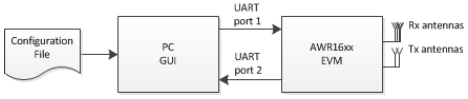
Millimeter Wave (mmw) Demo for XWR16XX

Introduction



The millimeter wave demo shows some of the capabilities of the XWR16xx SoC using the drivers in the mmWave SDK (Software Development Kit). It allows user to specify the chirping profile and displays the detected objects and other information in real-time.

Following is a high level description of the features of this demo:

- Be able to specify desired chirping profile through command line interface (CLI) on a UART port or through the TI Gallery App - **mmWave Demo Visualizer** - that allows user to provide a variety of profile configurations via the UART input port and displays the streamed detected output from another UART port in real-time, as seen in picture above.
- Some sample profile configurations have been provided in the demo directory that can be used with CLI directly or via **mmWave Demo Visualizer** under following directory:

  ```
  mmw/profiles
  ```

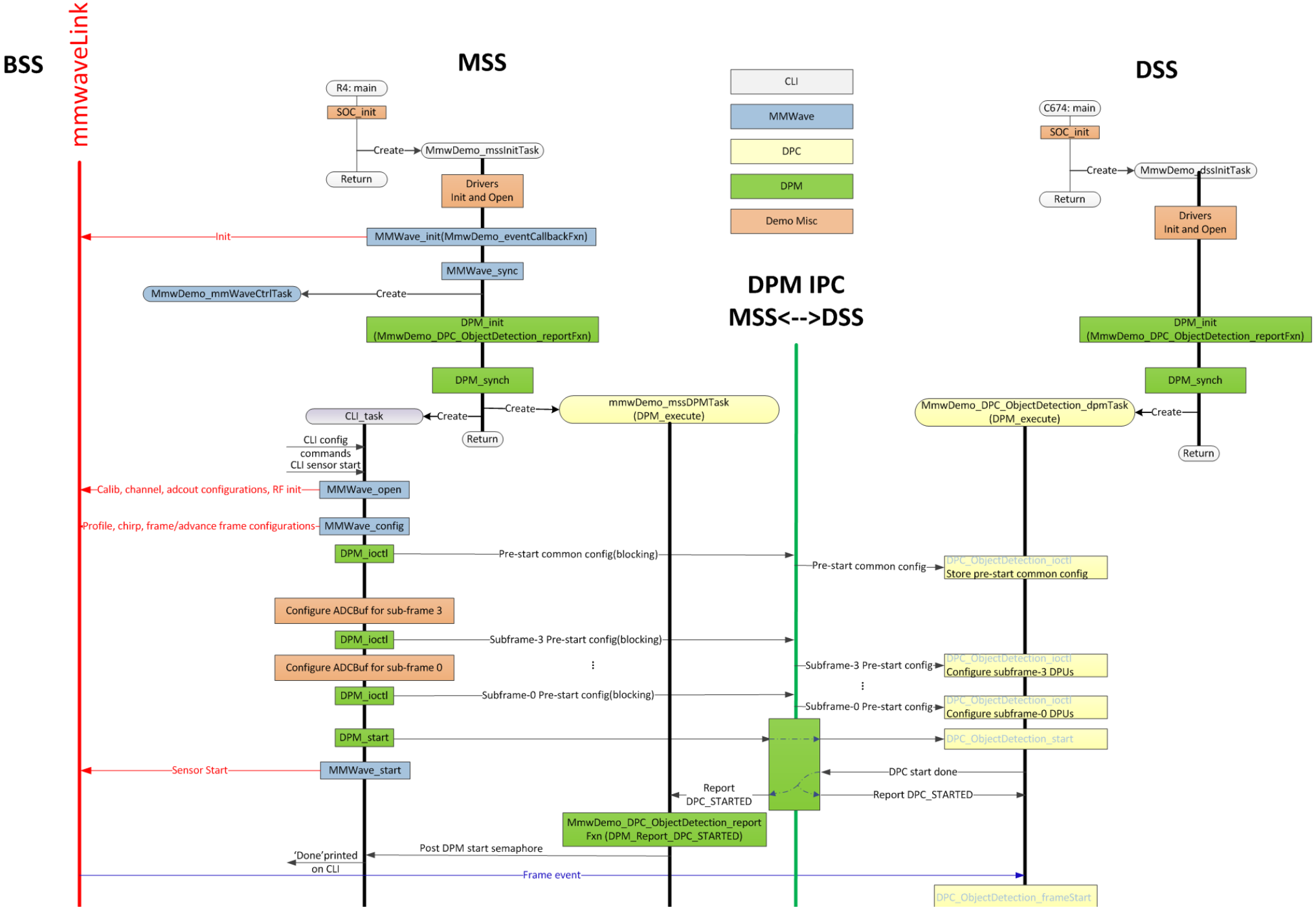
  .
- Do 1D, 2D, CFAR, and Azimuth processing and stream out velocity and two spatial coordinates (x,y) of the detected objects in real-time.
- Various display options besides object detection like azimuth heat map and Doppler-range heat map.

