peak is greater than its neighbors in both directions range and angle, and the peak is greater than the sidelobe threshold. If the conditions are satisfied, the point is added to the final list.

This processing is repeated for all angle bins.

The angle CFAR is calculating noise in cyclic mode: at the angle edges the noise window wraps around the angle dimension.

The output list contains the following information per detected point

- range index,
- angle index, (note: in the function referred to as Doppler index) ,
- noise, used in later stage for fine SNR estimation (ceil-mount),
- snr estimate, final estimate placed in the output detection list (wall-mount)

The output list is shared dynamic and static CFAR instances. Currently the size of the list is hardcoded, (defined by MAX_DYNAMIC_CFAR_PNTS) and set to 150. The list is currently allocated according to MAX_DYNAMIC_CFAR_PNTS. Note that the maximum number of detected points for static scene MAX_STATIC_CFAR_PNTS, currently set also to 150, should not be set more than the size of the list.

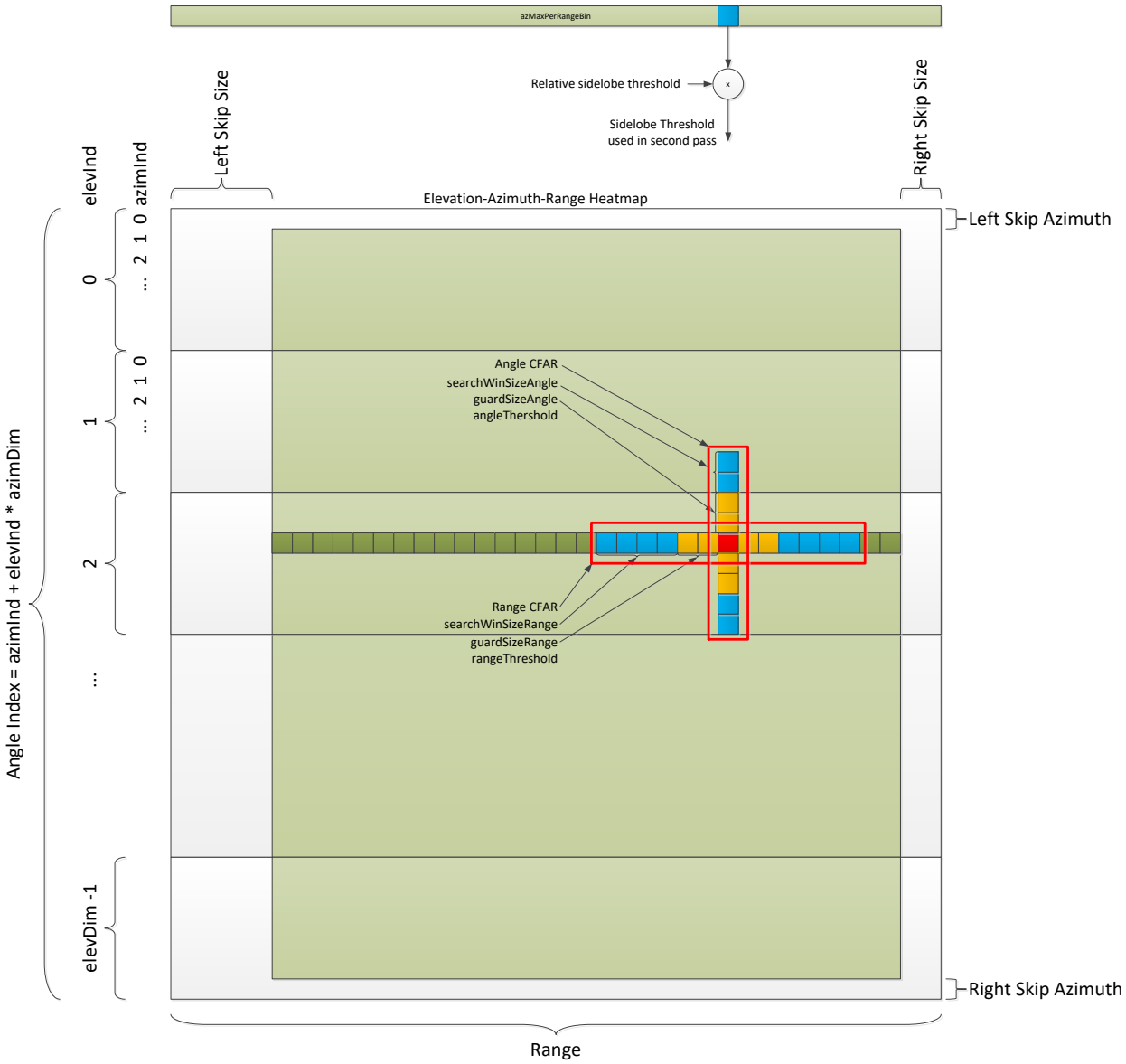


Figure 33 – Example of CFAR input range-angle heatmap in ceil-mount scenario

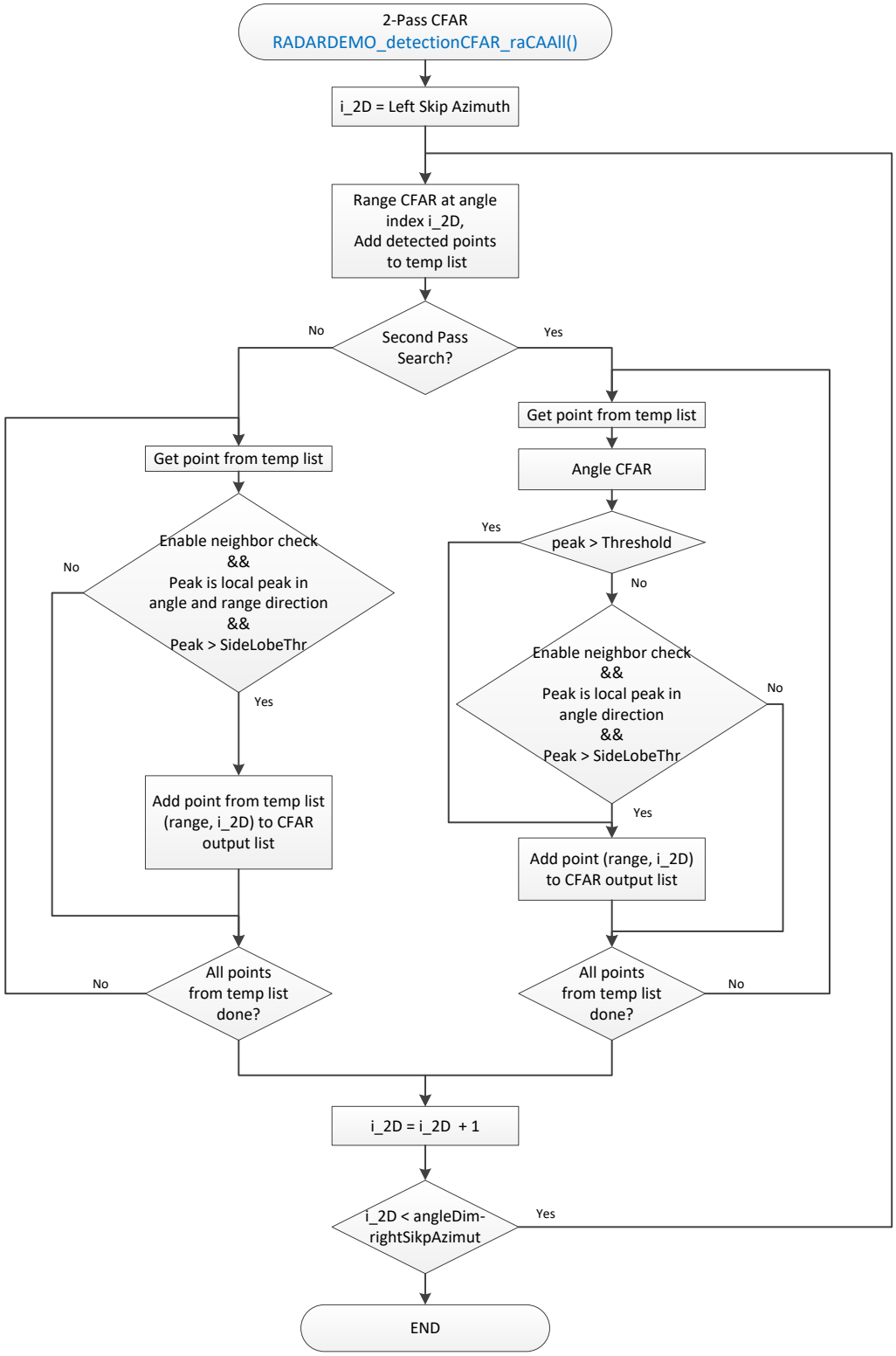


Figure 34 – 2Pass CFAR-CASO flow diagram

7 Memory usage

7.1 Memory Allocation

In the demo code the memory allocation is achieved in a few ways:

- Using memory allocation from the system heap in both domains, (using osal function MemoryP_ctrlAlloc()). The heap sizes can be tuned in pcount32_mss.cfg (112KB) and pcount32_dss.cfg (11KB). This form of memory allocation is mostly used on MSS by the Tracker module.
- Using simple OSAL memory allocation functions on DSS and MSS.

7.1.1 OSAL memory management functions

The OSAL memory management functions on DSS are used to allocate memory from three different memories L1, L2 and L3. The functions are located in radarOsal_malloc.c The three basic APIs are radarOsal_memInit(), radarOsal_memAlloc(), and radarOsal_memFree(). The size and the location of the heaps are defined in dss_main.c. Table 10 shows the heap sizes for three building options. The initialization is done as part of DSS DPM/DPC initialization.

Heap	Wall-mount	Ceil-mount
L1	0x2E00 (11.5KB)	0x1600 (5.5KB)
L1 SCRATCH	0x1200 (4.5KB)	0x2A00 (10.5KB)
L2	0x1A000 (104KB)	0x1FF00 (127.75KB)
L2 SCRATCH	0x900 (2.25KB)	0x900 (2.25KB)
L3	0x21000 (132KB)	0x13000 (76KB)

Table 10 – DSP Memory heap and scratch memory

Similar approach for memory allocation is used on MSS with the following APIs

DPC_ObjDetRangeHwa_MemPoolAlloc() and DPC_ObjDetRangeHwa_MemPoolReset().

7.1.2 Allocating Memory for Radar Cube Matrix

