MMWAVE SDK User Guide

Product Release 3.1.1

Release Date: Jan 18, 2019

Document Version: 1.0

COPYRIGHT

DISCLAIMER

This mmWave SDK User guide is generic and contains details about all the mmWave devices that are supported by TI in general. However, note that not all mmWave devices may be supported in a given mmWave SDK release. Please refer to the mmWave SDK Release notes to understand the list of devices/platforms supported in a given mmWave SDK release.

CONTENTS

1 Out-of-box mmWave Experience
2 System Overview
- 2.1 mmWave Suite
- 2.2 mmWave Demos
- 2.3 External Dependencies
- 2.4 Terms used in this document
- 2.5 Related documentation/links

3 Getting started
- 3.1 Programming mmWave devices
- 3.2 Loading images onto mmWave EVM
  - 3.2.1 Demonstration Mode
  - 3.2.2 CCS development mode
- 3.3 Running the Demos
  - 3.3.1 mmWave Demo
- 3.4 Configuration (.cfg) File Format
- 3.5 Running the prebuilt unit test binaries (.xer4f and .xe674)

4 How-To Articles
- 4.1 How to identify the COM ports for mmWave EVM
- 4.2 How to flash an image onto mmWave EVM
- 4.3 How to erase flash on mmWave EVM
- 4.4 How to connect mmWave EVM to CCS using JTAG
  - 4.4.1 Emulation Pack Update
  - 4.4.2 Device support package Update
  - 4.4.3 Target Configuration file for CCS (CCXML)
    - 4.4.3.1 Creating a CCXML file
    - 4.4.3.2 Connecting to mmWave EVM using CCXML in CCS
- 4.5 Developing using SDK
  - 4.5.1 Build Instructions
  - 4.5.2 Setting up build environment
    - 4.5.2.1 Windows
    - 4.5.2.2 Linux
  - 4.5.3 Building demo
    - 4.5.3.1 Building demo in Windows
    - 4.5.3.2 Building demo in Linux
  - 4.5.4 Advanced build
    - 4.5.4.1 Building drivers/control/alg components
    - 4.5.4.2 "Error on warning" compiler and linker setting

5 MMWAVE SDK deep dive
- 5.1 System Deployment
- 5.2 Typical mmWave Radar Processing Chain
- 5.3 Typical Programming Sequence
  - 5.3.1 RF Control Path
    - 5.3.1.1 Single RF Control (MSSRADARSS or DSSRADARSS)
    - 5.3.1.2 Co-operative RF control ((MSS+DSS)<->RADARSS)
  - 5.3.2 Data Path
    - 5.3.2.1 Data processing flow with local domain control
    - 5.3.2.2 Data processing flow with remote domain control
    - 5.3.2.3 Distributed Data processing flow and control
- 5.4 mmWave SDK - TI components
  - 5.4.1 Drivers
  - 5.4.2 OSAL
  - 5.4.3 mmWaveLink
  - 5.4.4 mmWave API
    - 5.4.4.1 Full configuration
    - 5.4.4.2 Minimal configuration
  - 5.4.5 Datapath Interface (DPIF)
  - 5.4.6 Data Processing Units (DPUs)
  - 5.4.7 Data Path Manager (DPM)
  - 5.4.8 Data processing chain (DPC)
  - 5.4.9 mmWaveLib

5.4.10 Group Tracker
5.4.11 RADARSS Firmware
5.4.12 CCS Debug Utility
5.4.13 HSI Header Utility
5.4.14 Secondary Bootloader
5.4.15 mmWave SDK - System Initialization
5.4.15.1 ESM
5.4.15.2 SOC
5.4.15.3 Pinmux
5.4.16 Usecases
5.4.16.1 Data Path tests using Test vector method
5.4.16.2 CSI-2 based streaming of ADC data
5.4.16.3 Basic configuration of Front end and capturing ADC data in L3 memory
6 Appendix
6.1 Memory usage
6.2 Register layout
6.3 Enable DebugP logs
6.4 Shared memory usage by SDK demos
6.5 mmWave Device Image Creator
6.6 mmw Demo: cryptic message seen on DebugP_assert
6.7 How to execute Idle instruction in idle task when using SYSBIOS
6.8 Range Bias and Rx Channel Gain/Offset Measurement and Compensation
6.9 Guidelines on optimizing memory usage
6.10 How to add a .const (table) beyond L3 heap in mmWave application where overlay is enabled
6.11 Enabling L3 cache for DSP/C674x on mmWave devices
6.12 DSPlib integration in mmWave C674x based application (Using 2 libraries simultaneously)
6.12.1 Integrating individual functions from each library
6.12.2 Patching the installation
6.13 SDK Demos: miscellaneous information
6.14 Data size restriction for a given session when sending data over LVDS
6.15 CCS Debugging of real time application
6.15.1 Inter-chirp debugging
6.15.2 Inter-frame debugging
6.15.3 Using non-real time chain test code
6.15.4 Using printfs in real time
6.15.5 Viewing hardware registers
6.15.6 Viewing expressions/memory in real time

LIST OF FIGURES

Figure 1: mmWave Demo Visualizer- mmWave Device Connectivity

- Figure 2: Chirp Diagram
- Figure 3: mmWave EVM PC Connectivity - Device Manager - COM Ports

Figure 4: Creating a mmWave device CCXML in CCS
Figure 5: Connecting to mmWave Device in CCS

- Figure 6: Autonomous mmWave sensor (Standalone mode)
- Figure 7: SDK Layered block diagram
- Figure 8: Typical mmWave radar processing chain
- Figure 9: Typical mmWave radar processing chain using mmWave SDK components
- Figure 10: Scalable data processing chain using mmWave SDK

Figure 11: Typical mmWave radar control flow
Figure 12: mmWave Isolation mode: Detailed Control Flow (Init sequence)
Figure 13: mmWave Isolation mode: Detailed Control Flow (Config sequence)
Figure 14: mmWave Isolation mode: Detailed Control Flow (start sequence)
Figure 15: mmWave Co-operative Mode: Detailed Control Flow (Init sequence)
Figure 16: mmWave Co-operative Mode: Detailed Control Flow (Config sequence)
Figure 17: mmWave Co-operative Mode: Detailed Control Flow (Start sequence)
Figure 18: Typical mmWave Detection Processing Layers
Figure 19: Data processing flow with local domain control (init/config)
Figure 20: Data processing flow with local domain control (start/chirp/frame/stop)
Figure 21: Data processing flow with remote domain control (init/config)
Figure 22: Data processing flow with remote domain control (start/chirp/frame/stop)
Figure 23: Distributed Data processing flow and control (init/config)

Figure 24: Distributed Data processing flow and control (start/chirp/frame/stop)
Figure 25: mmWave SDK Drivers - Internal software design
Figure 26: mmWaveLink - Internal software design
Figure 27: mmWave API - Internal software design
Figure 28: mmWave API - 'Minimal' Config - Sample flow (mmWave devices with MSS and DSS cores and module in co-operative mode)
Figure 29: mmWave API - 'Minimal' Config - Sample flow (mmWave devices with single core or when module is used in isolation mode)
Figure 30: DPU - Internal software design
Figure 31: DPU - typical call flow
Figure 32: Datapath manager (DPM) - internal software design