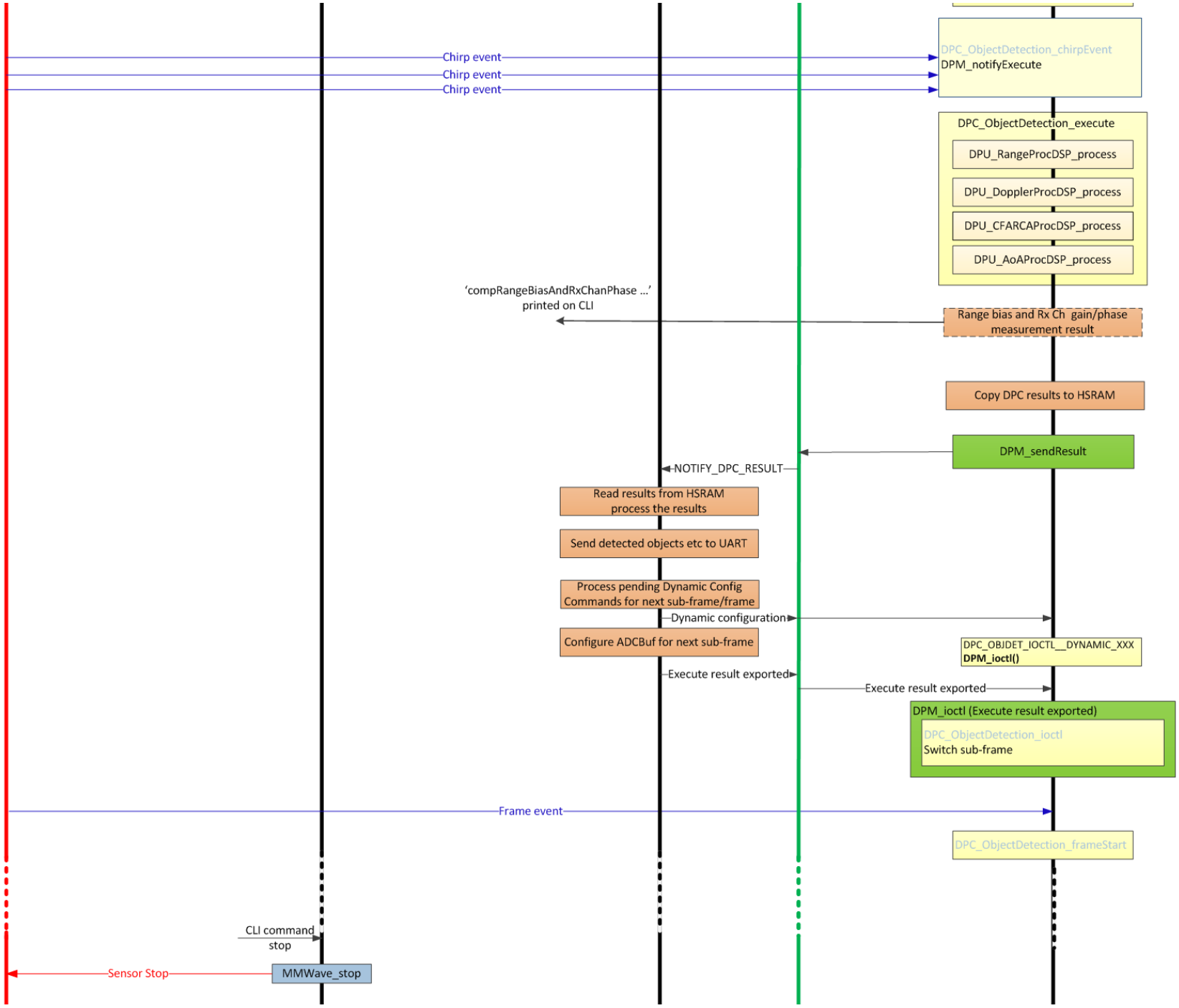
Limitations

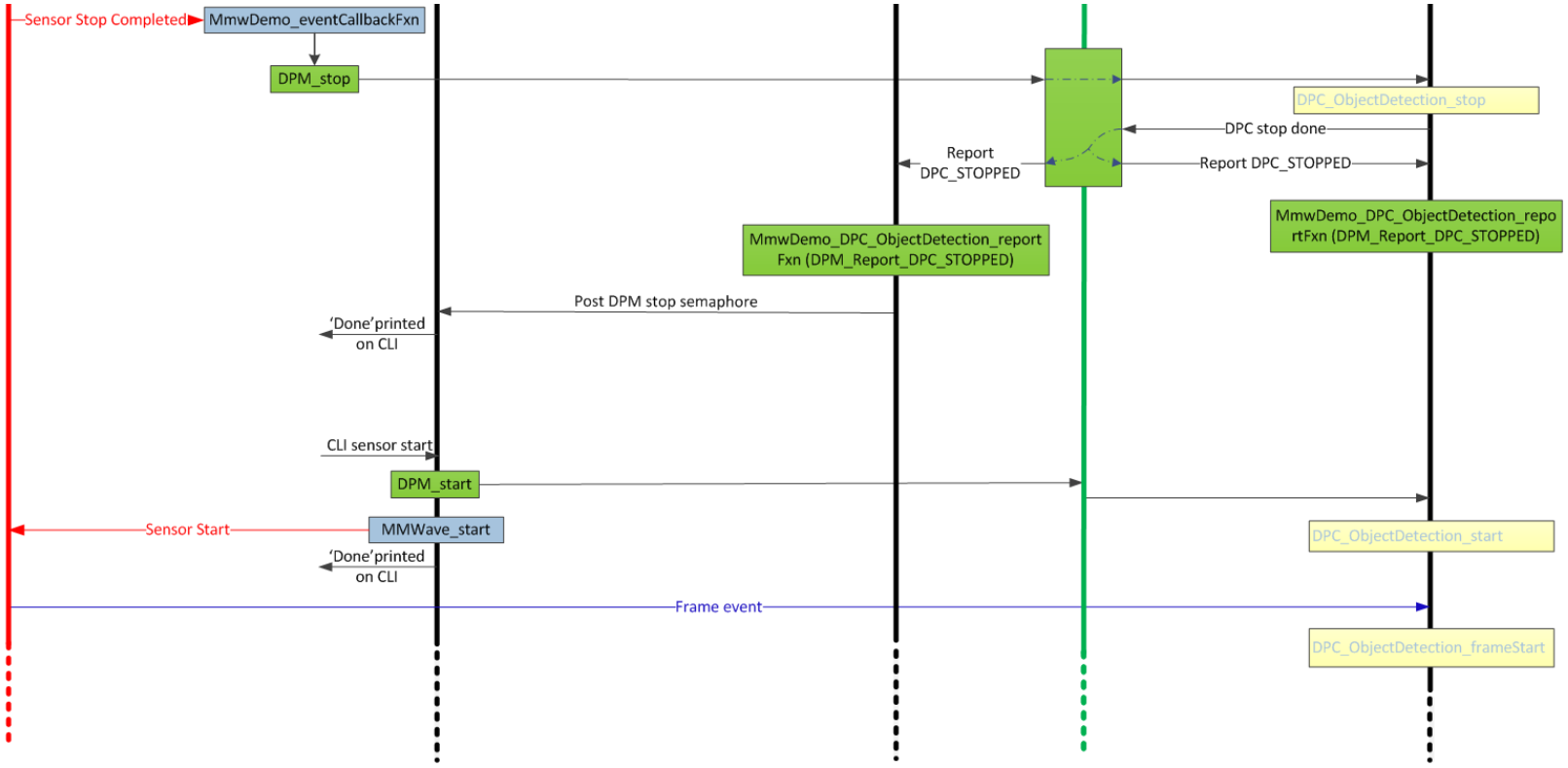
- Because of UART speed limit (< 1 Mbps), the frame time is more restrictive. For example, for the azimuth and Doppler heat maps for 256 FFT range and 16 point FFT Doppler, it takes about 200 ms to transmit.
- For most boards, a range bias of few centimeters has been observed. User can estimate the range bias on their board and correct using the calibration procedure described in **Range Bias and Rx Channel Gain/Phase Measurement and Compensation**.