Radar Cube Memory is shared between MSS and DSS domains and it is located in L3 memory. The size of the radar cube memory is determined based on the CLI parameter configuration. On the MSS side, the Radar Cube memory is allocated in the L3 memory heap in DPC pre-start configuration function using the OSAL memory allocation function. The address is then passed to DPC on DSS domain. The MSS L3 memory heap size should match the expected radar cube size since the Radar Cube is the only object allocated in it. The L3 memory heap size is defined in the c674x_linker.cmd by defining a variable MMWAVE_MSSUSED_L3RAM_SIZE with the value set based on the build option as shown in Table 11.

	Wall-mount	Ceil-mount
MMWAVE_MSSUSED_L3RAM_SIZE	576K	672K

(Radar Cube Size)		

Table 11 – Radar Cube size defined by MMWAVE_MSSUSED_L3RAM_SIZE

7.2 DSS Memory usage

The DSS memory usage is summarized in Table 12. Note that the DSS has 32KB L1P memory which is used as part code RAM and part cache. Similarly, the 32KB DSS L1D memory is used as part data RAM and part data cache as defined in the table below.

Memory	Size	Used	Left	Description
L1P RAM	28K	full		Fast signal processing code
L1P cache	4K	full		Program cache
L1D RAM	16K	full		Memory Heap, scratch
L1D cache	16K	full		Data cache
L2S RAM	256K		9.4K	Code: 102.7K, Processing chain data: 106.25K Framework and BIOS buffers and data structure (heap, stack etc): 37.65K
L3 RAM	768K	719.5K (wall-mount) 759.5K (ceil-mount)	48.5K (wall-mount) 8.5K (ceil-mount)	Data: 708KB: radar cube 576K, L3 Heap 132K (wall-mount) 748KB: radar cube 672K, L3 Heap 76K (ceil-mount) Slow system code: 11.4K Overlaid code including configuration code, one time init code (EDMA driver etc), unused algorithm code: ~40K
HS RAM	32K	13.125K		Very slow non runtime system code

Table 12 - DSS Memory usage, (wall-mount)

The DSS code is loaded in two memories L2 and L3 RAM as shown in Table 13. The four sections, .fastCode, .overlaidCode, .hsramCode, and .unUsedCode are loaded in L3 at the location overlapped with L3 data heap. The code in these sections is either executed or copied to different locations before this memory is used as a heap. The sections .fastCode and .hsramCode are copied at the init-time (using EDMA) to L1P and HSRAM respectively since the bootloader can load the code only to L2/L3 memory. The difference in .text size between wall-mount and ceil-mount is because in the wall mount scenario the function RADARDEMO_aoaEst2DCaponBF_aeEstElevOnly() is used for Capon based elevation spectral estimation with fully unrolled loops to get better cycle performance. In the ceil-mount scenario, since this function is not used it is placed in .unUsedCode section.