- Figure 33: Sample ROV log with debug prints

LIST OF TABLES

Table 1: mmWave SDK Demos - CLI commands and parameters
Table 2: Supported drivers and their functionality

1. Out-of-box mmWave Experience

To experience the mmWave technology offered by TI, you will need to procure the following

- Hardware
    1. mmWave TI EVM
    2. Power supply cable as recommended in TI EVM user guide
    3. PC
- Software
    1. Pre-flashed mmWave Demo running on TI EVM (See instructions in this user guide on how to update the flashed demo)
    2. Chrome browser running on PC

Next, to visualize the data flowing out of TI mmWave devices, follow these steps

1. Connect the EVM to a power outlet via the power cable and to the PC via the included USB cable. EVM should be powered up and connected to PC now.
2. On your PC, browse to https://dev.ti.com/mmWaveDemoVisualizer in Chrome browser and follow the prompts to install one-time software. [No other software installation is needed at this time]
3. The Visualizer app should detect and connect to your device via COM ports automatically (except for the very first time where users will need to confirm the selection via OptionsSerial Port). Select the right Platform and SDK version and start your evaluation!
    1. **Hint** : Use HelpAbout to know your Platform and SDK version

For details on how to evaluate, any troubleshooting needs and/or to understand the know-how behind these steps, continue reading this SDK User Guide...

If the flashed demo on the EVM is an old version and you would like to upgrade to latest demo, continue reading this SDK User Guide...

2. System Overview

The mmWave SDK is split in two broad components: mmWave Suite and mmWave Demos.

2. 1. mmWave Suite

mmWave Suite is the foundational software part of the mmWave SDK and would encapsulate these smaller components:

- Drivers
- OSAL
- mmWaveLink
- mmWaveLib
- mmWave API
- Data processing layer (manager, processing units)
- RADARSS Firmware
- Board Setup and Flash Utilities

2. 2. mmWave Demos

SDK provides demos that depict the various control and data processing aspects of a mmWave application. Data visualization of the demo's output on a PC is provided as part of these demos. These demos are example code that are provided to customers to understand the inner workings of the mmWave devices and the SDK and to help them get started on developing their own application.

- mmWave Processing Demo with TI Gallery App - "mmWave Demo Visualizer"

2. 3. External Dependencies

All tools/components needed for building mmWave sdk are included in the mmwave sdk installer. But the following external components (for debugging) are not included in the mmWave SDK.

- CCS (for debugging)

Please refer to the mmWave SDK Release Notes for detailed information on these external dependencies and the list of platforms that are supported.

2. 4. Terms used in this document

Terms used	Comment
xWR	This is used throughout the document where that section/component/module applies to both AWR and IWR variants
BSS	This is used in the source code and sparingly in this document to signify the RADARSS. It is also interchangeably referred to as the mmWave Front End.
MSS	Master Sub-system. It is also interchangeably referred to as Cortex R4F.
DSS	DSP Sub-system. It is also interchangeably referred to as DSS or C674x core.

2. 5. Related documentation/links

Other than the documents included in the mmwave_sdk package the following documents/links are important references.

- SoC links:
  - Automotive mmWave Sensors
  - Industrial mmWave Sensors
- Evaluation Modules (EVM) links:
  - Automotive Evaluation modules (Booster Pack, DEVPACK)
  - Industrial Evaluation modules (Booster Pack, ISK)

3. Getting started

The best way to get started with the mmWave SDK is to start running one of the various demos that are provided as part of the package. TI mmWave EVM comes pre-flashed with the mmWave demo. However, the version of the pre-flashed demo maybe older than the SDK version mentioned in this document. Users can follow this section and upgrade/run the flashed demo version. The demos (source and pre-built binaries) are placed at mmwave_sdk_<ver>/packages/ti/demo/<platform> folder.

mmWave Demo

This demo is located at mmwave_sdk_<ver>/packages/ti/demo/<platform>/mmw folder. The millimeter wave demo shows some of the radar sensing and object detection capabilities of the 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. A detailed explanation of this demo is available in the demo's docs folder: mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\docs\doxygen\html\index.html. This demo ships out detected objects and other real-time information that can be visualized using the TI Gallery App - 'mmWave Demo Visualizer' hosted at https://dev.ti.com/mmWaveDemoVisualizer. The version of the mmWave Demo running on TI mmWave EVM can be obtained from the Visualier app using the HelpAbout menu.

Following sections describe the general procedure for booting up the device with the demos and then executing it.

3. 1. Programming mmWave devices

Here is a little insight into the mmWave devices and the programmable cores they offer. For more detailed information, please refer to the Technical reference manual for the respective mmWave device. These details are needed when loading the binaries using CCS and/or to understand the various terminologies that exist in the "Getting started" section.

xWR14xx

xWR14xx has one cortex R4F core available for user programming and is referred to in this section as MSS or R4F. The demos and the unit tests executable are provided to be loaded on MSS/R4F.

xWR16xx/xWR18xx/xWR68xx

These devices have one cortex R4F core and one DSP C674x core available for user programming and are referred to as MSS/R4F and DSS /C674X respectively. The demos have 2 executables - one for MSS and one for DSS which should be loaded concurrently for the demos to work. See Running the Demos section for more details. The unit tests may have executables for either MSS or DSS or both. These executables are meant to be run in standalone operation. This means MSS unit test executable can be loaded and run on MSS R4F without downloading any code on DSS. Similarly, DSS unit test execuable can be loaded and run on DSS C674x without downloading any code on DSS. The exceptions to this are the Mailbox unit test named "test_mss_dss_msg_exchange", mmWave unit tests under full and minimal and datapath manager (DPM) unit tests.

3. 2. Loading images onto mmWave EVM

User can choose either one of these modes for loading images onto the EVM.

3. 2. 1. Demonstration Mode

This mode should be used when either upgrading the factory flashed binaries on the EVM to latest SDK version using the pre-built binaries provided in the SDK release or for field deployment of mmWave sensors.

1. Follow the procedure mentioned in the section (How to flash an image onto mmWave EVM). Use the mmwave_sdk_<ver>/packages/ti/demo/<platform>/<demo> /<platform>_<demo>.bin as the METAIMAGE1 file name.
2. Remove the "SOP2" jumper or toggle the SOP2 switch to OFF and reboot the device to run the demo image every time on power up. No other image loading step is required on subsequent boot to run the demo.

3. 2. 2. CCS development mode

This mode should be used when debugging with CCS is involved and/or developing an mmWave application where the .bin files keep changing constantly and frequent flashing of image onto the board is not desirable. This mode allows you to flash once and then use CCS to download a different image to the device's RAM on every boot.