System Execution Flow

The millimeter wave demo runs on R4F (MSS) and C674x (DSS). Following diagram shows the system execution flow


BSS
mmwaveLink
MSS
DSS
CLI
MMWave
DPC
DPM
Demo Misc
DPM IPC
MSS<-->DSS
R4: main
SOC_init
Create
MmwDemo_mssInitTask
Return
Drivers
Init and Open
Init
MMWave_init(MmwDemo_eventCallbackFxn)
MMWave_sync
MmwDemo_mmWaveCtrlTask
Create
DPM_init
(MmwDemo_DPC_ObjectDetection_reportFxn)
DPM_synch
Create
mmwDemo_mssDPMTask
(DPM_execute)
CLI_task
Create
Return
CLI config
commands
CLI sensor start
Calib, channel, adcout configurations, RF init
MMWave_open
Profile, chirp, frame/advance frame configurations
MMWave_config
DPM_ioctl
Pre-start common config(blocking)
Pre-start common config
DPC_ObjectDetection_ioctl
Store pre-start common config
Configure ADCBuf for sub-frame 3
DPM_ioctl
Subframe-3 Pre-start config(blocking)
Subframe-3 Pre-start config
DPC_ObjectDetection_ioctl
Configure subframe-3 DPUs
Configure ADCBuf for sub-frame 0
DPM_ioctl
Subframe-0 Pre-start config(blocking)
Subframe-0 Pre-start config
DPC_ObjectDetection_ioctl
Configure subframe-0 DPUs
DPM_start
DPC_ObjectDetection_start
Sensor Start
MMWave_start
DPC start done
Report
DPC_STARTED
Report DPC_STARTED
MmwDemo_DPC_ObjectDetection_report
Fxn (DPM_Report_DPC_STARTED)
Post DPM start semaphore
'Done'printed
on CLI
Frame event
DPC_ObjectDetection_frameStart
C674: main
SOC_init
Create
MmwDemo_dssInitTask
Return
Drivers
Init and Open
DPM_init
(MmwDemo_DPC_ObjectDetection_reportFxn)
DPM_synch
MmwDemo_DPC_ObjectDetection_dpmTask
(DPM_execute)
Create
Return


Chirp event
Chirp event
Chirp event
DPC_ObjectDetection_chirpEvent
DPM_notifyExecute
DPC_ObjectDetection_execute
DPU_RangeProcDSP_process
DPU_DopplerProcDSP_process
DPU_CFARCAProcDSP_process
DPU_AoAProcDSP_process
'compRangeBiasAndRxChanPhase ...'
printed on CLI
Range bias and Rx Ch gain/phase measurement result
Copy DPC results to HSRAM
DPM_sendResult
NOTIFY_DPC_RESULT
Read results from HSRAM process the results
Send detected objects etc to UART
Process pending Dynamic Config Commands for next sub-frame/frame
Dynamic configuration
Configure ADCBuf for next sub-frame
DPC_OBJDET_IOCTL__DYNAMIC_XXX
DPM_ioctl()
Execute result exported
Execute result exported
DPM_ioctl (Execute result exported)
DPC_ObjectDetection_ioctl
Switch sub-frame
Frame event
DPC_ObjectDetection_frameStart
CLI command
stop
Sensor Stop
MMWave_stop



System Execution Flow

Software Tasks

The demo consists of the following (SYSBIOS) tasks:

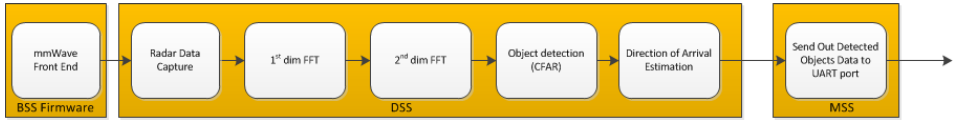
MSS

- **MmwDemo_initTask**. This task is created/launched by **main** and is a one-time active task whose main functionality is to initialize drivers (<driver>_init), MMWave module (MMWave_init), DPM module (DPM_init), open UART and data path related drivers (ADCBUF), and create/launch the following tasks (the CLI_task is launched indirectly by calling CLI_open).
- CLI_task. This command line interface task provides a simplified 'shell' interface which allows the configuration of the BSS via the mmWave interface (MMWave_config). It parses input CLI configuration commands like chirp profile and GUI configuration. When sensor start CLI command is parsed, all actions related to starting sensor and starting the processing the data path are taken. When sensor stop CLI command is parsed, all actions related to stopping the sensor and stopping the processing of the data path are taken
- **MmwDemo_mmWaveCtrlTask**. This task is used to provide an execution context for the mmWave control, it calls in an endless loop the MMWave_execute API.
- **mmwDemo_mssDPMTask**. This task is used to provide an execution context for DPM (Data Path Manager) execution, it calls in an endless loop. There is no DPC registered with DPM.