The memory maps for wall-mount and ceil-mount scenarios are shown in Figure 35 and Figure 36 respectively.

Section	Size (wall-mount)	Size (ceil-mount)	Loaded to	Executed from	Overlapped section in Page 1
.text	102.6K	86.5K	L2	L2	Part of signal processing code, framework, bios and SDK
.fastCode	27.6K	27.6K	L3	L1P	Intense signal processing code executed in L1P RAM. Table 14 shows the functions placed to this section.
.overlaidCode	32.8K	30.5K	L3	L3 one-time	Initialization code, executed one time
.hsramCode	18.9K	18.9K	L3	HSRAM	Slow system non-real time code
.unUsedCode	7.5K	26K	L3	Not used	Linked but not used in demo
.slowCode	11.4K	11.4K	L3	L3	Non-real time critical code

Table 13 – DSS code sections in L2 and L3 memory

Function Name	Description
RADARDEMO_detectionCFAR_raCAAll()	CFAR
RADARDEMO_aoaEst2DCaponBF_raHeatmap()	Range-angle heatmap generation
RADARDEMO_aoaEst2DCaponBF_covInv()	Per range bin, estimate the covariance matrices from input 1D FFT results, and calculate the inverse of these matrices.
MATRIX_cholesky_flp_inv()	Calculate the inverse of these matrices.
RADARDEMO_aoaEst2DCaponBF_clutterRemoval()	Per range bin, removal static clutter from the input signal.
copyTranspose()	Store angle bins per range to Range-angle heatmap

Table 14 – Signal processing run-time critical code executed from L1P memory

The placement of the functions into particular code sections other than default .text section is specified in pcount3D_dss_linker.cmd. The TI linker for both R4F and C674x produce the map files that contain a

module summary of all the object files included in the application. Users can use this as a guide towards identifying components/source code that could be optimized.

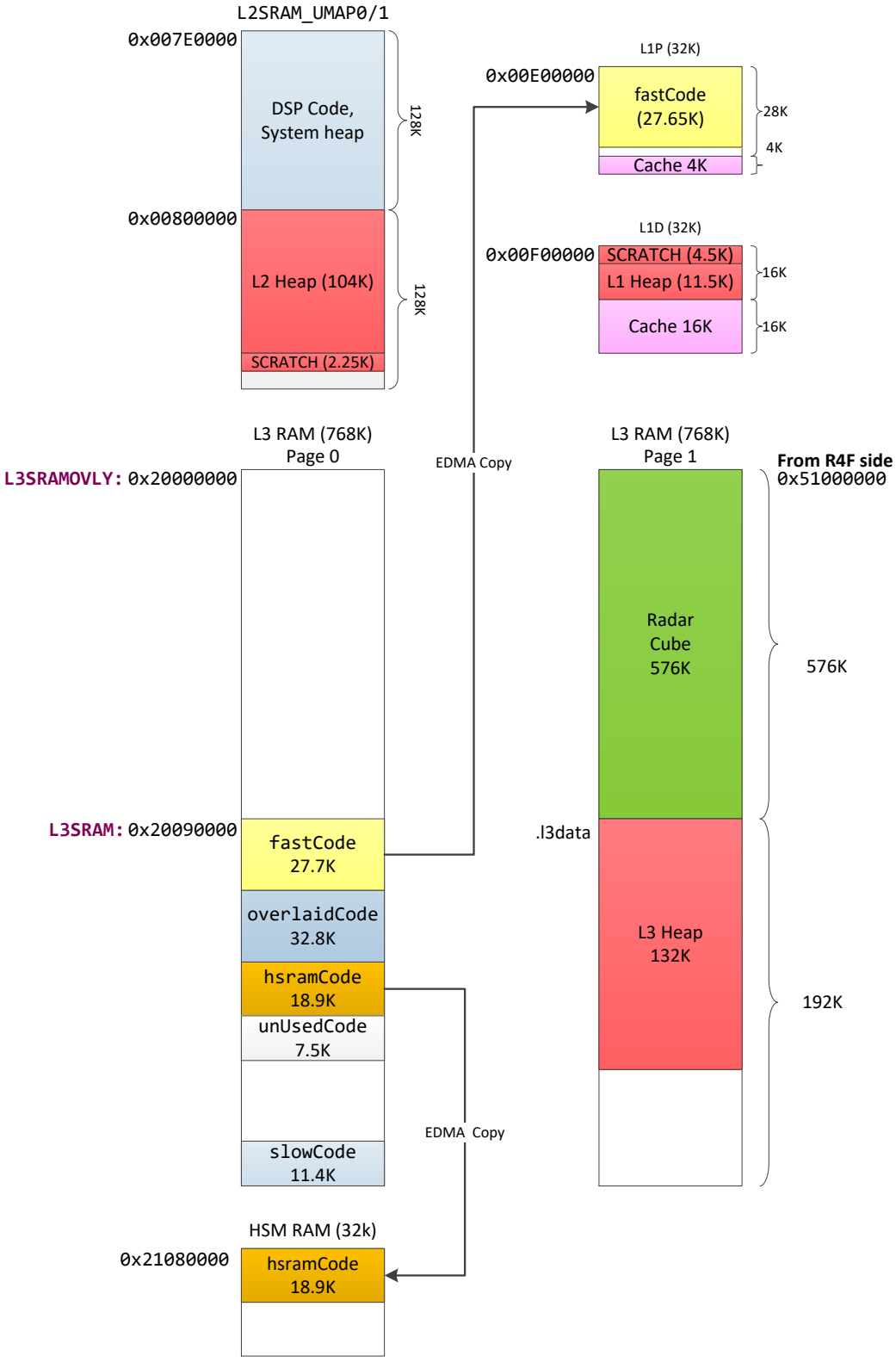


Figure 35 – DSS Memory usage – (wall-mount)