This mode is the recommended way to run the unit tests for the drivers and components which can be found in the respective test directory for that component. See section mmWave SDK - TI components for location of each component's test code

boot-up sequence

When the mmWave device boots up in functional mode, the device bootloader starts executing and checks if a serial flash is attached to the device. If yes, then it expects valid MSS application (and a valid RADARSS firmware and/or DSS application) to be present on the flash. During development phase of mmWave application, flashing the device with the application under development for every small change can be cumbersome. To overcome this, user should perform a one-time flash as mentioned in the steps below. The actual user application under development can then be loaded and reloaded to the MSS program memory (TCMA) and/or DSP L2/L3 memory (only for mmWave devices with DSP) directly via CCS in the device's functional mode.

Refer to Help inside Code Composer Studio (CCS) to learn more about connecting, loading, running the cores, in general.

1. EVM and CCS setup
    1. Follow the procedure mentioned in the section: How to flash an image onto mmWave EVM. Use mmwave_sdk_<ver> /packages/ti/utils/ccsdebug /<platform>_ccsdebug.bin as the METAIMAGE1 filename for the one-time flash.
    2. Follow the steps in How to connect mmWave EVM to CCS using JTAG to setup the environment for CCS connectivity.
2. With "SOP2" jumper removed or SOP2 switch toggled to off, after every power cycle/reboot of the EVM, follow these steps to load the application:
    1. Power up the EVM
    2. Launch ccxml file created in step 1.b above.
    3. If the test requires an application to run on MSS
        1. Connect CCS to Cortex_R4_0
        2. Load the MSS program. (for example: xwr16xx_<module>_mss.xer4f prebuilt executables provided in the SDK release package)
    4. If the test requires an application to run on DSP
        1. Connect CCS to C674X_0
        2. Load the DSS program. (for example: xwr16xx_<module>_dss.xe674 prebuilt executables provided in the SDK release package)
    5. Run the R4 and/or C674 cores
    6. To reload, disconnect the connected cores, power cycle and connect again

3. 3. Running the Demos

Follow this subsection to experience the mmWave functionality using the out-of-box mmWave demo. Before you proceed further, make sure that you have loaded the right demo binary using the section above, set the EVM to functional mode and powered up the device. Connect the EVM to the PC using its XDS110 micro-USB port/cable.

3. 3. 1. mmWave Demo

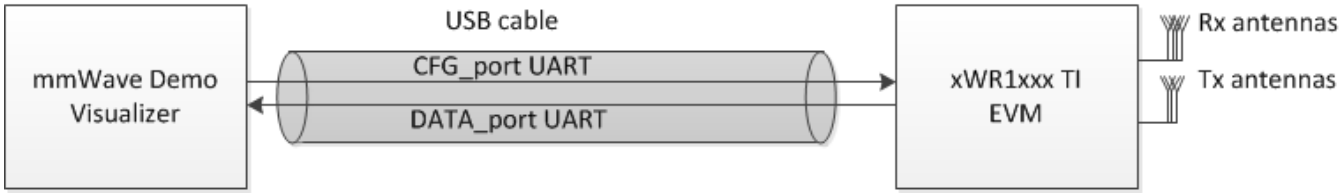


Figure 1: mmWave Demo Visualizer- mmWave Device Connectivity

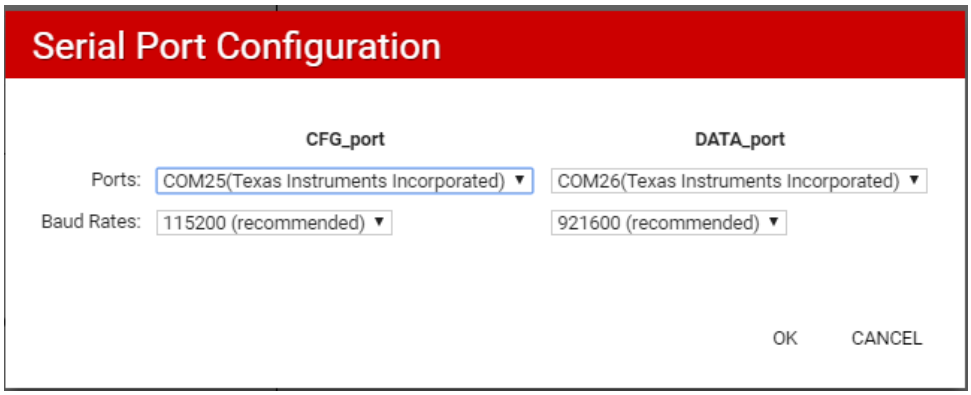
1. Power on the EVM in functional mode with right binary loaded (see section above) and connect it to the PC as shown above with the USB cable.
2. Browse to the TI gallery app "mmWave Demo Visualizer" at http://dev.ti.com/gallery or use the direct link https://dev.ti.com /mmWaveDemoVisualizer. Use HelpREADME.md from inside this app for more information on how to run/configure this app.

First Time Setup

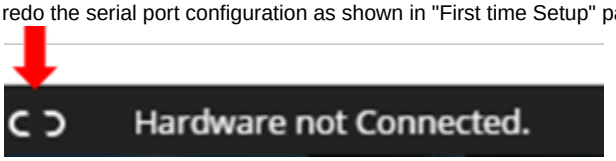
1. If this is the first time you are using this App, you may be requested to install a plug-in and the TI Cloud Agent Application. This step will also install the right **XDS110 drivers** needed for UART port detection.
2. Once the demo is running on the mmWave sensors and the USB is connected from the board to the PC, the app will try to automatically detect the COM ports for your device.
    1. If auto-detection doesn't work, then you will need to configure the serial ports in this App. Run the device manager on the PC and locate the following COM ports as shown in the section "How to identify the COM ports for mmWave EVM" below. In the Visualizer App, go to the Menu->Options->Serial Port and perform the settings as shown below.
        - **CFG_port**: Use COM port number for "**XDS110 Class Application/User UART**": Baud: 115200. This is the port where **CLI (command line interface)** runs for all the demos.
        - **Data_port**: Use COM port "**XDS110 Class Auxiliary Data port**": Baud: 921600. This is the port on which binary data generated by the processing chain in the mmWave demo will be received by the PC. This is the detected object list and its properties (range, doppler, angle, etc).

COM Port

Please note that the COM port numbers on your setup maybe different from the one shown below. Please use the correct COM port number from your setup for following steps.



1. At this point, this app will automatically try to connect to the target (mmWave Sensor). If it does not connect or if the connection fails, you should try to connect to the target by clicking in the bottom left corner of this App. If that fails as well, redo the serial port configuration as shown in "First time Setup" panel above.