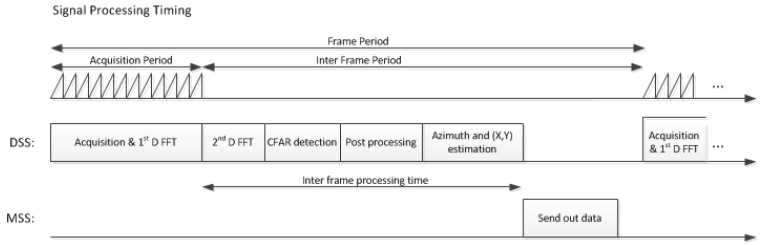
DSS

- **MmwDemo_initTask**. This task is created/launched by **main** and is a one-time active task whose main functionality is to initialize drivers (<driver>_init), DPM module (DPM_init), data path related drivers (EDMA), and create/launch the following tasks.
- **MmwDemo_DPC_ObjectDetection_dpmTask**. This task is used to provide an execution context for DPM (Data Path Manager) execution, it calls in an endless loop the DPM_execute API. In this context, all of the registered object detection DPC (Data Path Chain) APIs like configuration, control and execute will take place. In this task. When the DPC's execute API produces the detected objects and other results, they are reported to MSS where they are transmitted out of the UART port for display using the visualizer.

Data Path



Top Level Data Path Processing Chain



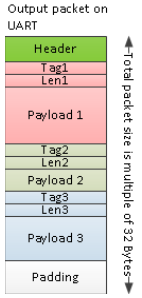
Top Level Data Path Timing

The data path processing consists of taking ADC samples as input and producing detected objects (point-cloud and other information) to be shipped out of UART port to the PC. The algorithm processing is realized using the DPM registered Object Detection DPC. The details of the processing in DPC can be seen from the following doxygen documentation:

```
ti/datapath/dpc/objectdetection/objdetdsp/docs/doxygen/html/index.html
```

Output information sent to host

Output packets with the detection information are sent out every frame through the UART. Each packet consists of the header **MmwDemo_output_message_header_t** and the number of TLV items containing various data information with types enumerated in **MmwDemo_output_message_type_e**. The numerical values of the types can be found in **mmw_output.h**. Each TLV item consists of type, length (**MmwDemo_output_message_tl_t**) and payload information. The structure of the output packet is illustrated in the following figure. Since the length of the packet depends on the number of detected objects it can vary from frame to frame. The end of the packet is padded so that the total packet length is always multiple of 32 Bytes.



Output packet structure sent to UART

The following subsections describe the structure of each TLV.

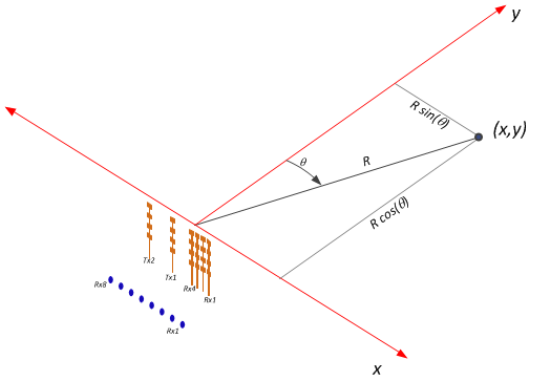
List of detected objects

Type: (**MMWDEMO_OUTPUT_MSG_DETECTED_POINTS**)

Length: (Number of detected objects) x (size of DPIF_PointCloudCartesian_t)

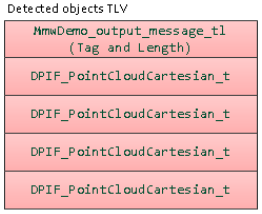
Value: Array of detected objects. The information of each detected object is as per the structure DPIF_PointCloudCartesian_t. When the number of detected objects is zero, this TLV item is not sent. The maximum number of objects that can be detected in a sub-frame/frame is DPC_OBJDET_MAX_NUM_OBJECTS.

The orientation of x,y and z axes relative to the sensor is as per the following figure.



Coordinate Geometry

The whole detected objects TLV structure is illustrated in figure below.



Detected objects TLV

Range profile

Type: (**MMWDEMO_OUTPUT_MSG_RANGE_PROFILE**)

Length: (Range FFT size) x (size of uint16_t)

Value: Array of profile points at 0th Doppler (stationary objects). The points represent the sum of log2 magnitudes of received antennas expressed in Q9 format.

Noise floor profile

Type: (**MMWDEMO_OUTPUT_MSG_NOISE_PROFILE**)

Length: (Range FFT size) x (size of uint16_t)

Value: This is the same format as range profile but the profile is at the maximum Doppler bin (maximum speed objects). In general for stationary scene, there would be no objects or clutter at maximum speed so the range profile at such speed represents the receiver noise floor.

Azimuth static heatmap

Type: (**MMWDEMO_OUTPUT_MSG_AZIMUT_STATIC_HEAT_MAP**)

Length: (Range FFT size) x (Number of virtual antennas) (size of **cmplx16ImRe_t_**)

Value: Array DPU_AoAProcDSP_HW_Resources::azimuthStaticHeatMap. The antenna data are complex symbols, with imaginary first and real second in the following order:

```
        Imag(ant 0, range 0), Real(ant 0, range 0),...,Imag(ant N-1, range 0),Real(ant N-1, range 0)
        ...
        Imag(ant 0, range R-1), Real(ant 0, range R-1),...,Imag(ant N-1, range R-1),Real(ant N-1, range R-1)
```

Based on this data the static azimuth heat map is constructed by the GUI running on the host.