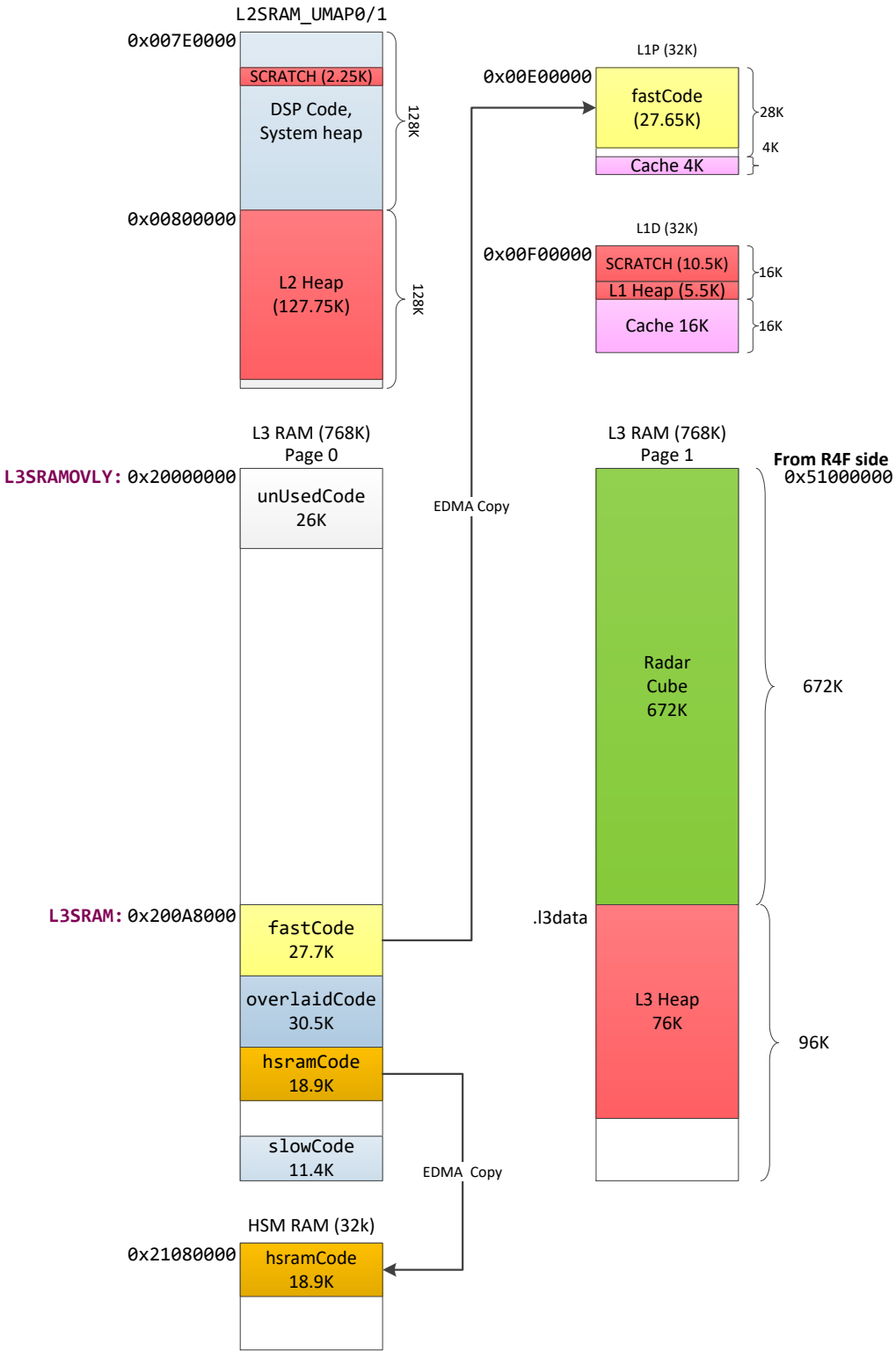


Figure 36 – DSS Memory usage – (ceil-mount)

7.3 DSS Memory Heap Allocation

Table 16, Table 17 and Table 18 illustrate memory heap allocations in L3, L2 and L1 respectively. Only two memory allocations are done by the object detection DPC running on DSS, (instance named "objDetObj->"), and all other allocations by the radar signal processing DPU, (instance named "inst->") in the tables). The allocated sizes correspond to the configurations shown in Table 15. Please note the convention for FOV specification: the left and right numbers are the angle search extents and the middle number is the angle search step. The names of the module instances and arrays are according to the names in the source code. The DPU instance structure is shown in Figure 11.

Configuration Parameters	Wall-mount	Ceil-mount
Range FFT Size	128	64
Number of Chirps per frame	3x96	3x224
Azimuth FOV[1]	-70° : 0.75° : 70°	-69° : 7° : 69°
Elevation FOV	-20° : 0.75° : 20°	-62° : 7° : 62°
Static scene Azimuth decimation factor	8	1
Static scene Elevation decimation factor	8	1
Ceil-mount zoom-in factor	N/A	5

Table 15 – Configuration parameters related to the memory allocation sizes shown in the following tables.

Wall-Mount	Size (Bytes)	Ceil-Mount	Size (Bytes)
objDetObj->executeResult	15036	objDetObj->executeResult	15036
objDetObj->stats	52	objDetObj->stats	52
inst->localHeatmap	95744	aoaOutput->static_information	6144
aoaOutput->static_information	12288	aoaOutput->invRnMatrices	39936
inst->benchmarkPtr	12	inst->detectionCFAROutput	36
benchmarkPtr->buffer	640	detectionCFAROutput->rangeInd	300
		detectionCFAROutput->dopplerInd	300
		detectionCFAROutput->snrEst	600
		detectionCFAROutput->noise	600
		inst->benchmarkPtr	12
		benchmarkPtr->buffer	640
Total	123772	Total	63656

Table 16 – L3 Memory Heap Allocations

[1] Note the convention for FOV specification: the left and right numbers are the angle search extents and the middle number is the approximate angle search step.