2. After the App is connected to the target, you can select the configuration parameters (Frequency Band, Platform, etc) in the "Setup details" and "Scene Selection" area of the **CONFIGURE** tab.
3. Besides selecting the configuration parameters, you should select which plots you want to see. This can be done using the "check boxes" in the "Plot Selection" area. Adjust the frame rate depending on number of plots you want to see. For selecting heatmap plots, set frame rate to less than or equal to 4 fps. When selecting frame rate to be 25-30fps, for better GUI performance, select only the scatter plot and statistics plot.
4. Once the configuration is selected, you can send the configuration to the device (use "SEND CONFIG TO MMWAVE DEVICE" button).
5. After the configuration is sent to the device, you can switch to the **PLOTS** view/tab and the plots that you selected will be shown.
6. You can switch back from "Plots" tab to "Configure" tab, reconfigure your "Scene Selection", "Object Detection" and/or "Plot Selection" values and re-send the configuration to the device to try a different profile. After a new configuration has been selected, just press the "SEND CONFIG TO MMWAVE DEVICE" button again and the device will be reconfigured. This can be done without rebooting the device. If you change the parameters in the "Setup Details", then you will need to take further action before trying the new configurations
    1. If Platform is changed: make sure the COM ports match the TI EVM/platform you are trying to configure and visualizer
    2. If SDK version is changed: make sure the mmW demo running on the connected TI EVM matches the selected SDK version in the GUI
    3. If Antenna Config is changed: make sure the TI EVM is rebooted before sending the new configuration.
3. If board is rebooted, follow the steps starting from 1 above.

COM port after reboot

Whenever TI EVM is power-cycled (rebooted), you will need to use the bottom left serial port connection icon inside TI gallery app "mmWave Demo Visualizer" for disconnecting and reconnecting the COM ports. Note that if you used the CLI COM port directly to send the commands (instead of TI gallery app) you will have to close the CLI teraterm window and open a new one on every reboot.

Inner workings of the GUI

In the background, GUI performs the following steps:

- Creates or reads the configuration file and sends to the mmWave device using the COM port called **CFG_port**. It saves the information locally to be able to make sense of the incoming data that it will display. Refer to the CFG Section for details on the configuration file contents.
- Receives the data generated by the demo on the visualization/Data COM port and processes it to create various displays based on the GUI configuration in the cfg file.
  - The format of the data streamed out of the demo is documented in mmw demo's doxygen mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\docs\doxygen\html\index.html under section: "Output information sent to host".
- On every reconfiguration, it sends a 'sensorStop' command to the device first to stop the active run of the mmWave device. Next, it sends the command 'flushCfg' to flush the old configuration before sending the new configuration. It is mandatory to flush the old configuration before sending a new configuration. Additionally, it is mandatory to send all the commands for that demo/platform even if the user desires the functionality to be disabled i.e. no commands are optional.

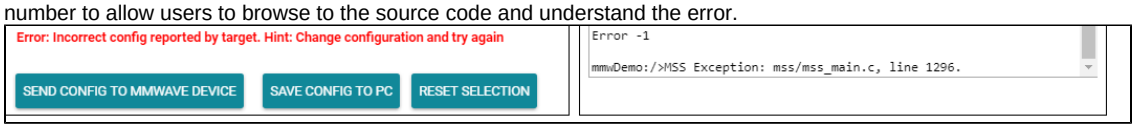
Advanced GUI options

- User can configure the device from their own configuration file or the saved app-generated configuration file by using the "LOAD CONFIG FROM PC AND SEND" button on the **PLOTS** tab. Make sure the first two commands in this config file are "sensorStop" followed by "flushCfg".
- User can temporarily pause the mmWave sensor by using the "STOP" button on the plots tab. The sensor can be restarted by using the "START" button. In this case, sensor starts again with the already loaded configuration and no new configuration is sent from the App.
- User can simultaneously plot and record the processed/detected objects data coming out of the DATA_port using the "RECORD START" button in the plots tab. Set the max limits for file size or record time as per your requirements to prevent infinite capturing of data. The saving of data can be manually stopped using the "Record Stop" button (if the max limits are not reached).

Console Messages window in Visualizer

Console message window echoes the following debug information for the users

- Every command that is sent to the TI mmWave EVM and the response back from the EVM
- Any runtime assert conditions detected by the demo running on TI mmWave EVM after the sensor is started. This is helpful when mmW demo is flashed onto the EVM and CCS connectivity is not available. It spits out file name and line number to allow users to browse to the source code and understand the error.

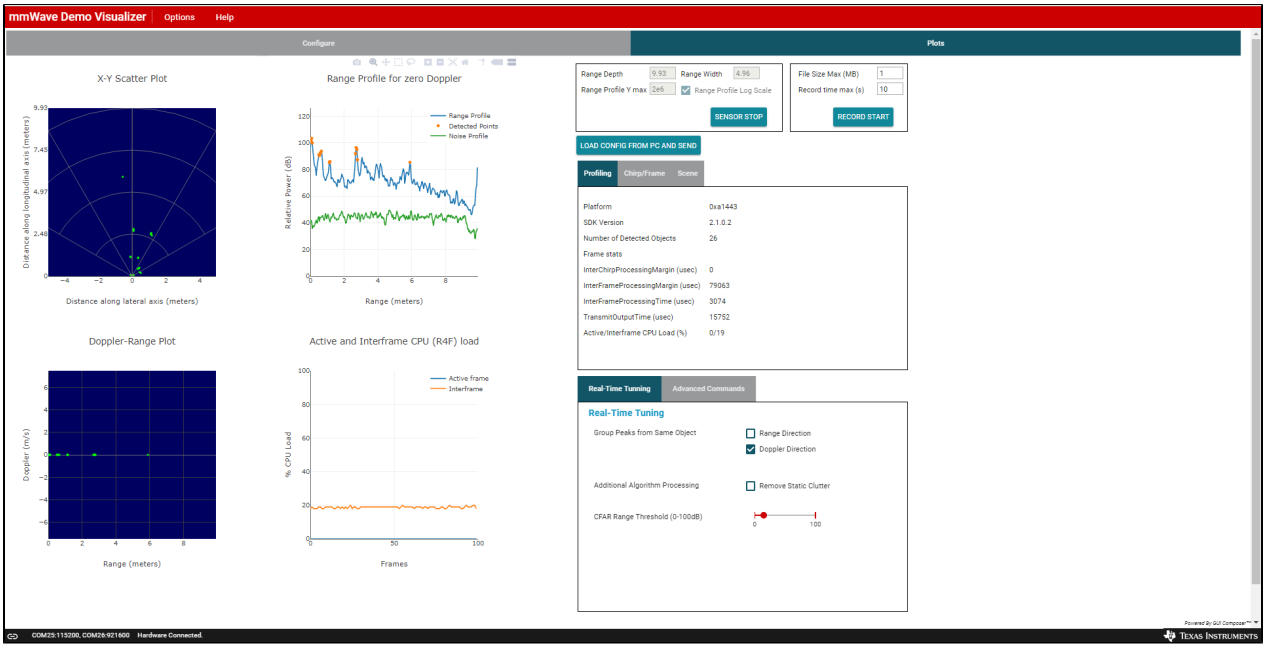


- Init time calibration status after the first sensorStart is issued post reboot for debugging boot time or start failures

```
mmwDemo:/>sensorStart
Debug: Init Calibration Status = 0x7fe

Done
```

Here is an example of plots that mmWave Demo Visualizer produces based on the config that is passed to the demo application running on mmWave sensor.