Range/Doppler heatmap

Type: (**MMWDEMO_OUTPUT_MSG_RANGE_DOPPLER_HEAT_MAP**)

Length: (Range FFT size) x (Doppler FFT size) (size of uint16_t)

Value: Detection matrix DPIF_DetMatrix::data. The order is :

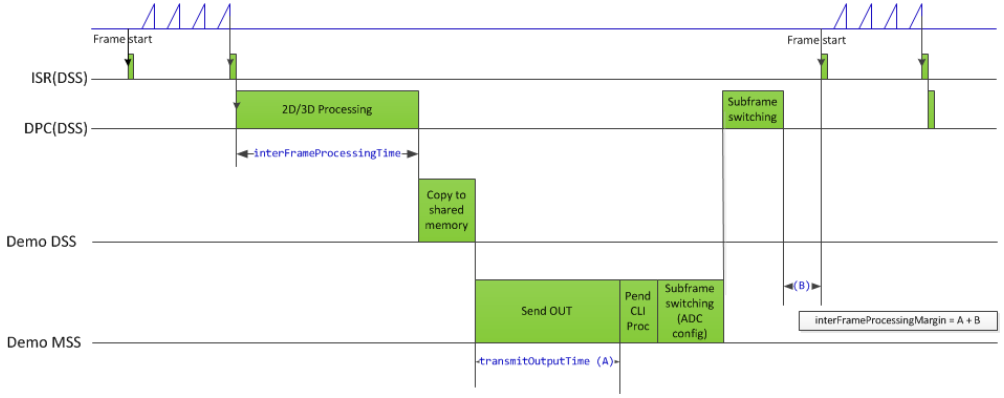
```
X(range bin 0, Doppler bin 0),...,X(range bin 0, Doppler bin D-1),
...
X(range bin R-1, Doppler bin 0),...,X(range bin R-1, Doppler bin D-1)
```

Stats information

Type: (**MMWDEMO_OUTPUT_MSG_STATS**)

Length: (size of **MmwDemo_output_message_stats_t**)

Value: Timing information as per **MmwDemo_output_message_stats_t**. See timing diagram below related to the stats.



Processing timing

Note:

1. While the **MmwDemo_output_message_stats_t::interFrameProcessingTime** reported will be of the current sub-frame/frame, the **MmwDemo_output_message_stats_t::interFrameProcessingMargin** and **MmwDemo_output_message_stats_t::transmitOutputTime** will be of the previous sub-frame (of the same **MmwDemo_output_message_header_t::subFrameNumber** as that of the current sub-frame) or of the previous frame.
2. The **MmwDemo_output_message_stats_t::interFrameProcessingMargin** excludes the UART transmission time (available as **MmwDemo_output_message_stats_t::transmitOutputTime**). This is done intentionally to inform the user of a genuine inter-frame processing margin without being influenced by a slow transport like UART, this transport time can be significantly longer for example when streaming out debug information like heat maps. Also, in a real product deployment, higher speed interfaces (e.g LVDS) are likely to be used instead of UART. User can calculate the margin that includes transport overhead (say to determine the max frame rate that a particular demo configuration will allow) using the stats because they also contain the UART transmission time.

The CLI command "guMonitor" specifies which TLV element will be sent out within the output packet. The arguments of the CLI command are stored in the structure **MmwDemo_GuiMonSel_t**.

Side information of detected objects

Type: (**MMWDEMO_OUTPUT_MSG_DETECTED_POINTS_SIDE_INFO**)

Length: (Number of detected objects) x (size of DPIF_PointCloudSideInfo_t)

Value: Array of detected objects side information. The side information of each detected object is as per the structure DPIF_PointCloudSideInfo_t). When the number of detected objects is zero, this TLV item is not sent.

Temperature Stats

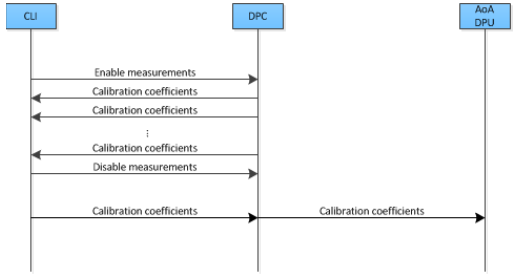
Type: (**MMWDEMO_OUTPUT_MSG_TEMPERATURE_STATS**)

Length: (size of **MmwDemo_temperatureStats_t**)

Value: Structure of detailed temperature report as obtained from Radar front end. **MmwDemo_temperatureStats_t::tempReportValid** is set to return value of rlRfGetTemperatureReport. If **MmwDemo_temperatureStats_t::tempReportValid** is 0, values in **MmwDemo_temperatureStats_t::temperatureReport** are valid else they should be ignored. This TLV is sent along with Stats TLV described in **Stats information**

Range Bias and Rx Channel Gain/Phase Measurement and Compensation

Because of imperfections in antenna layouts on the board, RF delays in SOC, etc, there is need to calibrate the sensor to compensate for bias in the range estimation and receive channel gain and phase imperfections. The following figure illustrates the calibration procedure.