Wall-Mount	Size (Bytes)	Ceil-Mount	Size (Bytes)
raHeatMap_handle->steeringVecAzim	17952	raHeatMap_handle->steeringVec	27648
raHeatMap_handle->steeringVecElev	5184	inst->localHeatmap	73728
aoaOutput->invRnMatrices	79872	inst->dynamicCFARInstance	64
inst->dynamicCFARInstance	64	inst->dynamicHeatmapPtr	1152
inst->dynamicHeatmapPtr	748	inst->tempHeatMapOut	1152
inst->staticCFARInstance	64		
inst->tempHeatMapOut	676		
Total	104560	Total	103744

Table 17 – L2 Memory Heap Allocation

Wall-Mount	Size (Bytes)	Ceil-Mount	Size (Bytes)
radarProcessInstance_t (inst)	140	radarProcessInstance_t (inst)	140
inst->perRangeBinMax	516	inst->perRangeBinMax	260
inst->aoaInstance	52	inst->aoaInstance	52
raHeatMap_handle	56	raHeatMap_handle	56
raHeatMap_handle->virtAntInd2Proc	8	raHeatMap_handle->virtAntInd2Proc	12
aeEstimation_handle	88	aeEstimation_handle	88
aeEstimation_handle->virtAntInd2Proc	12	aeEstimation_handle->virtAntInd2Proc	12
aeEstimation_handle->procRngBinMask	16	aeEstimation_handle->steeringVecAzimInit	100
handle->dopTwiddle	1024	aeEstimation_handle->steeringVecAzimStep	96
inst->aoaInput	24	aeEstimation_handle->steeringVecElevInit	96
inst->aoaOutput	44	aeEstimation_handle->steeringVecElevStep	96
aoaOutput->azimEst	4	handle->dopTwiddle	2048
aoaOutput->elevEst	4	inst->aoaInput	24
aoaOutput->peakPow	4	inst->aoaOutput	44
aoaOutput->bwFilter	96	aoaOutput->azimEst	36
aoaOutput->dopplerIdx	2	aoaOutput->elevEst	36
inst->detectionCFAROutput	36	aoaOutput->peakPow	36
detectionCFAROutput->rangeInd	300	aoaOutput->bwFilter	864
detectionCFAROutput->dopplerInd	300	aoaOutput->dopplerIdx	18
detectionCFAROutput->snrEst	600	inst->detectionCFARInput	16
detectionCFAROutput->noise	600		
inst->detectionCFARInput	16		
inst->staticHeatmapPtr	748		
Total	4690	Total	4130

Table 18 – L1 Memory Heap Allocations

7.4 MSS Memory Usage

The MSS memory usage is shown in Table 19.

Memory	Size	Used	Left
L2P	512K	147K	365K
L2D	192K	162K	30K

Table 19 – MSS Memory Usage

8 Benchmarks

The 3D People Counting Demo measures on the fly the processing time of the detection layer, the tracker layer, and the data transfer time to the Host. These three measurements are sent per frame, as a part of the frame header (see Section 1.1 for packet header structure). These three measurements are denoted in Figure 10 as

- activeFrameProcTimeInUsec - DSP low level signal processing time excluding chirping time,
- trackingProcessingTimeInUsec - Group tracker processing time on R4F, and
- uartSendingTimeInUsec - DMA transfer time to Host via UART.

Also, within the low level signal processing chain, the additional measurements are tracked and kept internally within the structure radarProcessBenchmarkObj. The measurements include three main signal processing steps:

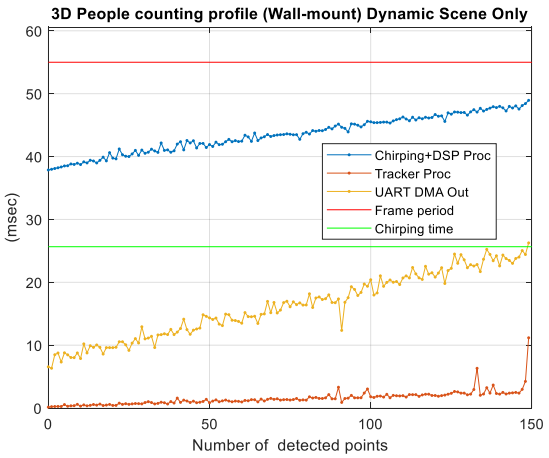
1. Capon BF Step 1 - Includes clutter removal, covariance estimation and inverse, and range-angle heatmap generation,
2. CFAR – 2Pass CFAR detection, and
3. Capon BF Step 2 – Includes, per detected point, elevation estimation (wall-mount)/fine angle estimation (ceil-mount) and Doppler estimation.

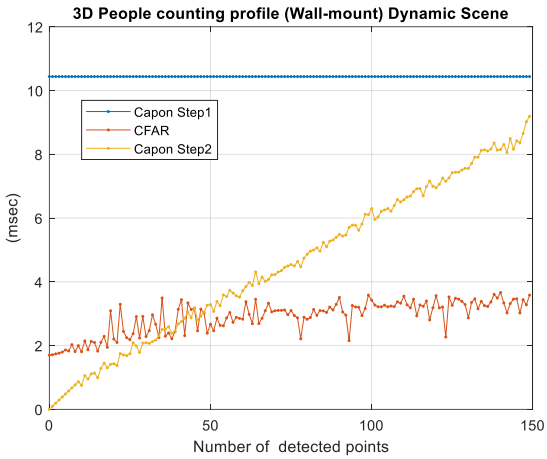
These measurements are collected and presented in the following two sections. The measurement procedure is explained in section .

8.1 Benchmarks – (Wall-mount)

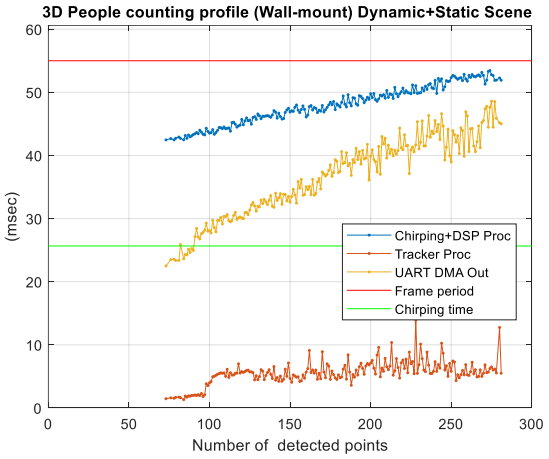
The measurements results are collected for the following profile configuration, with and without static scene processing: Frame period = 55 msec, numAzimBins = 187 (-70^o : 0.75^o : 70^o), numElevBins = 54 (-20^o : 0.75^o : 20^o), Chirping time = 3*96*(30+59.1) usec = 25.7 msec.