mmWave Demo Visualizer
Options
Help
Configure
Plots
X-Y Scatter Plot
Distance along longitudinal axis (meters)
Distance along lateral axis (meters)
Range Profile for zero Doppler
Range Profile
Detected Points
Noise Profile
Relative Power (dB)
Range (meters)
Doppler-Range Plot
Doppler (m/s)
Range (meters)
Active and Interframe CPU (R4F) load
Active frame
Interframe
% CPU Load
Frames
Range Depth
Range Width
Range Profile Y max
Range Profile Log Scale
SENSOR STOP
File Size Max (MB)
Record time max (s)
RECORD START
LOAD CONFIG FROM PC AND SEND
Profiling
Chirp/Frame
Scene
Platform
0xa1443
SDK Version
2.1.0.2
Number of Detected Objects
26
Frame stats
InterChirpProcessingMargin (usec)
0
InterFrameProcessingMargin (usec)
79063
InterFrameProcessingTime (usec)
3074
TransmitOutputTime (usec)
15752
Active/Interframe CPU Load (%)
0/19
Real-Time Tunning
Advanced Commands
Real-Time Tuning
Group Peaks from Same Object
Range Direction
Doppler Direction
Additional Algorithm Processing
Remove Static Clutter
CFAR Range Threshold (0-100dB)
COM25:115200, COM26:921600 Hardware Connected.
Texas Instruments

3. 4. Configuration (.cfg) File Format

Each line in the .cfg file describes a command with parameters. The various commands and their arguments are described in the table below (arguments are in sequence). For mmW demo, users can create their own config files from the Visualizer GUI by using the "Save Config to PC" button or starting from the few sample profiles provided in the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\profiles directory.

Converting configuration from older SDK release to current SDK release

As new versions of SDK releases are available, there are usually changes to the configuration commands that are supported in the new release. Now, users may have some hand crafted config file which worked perfectly well on older SDK release version but will not work as is with the new SDK release. If user desires to run the same configuration against the new SDK release, then there is a script mmwDemo_<platform>_update_config.pl provided in the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\profiles directory that they can use to convert the configuration file from older release to a compatible version for the new release. Refer to the perl file for details on how to run the script. Note that users will need to install perl on their machine (there is no strict version requirement at this time). For any new commands inserted by the script, there will be a comment preceeding that line which is similar to something like this: "Inserting new mandatory command. Check users guide for details."

Most of the parameters described below are the same as the mmwavelink API specifications (see doxygen mmwave_sdk_<ver>\packages\ti\control\mmwavelink\docs\doxygen\html\index.html.) Additionally, users can refer to the chirp diagram below to understand the chirp and profile related parameters.

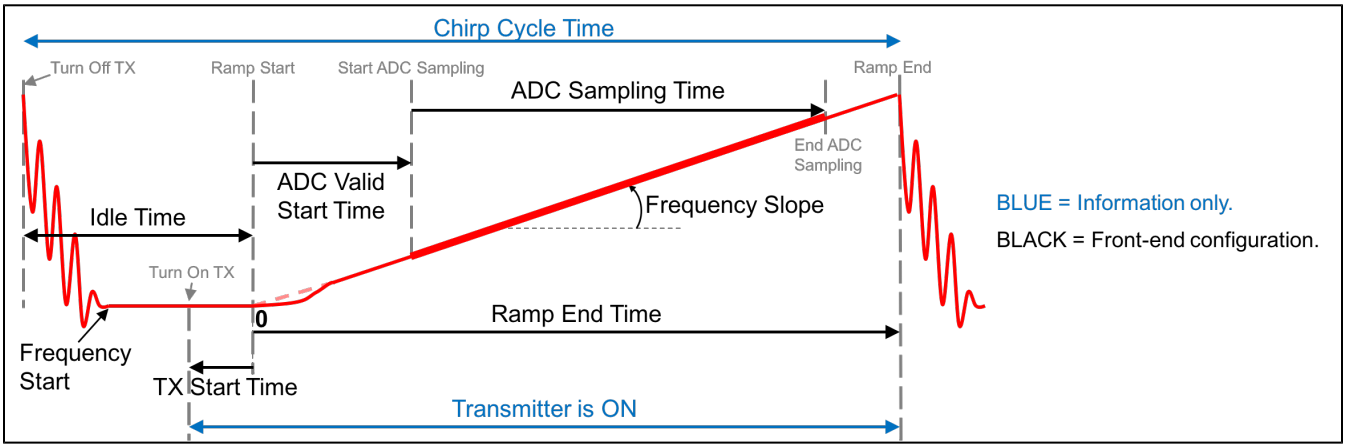


Figure 2: Chirp Diagram

Configuration command	Command details	Command Parameters	Usage in mmW demo xwr68xx/xwr18xx
dfeDataOutputMode	The values in this command should not change between sensorStop and sensorStart. Reboot the board to try config with different set of values in this command This is a mandatory command.	 <modeType> 1 - frame based chirps 2 - continuous chirping 3 - advanced frame config	 only option 1 and 3 are supported
channelCfg	Channel config message to RadarSS. See mmwavelink doxgen for details. The values in this command should not change between sensorStop and sensorStart. Reboot the board to try config with different set of values in this command This is a mandatory command.	 <rxChannelEn> Receive antenna mask e.g for 4 antennas, it is 0x1111b = 15	 4 antennas supported

		<txChannelEn> Transmit antenna mask	The 2 azimuth antennas can be enabled using bitmask 0x5 (i.e. tx1 and tx3) The azimuth and elevation antennas can be enabled using bitmask 0x7 (i.e. tx1, tx2 and tx3)
		<cascading> SoC cascading, not applicable, set to 0	n/a
adcCfg	ADC config message to RadarSS. See mmwavelink doxgen for details. The values in this command should not change between sensorStop and sensorStart. Reboot the board to try config with different set of values in this command This is a mandatory command.		
		<numADCBits> Number of ADC bits (0 for 12-bits, 1 for 14-bits and 2 for 16-bits)	only 16-bit is supported
		<adcOutputFmt> Output format : 0 - real 1 - complex 1x (image band filtered output) 2 - complex 2x (image band visible))	only complex modes are supported
adcbufCfg	adcBuf hardware config. The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1. For advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<adcOutputFmt> ADCBUF out format 0-Complex, 1-Real	only complex modes are supported
		<SampleSwap> ADCBUF IQ swap selection: 0-I in LSB, Q in MSB, 1-Q in LSB, I in MSB	only option 1 is supported
		<ChanInterleave> ADCBUF channel interleave configuration: 0 - interleaved(only supported for XWR14xx), 1 - non-interleaved	only option 1 is supported
		<ChirpThreshold> Chirp Threshold configuration used for ADCBUF buffer to trigger ping/pong buffer switch. Valid values: 0-8 for demos that use DSP for 1D FFT only 1 for demos that use HWA for 1D FFT	only value of 1 is supported
profileCfg	Profile config message to RadarSS and datapath. See mmwavelink doxgen for details. The values in this command can be changed between sensorStop and sensorStart.		