Calibration procedure ladder diagram

The calibration procedure includes the following steps:

1. Set a strong target like corner reflector at the distance of X meter (X less than 50 cm is not recommended) at boresight.
2. Set the following command in the configuration profile in .../profiles/profile_calibration.cfg, to reflect the position X as follows:

   ```
   measureRangeBiasAndRxChanPhase 1 X D
   ```

   where D (in meters) is the distance of window around X where the peak will be searched. The purpose of the search window is to allow the test environment from not being overly constrained say because it may not be possible to clear it of all reflectors that may be stronger than the one used for calibration. The window size is recommended to be at least the distance equivalent of a few range bins. One range bin for the calibration profile (profile_calibration.cfg) is about 5 cm. The first argument "1" is to enable the measurement. The stated configuration profile (.cfg) must be used otherwise the calibration may not work as expected (this profile ensures all transmit and receive antennas are engaged among other things needed for calibration).
3. Start the sensor with the configuration file.
4. In the configuration file, the measurement is enabled because of which the DPC will be configured to perform the measurement and generate the measurement result (DPU_AoAProc_compRxChannelBiasCfg_t) in its result structure (DPC_ObjectDetection_ExecuteResult_t::compRxChanBiasMeasurement), the measurement results are written out on the CLI port (**MmwDemo_measurementResultOutput**) in the format below:

   ```
   compRangeBiasAndRxChanPhase <rangeBias> <Re(0,0)> <Im(0,0)> <Re(0,1)> <Im(0,1)> ... <Re(0,R-1)> <Im(0,R-1)> <Re(1,0)> <Im(1,0)> ... <Re(T-1,R-1)> <Im(T-1,R-1)>
   ```

   For details of how DPC performs the measurement, see the DPC documentation.
5. The command printed out on the CLI now can be copied and pasted in any configuration file for correction purposes. This configuration will be passed to the DPC for the purpose of applying compensation during angle computation, the details of this can be seen in the DPC documentation. If compensation is not desired, the following command should be given

   ```
   compRangeBiasAndRxChanPhase 0.0   1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
   ```

   Above sets the range bias to 0 and the phase coefficients to unity so that there is no correction. Note the two commands must always be given in any configuration file, typically the measure commmand will be disabled when the correction command is the desired one.

Streaming data over LVDS

The LVDS streaming feature enables the streaming of HW data (a combination of ADC/CP/CQ data) and/or user specific SW data through LVDS interface. The streaming is done mostly by the CBUFF and EDMA peripherals with minimal CPU intervention. The streaming is configured through the **MmwDemo_LvdsStreamCfg_t** CLI command which allows control of HSI header, enable/disable of HW and SW data and data format choice for the HW data. The choices for data formats for HW data are:

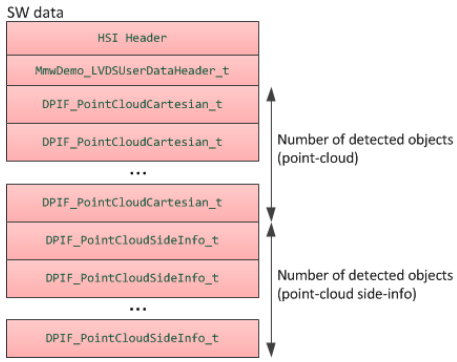
- **MMW_DEMO_LVDS_STREAM_CFG_DATAFMT_DISABLED**
- **MMW_DEMO_LVDS_STREAM_CFG_DATAFMT_ADC**
- **MMW_DEMO_LVDS_STREAM_CFG_DATAFMT_CP_ADC_CQ**

In order to see the high-level data format details corresponding to the above data format configurations, refer to the corresponding slides in ti\drivers\cbuff\docs\CBUFF_Transfers.pptx

When HW data LVDS streaming is enabled, the ADC/CP/CQ data is streamed per chirp on every chirp event. When SW data streaming is enabled, it is streamed during inter-frame period after the list of detected objects for that frame is computed. The SW data streamed every frame/sub-frame is composed of the following in time:

1. HSI header (**HSIHeader_t**): refer to HSI module for details.
2. User data header: **MmwDemo_LVDSUserDataHeader**
3. User data payloads:
   a. Point-cloud information as a list : DPIF_PointCloudCartesian_t x number of detected objects
   b. Point-cloud side information as a list : DPIF_PointCloudSideInfo_t x number of detected objects

The format of the SW data streamed is shown in the following figure:



LVDS SW Data format

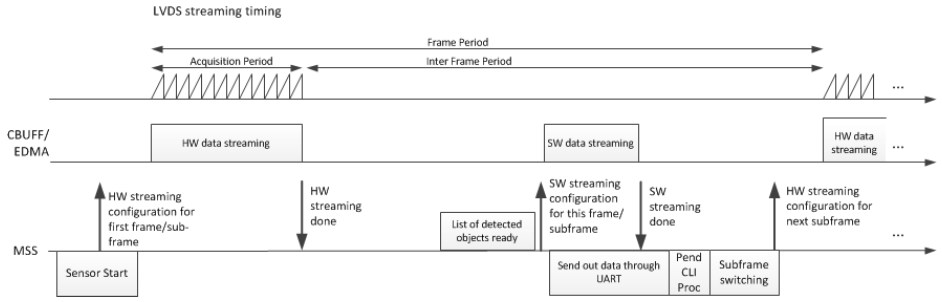
Note:

1. Only single-chirp formats are allowed, multi-chirp is not supported.
2. When number of objects detected in frame/sub-frame is 0, there is no transmission beyond the user data header.
3. For HW data, the inter-chirp duration should be sufficient to stream out the desired amount of data. For example, if the HW data-format is ADC and HSI header is enabled, then the total amount of data generated per chirp is:
   (numAdcSamples * numRxChannels * 4 (size of complex sample) + 52 [sizeof(HSIDataCardHeader_t) + sizeof(HSISDKHeader_t)]) rounded up to multiples of 256 [=sizeof(HSIHeader_t)] bytes.
   The chirp time Tc in us = idle time + ramp end time in the profile configuration. For n-lane LVDS with each lane at a maximum of B Mbps,
   maximum number of bytes that can be send per chirp = Tc * n * B / 8 which should be greater than the total amount of data generated per chirp i.e
   Tc * n * B / 8 >= round-up(numAdcSamples * numRxChannels * 4 + 52, 256).
   E.g if n = 2, B = 600 Mbps, idle time = 7 us, ramp end time = 44 us, numAdcSamples = 512, numRxChannels = 4, then 7650 >= 8448 is violated so this configuration will not work. If the idle-time is doubled in the above example, then we have 8700 > 8448, so this configuration will work.
4. For SW data, the number of bytes to transmit each sub-frame/frame is:
   52 [sizeof(HSIDataCardHeader_t) + sizeof(HSISDKHeader_t)] + sizeof(MmwDemo_LVDSUserDataHeader_t) [=8] +
   number of detected objects (Nd) * { sizeof(DPIF_PointCloudCartesian_t) [=16] + sizeof(DPIF_PointCloudSideInfo_t) [=4] } rounded up to multiples of 256 [=sizeof(HSIHeader_t)] bytes.
   or X = round-up(60 + Nd * 20, 256). So the time to transmit this data will be
   X * 8 / (n*B) us. The maximum number of objects (Ndmax) that can be detected is defined in the DPC (DPC_OBJDET_MAX_NUM_OBJECTS). So if Ndmax = 500, then time to transmit SW data is 68 us. Because we parallelize this transmission with the much slower UART transmission, and because UART transmission is also sending at least the same amount of information as the LVDS, the LVDS transmission time will not add any burdens on the processing budget beyond the overhead of reconfiguring and activating the CBUFF session (this overhead is likely bigger than the time to transmit).
5. The total amount of data to be transmitted in a HW or SW packet must be greater than the minimum required by CBUFF, which is 64 bytes or 32 CBUFF Units (this is the definition CBUFF_MIN_TRANSFER_SIZE_CBUFF_UNITS in the CBUFF driver implementation). If this threshold condition is violated, the CBUFF driver will return an error during configuration and the demo will generate a fatal exception as a result. When HSI header is enabled, the total transfer size is ensured to be at least 256 bytes, which satisfies the minimum. If HSI header is disabled, for the HW session, this means that numAdcSamples * numRxChannels * 4 >= 64. Although mmwavelink allows minimum number of ADC samples to be 2, the demo is supported for numAdcSamples >= 64. So HSI header is not required to be enabled for HW only case. But if SW session is enabled, without the HSI header, the bytes in each packet will be 8 + Nd * 20. So for frames/sub-frames where Nd < 3, the demo will generate exception. Therefore HSI header must be enabled if SW is enabled, this is checked in the CLI command validation.

Implementation Notes

1. The LVDS implementation is mostly present in **mmw_lvds_stream.h** and **mmw_lvds_stream.c** with calls in **mss_main.c**. Additionally HSI clock initialization is done at first time sensor start using **MmwDemo_mssSetHsiClk**.
2. EDMA channel resources for CBUFF/LVDS are in the global resource file (**mmw_res.h**, see **Hardware Resource Allocation**) along with other EDMA resource allocation. The user data header and two user payloads are configured as three user buffers in the CBUFF driver. Hence SW allocation for EDMA provides for three sets of EDMA resources as seen in the SW part (swSessionEDMAChannelTable[.]) of **MmwDemo_LVDSStream_EDMAInit**. The maximum number of HW EDMA resources are needed for the data-format **MMW_DEMO_LVDS_STREAM_CFG_DATAFMT_CP_ADC_CQ**, which as seen in the corresponding slide in ti\drivers\cbuff\docs\CBUFF_Transfers.pptx is 12 channels (+ shadows) including the 1st special CBUFF EDMA event channel which CBUFF IP generates to the EDMA, hence the HW part (hwSessionEDMAChannelTable[.]) of **MmwDemo_LVDSStream_EDMAInit** has 11 table entries.
3. Although the CBUFF driver is configured for two sessions (hw and sw), at any time only one can be active. So depending on the LVDS CLI configuration and whether advanced frame or not, there is logic to activate/deactivate HW and SW sessions as necessary.
4. The CBUFF session (HW/SW) configure-create and delete depends on whether or not re-configuration is required after the first time configuration.
   a. For HW session, re-configuration is done during sub-frame switching to re-configure for the next sub-frame but when there is no advanced frame (number of sub-frames = 1), the HW configuration does not need to change so HW session does not need to be re-created.
   b. For SW session, even though the user buffer start addresses and sizes of headers remains same, the number of detected objects which determines the sizes of some user buffers changes from one sub-frame/frame to another sub-frame/frame. Therefore SW session needs to be recreated every sub-frame/frame.
5. User may modify the application software to transmit different information than point-cloud in the SW data e.g radar cube data (output of range DPU). However the CBUFF also has a maximum link list entry size limit of 0x3FFF CBUFF units or 32766 bytes. This means it is the limit for each user buffer entry [there are maximum of 3 entries -1st used for user data header, 2nd for point-cloud and 3rd for point-cloud side information]. During session creation, if this limit is exceeded, the CBUFF will return an error (and demo will in turn generate an exception). A single physical buffer of say size 50000 bytes may be split across two user buffers by providing one user buffer with (address, size) = (start address, 25000) and 2nd user buffer with (address, size) = (start address + 25000, 25000), beyond this two (or three if user data header is also replaced) limit, the user will need to create and activate (and wait for completion) the SW session multiple times to accomplish the transmission.

The following figure shows a timing diagram for the LVDS streaming (the figure is not to scale as actual durations will vary based on configuration).