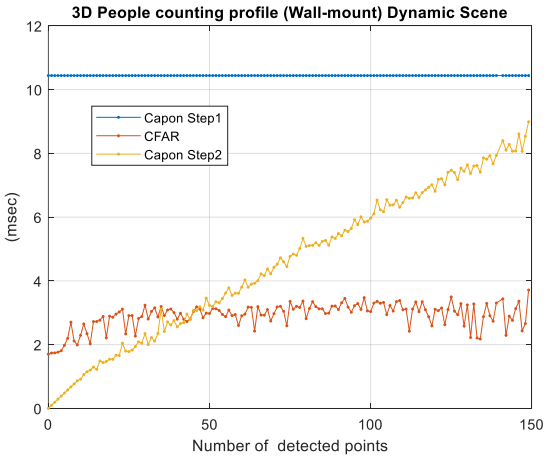
8.1.1 Dynamic scene only

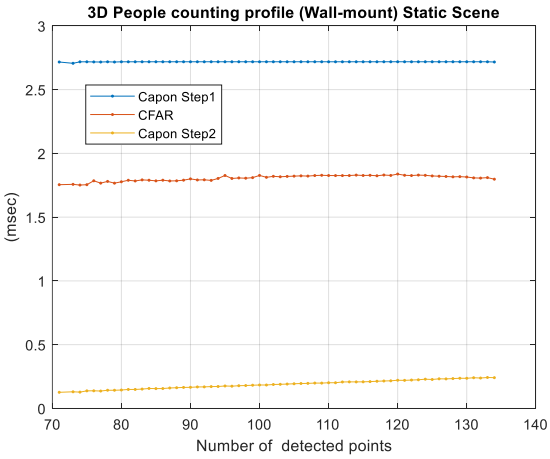




8.1.2 Dynamic and static scene



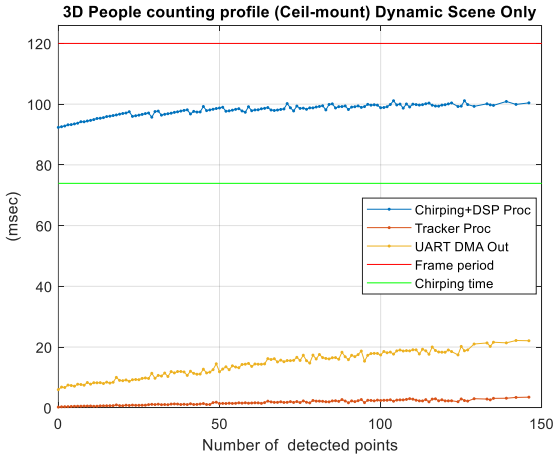


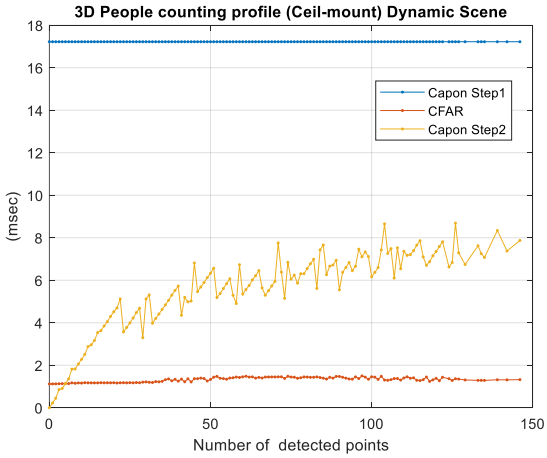


8.2 Benchmarks – (Ceil-mount)

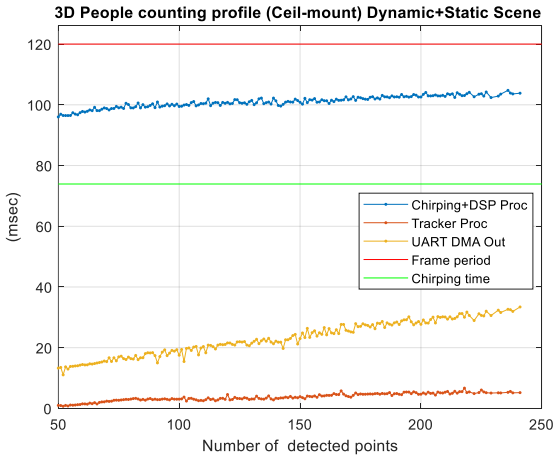
The measurements results are collected for the following profile configuration, with and without static scene processing: Frame period = 120 msec, numAngleBins = 18*16 = 288 (numAzimbins = 18 (-69° : 7° : 69°), numElevBins = 16 (-62° : 7° : 62°)), Chirping time = 3*224*(60+50) usec = 73.9 msec.

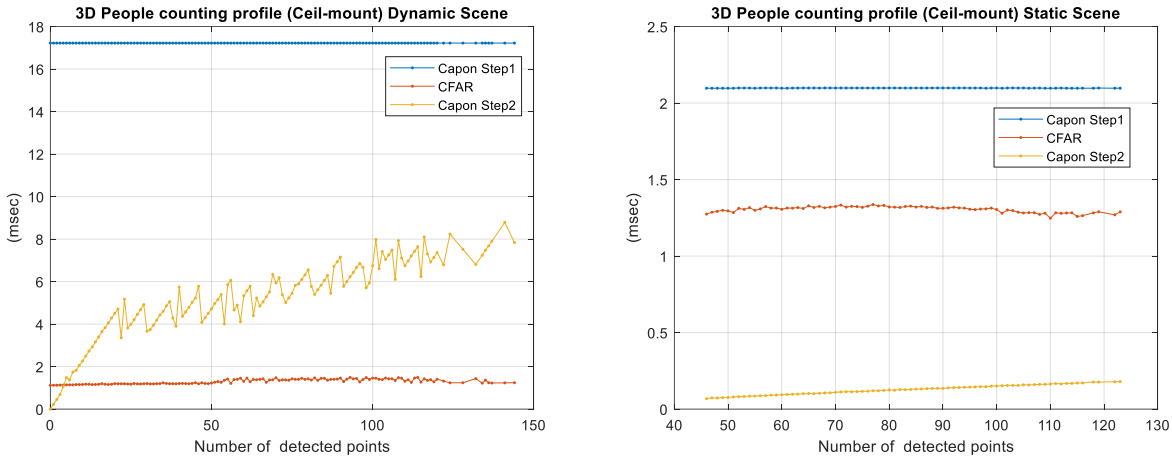
8.2.1 Dynamic scene only:





8.2.2 Dynamic and static scene





8.2.3 Profiling Procedure –DSP, tracker and UART transfer time

The measurements for the DSP and the tracker processing time and the UART transfer time are collected by running the demo code for several minutes with the random scene with few persons and objects moving in front of the sensor producing variable number of detected points and targets while recording the processing time of different components as a function of the number of detected points. The following code is added in the **mss_main.c** in the function (UART task) **MmwDemo_uartTxTask()**, inside the **while(1)** block at the very end of it (after the UART frame data transfer has been completed).

```
void MmwDemo_uartTxTask(UArg arg0, UArg arg1)
{

...

 while(1)
 {

  ...

  if(gMmwMssMCB.numDetectedPoints < (DOA_OUTPUT_MAXPOINTS+1))
  {
   if(gDbgInterFrameTime[gMmwMssMCB.numDetectedPoints] < gMmwMssMCB.frameStatsFromDSP->interFrameExecTimeInUsec)
     gDbgInterFrameTime[gMmwMssMCB.numDetectedPoints] = gMmwMssMCB.frameStatsFromDSP->interFrameExecTimeInUsec;

   if(gDbgTrackerTime[gMmwMssMCB.numDetectedPoints] <  gMmwMssMCB.trackerProcessingTimeInUsec)
     gDbgTrackerTime[gMmwMssMCB.numDetectedPoints] =  gMmwMssMCB.trackerProcessingTimeInUsec;

   if (gDbgUartTime[gMmwMssMCB.numDetectedPoints] < gMmwMssMCB.uartProcessingTimeInUsec)
     gDbgUartTime[gMmwMssMCB.numDetectedPoints] = gMmwMssMCB.uartProcessingTimeInUsec;
  }
 }
}
```

The three recording buffers are declared as

```
volatile uint32_t gDbgInterFrameTime[DOA_OUTPUT_MAXPOINTS+1];
volatile uint32_t gDbgTrackerTime[DOA_OUTPUT_MAXPOINTS+1];
volatile uint32_t gDbgUartTime[DOA_OUTPUT_MAXPOINTS+1];
```

The demo code is loaded and executed on the target using CCS (target configured in CCS development mode). The recording buffers were visually observed in CCS using the Graph tool until almost all bins had been populated. After the code was stopped, the buffers were extracted using the memory save option.

Note that as the tracker task is the lowest priority task on MSS the measured processing time includes any preemption by higher priority tasks or interrupts. Also, note that the tracker processing time, collected as a function of detected points and not as function of detected targets, shows mainly that the tracker task is the least intensive processing task, and that it is not as time critical as the task on DSS running the low level signal processing chain.

8.2.4 Profiling Procedure – three signal processing steps on DSP

The profiling procedure for the three main signal processing steps on the DSP, Capon BF Step 1, CFAR, and Capon BF Step 2, is similar as described in the previous section. The following code has been added inside the function Pcount3DDemo_handleObjectDetResult() (running in the context of main MSS task) just before sending the DPM_ioctl() notification to the DSS that the results are handled.

```
    radarProcessBenchmarkElem *benchmarkOut;
    benchmarkOut = (radarProcessBenchmarkElem *) outputFromDSP->benchmarkOut;
    benchmarkOut = (radarProcessBenchmarkElem *)SOC_translateAddress((uint32_t)benchmarkOut,
                                                        SOC_TranslateAddr_Dir_FROM_OTHER_CPU,
                                                        &retVal);
    if (gDynHeatmpGenCycles[benchmarkOut->dynNumDetPnts] < benchmarkOut->dynHeatmpGenCycles)
        gDynHeatmpGenCycles[benchmarkOut->dynNumDetPnts] = benchmarkOut->dynHeatmpGenCycles;

    if (gDynCfarDetectionCycles[benchmarkOut->dynNumDetPnts] < benchmarkOut->dynCfarDetectionCycles)
        gDynCfarDetectionCycles[benchmarkOut->dynNumDetPnts] = benchmarkOut->dynCfarDetectionCycles;

    if (gDynAngleDopEstCycles[benchmarkOut->dynNumDetPnts] < benchmarkOut->dynAngleDopEstCycles)
        gDynAngleDopEstCycles[benchmarkOut->dynNumDetPnts] = benchmarkOut->dynAngleDopEstCycles;


    if (gStaticHeatmpGenCycles[benchmarkOut->staticNumDetPnts] < benchmarkOut->staticHeatmpGenCycles)
        gStaticHeatmpGenCycles[benchmarkOut->staticNumDetPnts] = benchmarkOut->staticHeatmpGenCycles;

    if (gStaticCfarDetectionCycles[benchmarkOut->staticNumDetPnts] < benchmarkOut->staticCfarDetectionCycles)
        gStaticCfarDetectionCycles[benchmarkOut->staticNumDetPnts] = benchmarkOut->staticCfarDetectionCycles;

    if (gStaticAngleEstCycles[benchmarkOut->staticNumDetPnts] < benchmarkOut->staticAngleEstCycles)
        gStaticAngleEstCycles[benchmarkOut->staticNumDetPnts] = benchmarkOut->staticAngleEstCycles;
```

The recording buffers are declared as

```
int gDynHeatmpGenCycles[DOA_OUTPUT_MAXPOINTS + 1];
int gDynCfarDetectionCycles[DOA_OUTPUT_MAXPOINTS + 1];
int gDynAngleDopEstCycles[DOA_OUTPUT_MAXPOINTS + 1];

int gStaticHeatmpGenCycles[DOA_OUTPUT_MAXPOINTS + 1];
int gStaticCfarDetectionCycles[DOA_OUTPUT_MAXPOINTS + 1];
int gStaticAngleEstCycles[DOA_OUTPUT_MAXPOINTS + 1];
```

9 UART and Output to the Host

The demo outputs one packet of data every frame. The packet contains point cloud data of the current frame, and the group tracking data of the previous frame.