This is a mandatory command.

txCalibEnCfg Field

This CLI command doesn't expose the txCalibEnCfg field in the mmwavelink structure. User should follow the mmwavelink documentation and update the CLI profileCfg handler function accordingly. The current handler sets the value to 0 for this field (backward compatible mode)

Combination of numAdcSamples in profileCfg (and numRangeBins), numDopplerChirps = total number of chirps/(num TX in MIMO mode) in frameCfg or subFrameCfg, number of TX and RX antennas in channelCfg and chirpCfg determine the size of Radarcube and other internal buffers /heap in the demo. It is possible that some combinations of these values result in out of memory conditions for these heaps and demo will reject such configuration. Refer to demo and DPC doxygen to understand the data buffer layout and use the system printfs on sensorStart in CCS console window to understand the exact heap usage for a given configuration.

<profileId> profile Identifier	Legacy frame (dfeOutputMode=1): could be any allowed value but only one valid profile per config is supported Advanced frame (dfeOutputMode=3): could be any allowed value but only one profile per subframe is supported. However, different subframes can have different profiles
<startFreq> "Frequency Start" in GHz (float values allowed) Examples: 77 60.25	any value as per mmwavelink doxgen/device datasheet but represented in GHz
<idleTime> "Idle Time" in u-sec (float values allowed) Examples: 7 7.15	any value as per mmwavelink doxgen/device datasheet but represented in usec
<adcStartTime> "ADC Valid Start Time" in u-sec (float values allowed) Examples: 7 7.34	any value as per mmwavelink doxgen/device datasheet but represented in usec
<rampEndTime> "Ramp End Time" in u-sec (float values allowed) Examples: 58 216.15	any value as per mmwavelink doxgen/device datasheet but represented in usec
<txOutPower> Tx output power back-off code for tx antennas	only value of '0' has been tested within context of mmW demo
<txPhaseShifter> tx phase shifter for tx antennas	only value of '0' has been tested within context of mmW demo
<freqSlopeConst> "Frequency slope" for the chirp in MHz/usec (float values allowed) Examples: 68 16.83	any value greater than 0 as per mmwavelink doxgen /device datasheet but represented in MHz/usec

		<txStartTime> "TX Start Time" in u-sec (float values allowed) Examples: 1 2.92	any value as per mmwavelink doxgen/device datasheet but represented in usec
		<numAdcSamples> number of ADC samples collected during "ADC Sampling Time" as shown in the chirp diagram above Examples: 256 224	any value as per mmwavelink doxgen/device datasheet
		<digOutSampleRate> ADC sampling frequency in ksps. (<numAdcSamples> / <digOutSampleRate> = "ADC Sampling Time") Examples: 5500	any value as per mmwavelink doxgen/device datasheet
		<hpfCornerFreq1> HPF1 (High Pass Filter 1) corner frequency 0: 175 KHz 1: 235 KHz 2: 350 KHz 3: 700 KHz	any value as per mmwavelink doxgen/device datasheet
		<hpfCornerFreq2> HPF2 (High Pass Filter 2) corner frequency 0: 350 KHz 1: 700 KHz 2: 1.4 MHz 3: 2.8 MHz	any value as per mmwavelink doxgen/device datasheet
		<rxGain> OR'ed value of RX gain in dB and RF gain target (See mmwavelink doxgen for details)	any value as per mmwavelink doxgen/device datasheet
chirpCfg	Chirp config message to RadarSS and datapath. See mmwavelink doxgen for details. The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command.		
		chirp start index	any value as per mmwavelink doxygen
		chirp end index	any value as per mmwavelink doxygen
		profile identifier	should match the profileCfg->profileId
		start frequency variation in Hz (float values allowed)	only value of '0' has been tested within context of mmW demo
		frequency slope variation in kHz/us (float values allowed)	only value of '0' has been tested within context of mmW demo
		idle time variation in u-sec (float values allowed)	only value of '0' has been tested within context of mmW demo

		ADC start time variation in u-sec (float values allowed)	only value of '0' has been tested within context of mmW demo
		tx antenna enable mask (Tx2,Tx1) e.g (10)b = Tx2 enabled, Tx1 disabled.	See note under "Channel Cfg" command above. Individual chirps should have either only one distinct Tx antenna enabled (MIMO) or same TX antennas should be enabled for all chirps
lowPower	Low Power mode config message to RadarSS. See mmwavelink doxgen for details. The values in this command should not change between sensorStop and sensorStart. Reboot the board to try config with different set of values in this command. This is a mandatory command.		
		<don't_care>	set to 0
		ADC Mode 0x00 : Regular ADC mode 0x01 : Low power ADC mode	use value of '0' or '1' (depending on profileCfg->digOutSampleRate)
frameCfg	frame config message to RadarSS and datapath. See mmwavelink doxgen for details. dfeOutputMode should be set to 1 to use this command The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command when dfeOutputMode is set to 1.		
		chirp start index (0-511)	any value as per mmwavelink doxgen but corresponding chirpCfg should be defined
		chirp end index (chirp start index-511)	any value as per mmwavelink doxgen but corresponding chirpCfg should be defined
		number of loops (1 to 255)	any value as per mmwavelink doxgen/device datasheet but greater than or equal to 4. Note: If value of 2 is desired for number of Doppler Chirps, one must update the demo /object detection DPC source code to use rectangular window for Doppler DPU instead of Hanning window.
		number of frames (valid range is 0 to 65535, 0 means infinite)	any value as per mmwavelink doxgen
		frame periodicity in ms (float values allowed)	any value as per mmwavelink doxgen and represented in msec. However frame should not have more than 50% duty cycle (i.e. active chirp time should be <= 50% of frame period). Also it should allow enough time for selected UART output to be shipped out (selections based on guiMonitor command) else demo will assert if the next frame start trigger is received from the front end and current frame is still ongoing. User can use the output of stats TLV to tune this parameter.
		trigger select 1: Software trigger 2: Hardware trigger.	only option for Software trigger is supported
		Frame trigger delay in ms (float values allowed)	any value as per mmwavelink doxgen and represented in msec.
advFrameCfg	Advanced config message to RadarSS and datapath. See mmwavelink doxgen for details. The		