LVDS timing diagram

How to bypass CLI

Re-implement the file **mmw_cli.c** as follows:

1. **MmwDemo_CLIInit** should just create a task with input taskPriority. Lets say the task is called "MmwDemo_sensorConfig_task".
2. All other functions are not needed
3. Implement the MmwDemo_sensorConfig_task as follows:
   - Fill gMmwMssMCB.cfg.openCfg
   - Fill gMmwMssMCB.cfg.ctrlCfg
   - Add profiles and chirps using MMWave_addProfile and MMWave_addChirp functions
   - Call **MmwDemo_CfgUpdate** for every offset in **Offsets for storing CLI configuration** (MMWDEMO_xxx_OFFSET in **mmw_mss.h**)
   - Fill gMmwMssMCB.objDetCommonCfg.preStartCommonCfg
   - Call **MmwDemo_openSensor**
   - Call **MmwDemo_configSensor**
   - Call **MmwDemo_startSensor** (One can use helper function **MmwDemo_isAllCfgInPendingState** to know if all dynamic config was provided)

Hardware Resource Allocation

The Object Detection DPC needs to configure the DPUs hardware resources (EDMA). Even though the hardware resources currently are only required to be allocated for this one and only DPC in the system, the resource partitioning is shown to be in the ownership of the demo. This is to illustrate the general case of resource allocation across more than one DPCs and/or demo's own processing that is post-DPC processing. This partitioning can be seen in the **mmw_res.h** file. This file is passed as a compiler command line define

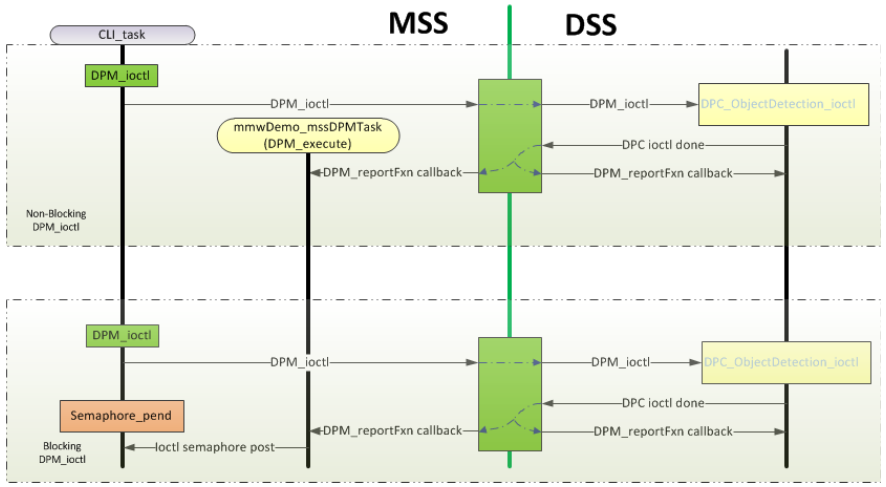
```
"--define=APP_RESOURCE_FILE="<ti/demo/xwr16xx/mmw/mmw_res.h>"
```

in mmw.mak when building the DPC sources as part of building the demo application and is referred in object detection DPC sources where needed as

```
#include APP_RESOURCE_FILE
```

Design Notes

Due to the limitation of DPM local queue size, for certain DPM functions such as DPM_start, DPM_stop and some of the DPC control through DPM_ioctl, semaphores are used to sync between calling task and function **MmwDemo_DPC_ObjectDetection_reportFxn**. So that it won't cause DPM crash because of running out of DPM local queues. The following diagram demonstrates the example calling flow for blocking DPM_ioctl() function call. Non-blocking DPM_ioctl is also shown for comparison.



DPM_ioctl calling flow

There are DPM report functions on both MSS and DSS for the same DPM_Report. However the sequence is not guaranteed between the two cores.

Memory Usage

Memory usage summary

The table below shows the usage of various memories available on the device across the demo application and other SDK components. The table is generated using the demo's map file and applying some mapping rules to it to generate a condensed summary. The numeric values shown here represent bytes.

MSS: For the mapping rules, please refer to demo_mss_mapping.txt. Refer to the xwr16xx_mmw_demo_mss_mem_analysis_detailed.txt for detailed analysis of the memory usage across drivers and control/alg components on MSS and to demo_mss_mapping_detailed.txt for detailed mapping rules .

```
                                OVERVIEW
Memory                           Used              Total        Percent Used
--------------------------------------------------------------------------
DATA_RAM                        57692             196608              29.34%
HS_RAM                              0              32768               0.00%
L3_RAM                              0             786432               0.00%
PROG_RAM                       115950             261888              44.27%
VECTORS                            60                256              23.44%


                          Type           DATA_RAM       HS_RAM       L3_RAM     PROG_RAM      VECTORS
--------------------------------------------------------------------------------------------------
APP                       code              16920            0            0        46625           60
APP                       heap              34816            0            0            0            0
BIOS                      code                  4            0            0        22443            0
COMPONENTS_CORE           code               1668            0            0        30134            0
COMPONENTS_OPTIONAL       code               2236            0            0        15244            0
linker-generated          linker             2048            0            0          321            0
linker-generated          unknown               0            0            0         1183            0
```

DSS: For the mapping rules, please refer to demo_dss_mapping.txt. Refer to the xwr16xx_mmw_demo_dss_mem_analysis_detailed.txt for detailed analysis of the memory usage across drivers and control/alg components on DSS and to demo_dss_mapping_detailed.txt for detailed mapping rules .

```
                                OVERVIEW
Memory                           Used              Total        Percent Used
--------------------------------------------------------------------------
PAGE 0:
HSRAM                           32768              32768             100.00%
L1DSRAM                         16384              16384             100.00%
L1PSRAM                             0              16384               0.00%
L2SRAM_UMAP0                   109394             131072              83.46%
L2SRAM_UMAP1                   131072             131072             100.00%
L3SRAM                         786432             786432             100.00%
PAGE 1:
L3SRAM                              0             786432               0.00%


                          Type             HSRAM      L1DSRAM      L1PSRAM L2SRAM_UMAP0 L2SRAM_UMAP1       L3SRAM
--------------------------------------------------------------------------------------------------------------
ALG                       code                 0            0            0         2336        12512            0
APP                       code                 0            0            0        24243        18560            0
APP                       heap             32768        16384            0        70664            0       786432
BIOS                      code                 0            0            0         5264        44416            0
COMPONENTS_CORE           code                 0            0            0         4828        55584            0
linker-generated          linker               0            0            0         2059            0            0
```

Copyright 2020, Texas Instruments Incorporated