The system design is to have a maximum of 150 dynamic CFAR detections, and 150 static CFAR detections, total of 300 points for wall-mount scenario, and total of 750 points for ceil-mount scenario after angle estimation and Doppler estimation.

Data are sent out to Host via UART in compressed mode. Point cloud data are sent as array of detected points (8 Bytes per point). The tracker data include the Target list (108-byte per 3D target) and the Target tag (1-byte per detected point).

The UART is configured at 921600 bps in DMA mode. DMA mode allows other tasks to be executed in parallel while the data are transmitted to the Host. In the worst case sending maximum number of points and tracked targets would take: (8*750+108*20+750)*10/(921600) = 96.7ms.

1.1 Output TLV Description

The packet structure consists of fixed sized frame header, followed by variable number of TLVs (see Figure 37). Each TLV has fixed header followed by variable size payload. The Byte order is Little Endian.

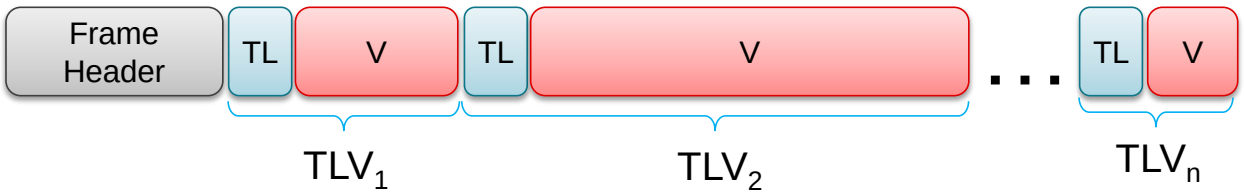


Figure 37 – Data packet structure sent to Host

1.1.1 Frame Header Structure

The frame header is of fixed size (52bytes). The structure field definition is shown in MATLAB syntax below.

```
frameHeaderStructType = struct(...
    'sync',              {'uint64', 8}, ... % Sync Pattern
    'version',           {'uint32', 4}, ... % mmWaveSDK version
    'packetLength',      {'uint32', 4}, ... % In bytes, including header
    'platform',          {'uint32', 4}, ... % 0xA1642 or 0xA1443
    'frameNumber',       {'uint32', 4}, ... % Starting from 1
    'subframeNumber',    {'uint32', 4}, ...
    'chirpMargin',       {'uint32', 4}, ... % Chirp Processing margin, in us
    'frameProcTimeInUsec', {'uint32', 4}, ... % Frame Processing time, in us
    'trackProcessTime', {'uint32', 4}, ... % Tracking Processing time, in us
    'uartSentTime' ,     {'uint32', 4}, ... % Time spent to send data, in us
    'numTLVs' ,          {'uint16', 2}, ... % Number of TLVs in thins frame
    'checksum',          {'uint16', 2});    % Header checksum
```

The field `version` is constructed as

```
MMWAVE_SDK_VERSION_BUILD | (MMWAVE_SDK_VERSION_BUGFIX << 8) |
(MMWAVE_SDK_VERSION_MINOR << 16) | (MMWAVE_SDK_VERSION_MAJOR << 24)
```

The `syncPattern` is constructed as

```
typecast(uint16([hex2dec('0102'),hex2dec('0304'),hex2dec('0506'),hex2dec('0708')]),'uint64');
```

1.1.2 TLV structure

The TLV structure consists of

Fixed Header (8bytes) followed by TLV specific payload. The TLV header structure is shown in MATLAB syntax bellow.

```
tlvHeaderStruct = struct(...
   'type',          {'uint32', 4}, ... % TLV object Type
   'length',        {'uint32', 4});    % TLV object Length, in bytes, including TLV header
```

1.1.3 Point Cloud TLV

Type = POINTCLOUD_3D

Length = sizeof (tlvHeaderStruct) + sizeof (pointCloudUnitStruct) + sizeof (pointStruct) x numberOfPoints

Point cloud unit structure is defined as:

```
pointCloudUnitStruct = struct(...
    'elevationUnit',    {'float', 4}, ... % unit resolution of elevation report, in rad
    'azimuthUnit',      {'float', 4}, ... % unit resolution of azimuth report, in rad
    'dopplerUint',      {'float', 4}, ... % unit resolution of Doppler report, in m/s
    'rangeUint',        {'float', 4}, ... % unit resolution of Range report, in m
    'snrUint',          {'float', 4});    % unit resolution of SNR report, ratio
```

Each point (pointStruct) is defined as:

```
pointStruct = struct(...
    'elevation',        {'int8', 1}, ...   % Elevation report, in number of elevationUnit
    'azimuth',          {'int8', 1}, ...   % Azimuth report, in number of azimuthUnit
    'doppler',          {'int16', 1}, ...  % Doppler, in number of dopplerUint
    'range',            {'uint16', 2}, ... % Range, in number of rangeUint
    'snr',              {'uint16', 2});    % SNR, in number of snrUint
```

1.1.4 Target List TLV

Type = TARGET_LIST_3D

Length = sizeof (tlvHeaderStruct) + sizeof (targetStruct) x numberOfTargets

Each target is defined as:

```
targetStruct2D = struct(...
    'tid',  {'uint32', 4}, ...  % Track ID
    'posX', {'float', 4}, ...   % Target position in X dimension, m
    'posY', {'float', 4}, ...   % Target position in Y dimension, m
    'velX', {'float', 4}, ...   % Target velocity in X dimension, m/s
    'velY', {'float', 4}, ...   % Target velocity in Y dimension, m/s
    'accX', {'float', 4}, ...   % Target acceleration in X dimension, m/s2
    'accY', {'float', 4}, ...   % Target acceleration in Y dimension, m/s
    'posZ', {'float', 4}, ...   % Target position in Z dimension, m
    'velZ', {'float', 4}, ...   % Target velocity in Z dimension, m/s
    'accZ', {'float', 4}, ...   % Target acceleration in Z dimension, m/s
    'EC',   {'float', 16*4}, ...% Tracking error covariance matrix, [4x4] in
                                % range/azimuth/elevation/doppler coordinates
    'G',    {'float', 4});      % Gating function gain
```

1.1.5 Target Index TLV

Type = TARGET_INDEX

Length = sizeof (tlvHeaderStruct) + numberOfPoints

Payload is a byte array, where each byte is a Target ID

```
targetIndex = struct(...
      'targetID',          {'uint8', 1}); % Track ID
```