	dfeOutputMode should be set to 3 to use this command. See profile_advanced_subframe.cfg profile in the mmW demo profiles directory for example usage. The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command when dfeOutputMode is set to 3.	<numOfSubFrames> Number of sub frames enabled in this frame	any value as per mmwavelink doxgen
		<forceProfile> Force profile	only value of 0 is supported
		<numFrames> Number of frames to transmit (1 frame = all enabled sub frames)	any value as per mmwavelink doxgen
		<triggerSelect> trigger select 1: Software trigger 2: Hardware trigger.	only option for Software trigger is supported
		<frameTrigDelay> Frame trigger delay in ms (float values allowed)	any value as per mmwavelink doxgen and represented in msec.
subFrameCfg	Subframe config message to RadarSS and datapath. See mmwavelink doxgen for details. The dfeOutputMode should be set to 3 to use this command. See profile_advanced_subframe.cfg profile in the mmW demo profiles directory for example usage The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command when dfeOutputMode is set to 3.		
		<subFrameNum> subframe Number for which this command is being given	value of 0 to RL_MAX_SUBFRAMES-1
		<forceProfileIdx> Force profile index	ignored as <forceProfile> in advFrameCfg should be set to 0
		<chirpStartIdx> Start Index of Chirp	any value as per mmwavelink doxgen but corresponding chirpCfg should be defined
		<numOfChirps> Num of unique Chirps per burst including start index	any value as per mmwavelink doxgen but corresponding number of chirpCfg should be defined
		<numLoops> No. of times to loop through the unique chirps	any value as per mmwavelink doxgen but greater than or equal to 4
		<burstPeriodicity> Burst periodicty in msec (float values allowed) and meets the criteria burstPeriodicity >= (numLoops)* (numOfChirps) + InterBurstBlankTime	any value as per mmwavelink doxgen and represented in msec but subframe should not have more than 50% duty cycle and allow enough time for selected UART output to be shipped out (selections based on guiMonitor command)
		<chirpStartIdxOffset> Chirp Start address increament for next burst	set it to 0 since demo supports only one burst per subframe
		<numOfBurst> Num of bursts in the subframe	set it to 1 since demo supports only one burst per subframe
		<numOfBurstLoops> Number of times to loop over the set of above defined bursts, in the sub frame	set it to 1 since demo supports only one burst per subframe
		<subFramePeriodicity> subFrame periodicty in msec (float values allowed) and meets the criteria subFramePeriodicity >= Sum total time of all bursts + InterSubFrameBlankTime	set to same as <burstPeriodicity> since demo supports only one burst per subframe

guiMonitor	Plot config message to datapath. The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command.		
		All parameters below are flags (1 to enable and 0 to disable)	
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<detected objects> 1 - enable export of point cloud (x,y,z, doppler) and point cloud sideinfo (SNR, noiseval) 2 - enable export of point cloud (x,y,z,doppler) 0 - disable	all values supported
		<log magnitude range> 1 - enable export of log magnitude range profile at zero Doppler 0 - disable	all values supported
		<noise profile> 1 - enable export of log magnitude noise profile 0 - disable	all values supported
		<rangeAzimuthHeatMap> range-azimuth heat map related information 1 - enable export of zero Doppler radar cube matrix, all range bins, all antennas to calculate and display azimuth heat map. 0 - disable (the GUI computes the FFT of this to show heat map)	all values supported
		<rangeDopplerHeatMap> range-doppler heat map 1 - enable export of the whole detection matrix. Note that the frame period should be adjusted according to UART transfer time. 0 - disable	all values supported
		<statsInfo> statistics (CPU load, margins, etc) 1 - enable export of stats data. 0 - disable	all values supported
cfarCfg	CFAR config message to datapath. The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.

		<procDirection> Processing direction: 0 – CFAR detection in range direction 1 – CFAR detection in Doppler direction	all values supported; 2 separate commands need to be sent; one for Range and other for doppler
		<mode> CFAR averaging mode: 0 - CFAR_CA (Cell Averaging) 1 - CFAR_CAGO (Cell Averaging Greatest Of) 2 - CFAR_CASO (Cell Averaging Smallest Of)	all values supported
		<noiseWin> noise averaging window length: Length of the noise averaged cells in samples	supported
		<guardLen> guard length in samples	supported
		<divShift> Cumulative noise sum divisor expressed as a shift. Sum of noise samples is divided by 2^<divShift>. Based on platform, <mode> and <noiseWin> , this value should be set as shown in next columns. The value to be used here should match the "CFAR averaging mode" and the "noise averaging window length" that is selected above. The actual value that is used for division (2^x) is a power of 2, even though the "noise averaging window length" samples may not have that restriction.	CFAR_CA: <divShift> = $\log_2$(2 x <noiseWin>) CFAR_CAGO/_CASO: <divShift> = $\log_2$ (<noiseWin>) In profile_2d.cfg, value of 3 means that the noise sum is divided by 2^3=8 to get the average of noise samples with window length of 8 samples in CFAR -CASO mode.
		cyclic mode or Wrapped around mode. 0- Disabled 1- Enabled	used for programming the CFAR engine inside HWA
		Threshold scale. This is used in conjuntion with the noise sum divisor (say x). the CUT comparison for log input is: CUT > Threshold scale + (noise sum / 2^x)	Detection threshold is specified as log2 value, expressed in Q9 format. The threshold value can be converted from the value expressed in dB as Tcli = 512 x TdB / 6 x N / N' where N is numVirtualAntennas N' = 2^ceil(log2(N)). Note: log input is used for this mmw demo
		peak grouping 0 - disabled 1 - enabled	supported
multiObjBeamForming			

	Multi Object Beamforming config message to datapath. This feature allows radar to separate reflections from multiple objects originating from the same range /Doppler detection. The procedure searches for the second peak after locating the highest peak in Azimuth FFT. If the second peak is greater than the specified threshold, the second object with the same range/Doppler is appended to the list of detected objects. The threshold is proportional to the height of the highest peak. The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.	<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<Feature Enabled> 0 - disabled 1 - enabled	supported
		<threshold> 0 to 1 – threshold scale for the second peak detection in azimuth FFT output. Detection threshold is equal to <thresholdScale> multiplied by the first peak height. Note that FFT output is magnitude squared.	supported
calibDcRangeSig	DC range calibration config message to datapath. Antenna coupling signature dominates the range bins close to the radar. These are the bins in the range FFT output located around DC. When this feature is enabled, the signature is estimated during the first N chirps, and then it is subtracted during the subsequent chirps. During the estimation period the specified bins (defined as [negativeBinIdx, positiveBinIdx]) around DC are accumulated and averaged. It is assumed that no objects are present in the vicinity of the radar at that time. This procedure is initiated by the following CLI command, and it can be initiated any time while radar is running. Note that the maximum number of compensated bins is 32. The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<enabled> Enable DC removal using first few chirps 0 - disabled 1 - enabled	supported
		<negativeBinIdx> negative Bin Index (to remove DC from farthest range bins) Maximum negative range FFT index to be included for compensation. Negative indices are indices wrapped around from far end of 1D FFT. Ex: Value of -5 means last 5 bins starting from the farthest bin	supported
		<positiveBinIdx> positive Bin Index (to remove DC from closest range bins) Maximum positive range FFT index to be included for compensation Value of 8 means first 9 bins (including bin#0)	supported

		<numAvg> number of chirps to average to collect DC signature (which will then be applied to all chirps beyond this). Value of 256 means first 256 chirps (after command is issued and feature is enabled) will be used for collecting (averaging) DC signature in the bins specified above. From 257th chirp, the collected DC signature will be removed from every chirp.	The value must be power of 2, and must be greater than the number of Doppler bins.
clutterRemoval	Static clutter removal config message to datapath. Static clutter removal algorithm implemented by subtracting from the samples the mean value of the input samples to the 2D-FFT The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<enabled> Enable static clutter removal technique 0 - disabled 1 - enabled	supported
aoaFovCfg	Command for datapath to filter out detected points outside the specified range in azimuth or elevation plane The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<minAzimuthDeg>	minimum azimuth angle (in degrees) that specifies the start of field of view
		<maxAzimuthDeg>	maximum azimuth angle (in degrees) that specifies the end of field of view
		<minElevationDeg>	minimum elevation angle (in degrees) that specifies the start of field of view
		<maxElevationDeg>	maximum elevation angle (in degrees) that specifies the end of field of view
cfarFovCfg	Command for datapath to filter out detected points outside the specified limits in the range direction or doppler direction The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<subFrameIdx> subframe Index	For legacy mode, that field should be set to -1 whereas for advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.

		<procDirection> Processing direction: 0 – point filtering in range direction 1 – point filtering in Doppler direction	both values supported but this command should be given twice - one for range direction and other for doppler direction
		<min (meters or m/s)> the units depends on the value for <procDirection> field above. meters for Range direction and meters/sec for Doppler direction	minimum limits for the range or doppler below which the detected points are filtered out
		<max (meters or m/s)> the units depends on the value for <procDirection> field above. meters for Range direction and meters/sec for Doppler direction	maximum limits for the range or doppler above which the detected points are filtered out
compRangeBiasAndRxChanPhase	Command for datapath to compensate for bias in the range estimation and receive channel gain and phase imperfections. Refer to the procedure mentioned here The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<rangeBias> Compensation for range estimation bias in meters	supported
		<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)> Set of Complex value representing compensation for virtual Rx channel phase bias in Q15 format. Pairs of I and Q should be provided for all Tx and Rx antennas in the device	12 pairs of values should be provided here since the device has 4 Rx and 3 Tx (total of 12 virtual antennas)
measureRangeBiasAndRxChanPhase	Command for datapath to enable the measurement of the range bias and receive channel gain and phase imperfections. Refer to the procedure mentioned here The values in this command can be changed between sensorStop and sensorStart and even when the sensor is running. This is a mandatory command.		
		<enabled> 1 - enable measurement. This parameter should be enabled only using the profile_calibration.cfg profile in the mmW demo profiles directory 0 - disable measurement. This should be the value to use for all other profiles.	supported
		<targetDistance> distance in meters where strong reflector is located to be used as test object for measurement. This field is only used when measurement mode is enabled.	supported

		<searchWin> distance in meters of the search window around <targetDistance> where the peak will be searched	supported
sensorStart	sensor Start command to RadarSS and datapath. Starts the sensor. This function triggers the transmission of the frames as per the frame and chirp configuration. By default, this function also sends the configuration to the mmWave Front End and the processing chain. This is a mandatory command.		
		Optionally, user can provide an argument 'doReconfig' 0 - Skip reconfiguration and just start the sensor using already provided configuration. <any other value> - not supported	supported
sensorStop	sensor Stop command to RadarSS and datapath. Stops the sensor. If the sensor is running, it will stop the mmWave Front End and the processing chain. After the command is acknowledged, a new config can be provided and sensor can be restarted or sensor can be restarted without a new config (i.e. using old config). See 'sensorStart' command. This is mandatory before any reconfiguration is performed post sensorStart.		supported
flushCfg	This command should be issued after 'sensorStop' command to flush the old configuration and provide a new one. This is mandatory before any reconfiguration is performed post sensorStart.		
%		Any line starting with '%' character is considered as comment line and is skipped by the CLI parsing utility.	

Table 1: mmWave SDK Demos - CLI commands and parameters

3. 5. Running the prebuilt unit test binaries (.xer4f and .xe674)

Unit tests for the drivers and components can be found in the respective test directory for that component. See section "mmWave SDK - TI components" for location of each component's test code. For example, UART test code that runs on TI RTOS is in mmwave_sdk_<ver>/packages/ti/drivers/uart/test/<platform>. In this test directory, you will find .xer4f and .xe674 files (either prebuilt or build as a part of instructions mentioned in "Building drivers/control components"). Follow the instructions in section "CCS development mode" to download and execute these unit tests via CCS.

4. How-To Articles

4. 1. How to identify the COM ports for mmWave EVM

When the EVM is powered on and connected to Windows PC via the supplied USB cable, there should be two additional COM Ports in Device Manager. See your mmWave devices' TI EVM User Guide for details on the COM port.

Troubleshooting Tip

If the COM ports don't show up in the Device Manager or are not working (i.e. no demo output seen on the data port), then one of these steps would apply depending on your setup:

1. If you want to run the Out-of-box demo, simple browse to the Visualizer (https://dev.ti.com/mmWaveDemoVisualizer) and follow the one-time setup instructions.
2. If you are trying to flash the board, using Uniflash tool and following the cloud or desktop version installation instructions would also install the right drivers for the COM ports.
3. If above methods didnt work and if TI code composer studio is not installed on that PC, then download and install the stand alone XDS110 drivers.
4. If TI code composer studio is installed, then version of CCS and emulation package need to be checked and updated as per the mmWave SDK release notes. See section Emulation Pack Update for more details.

After following the above steps, disconnect and re-connect the EVM and you should see the COM ports now. See the highlighted COM ports in the Figure below

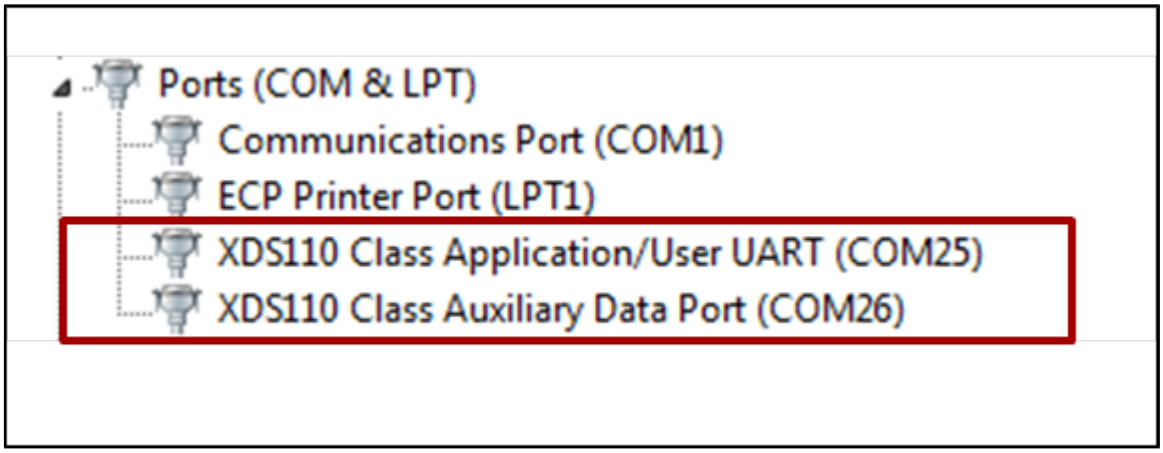


Figure 3: mmWave EVM PC Connectivity - Device Manager - COM Ports

COM Port

Please note that the COM port numbers on your setup maybe different from the one shown above. Please use the correct COM port number from your setup for following steps.

4. 2. How to flash an image onto mmWave EVM

You will need the mmWave Device TI EVM, USB cable and a Windows/Linux PC to perform these steps.

1. **Setup the Booster Pack EVM for Flashing**

   Refer to the EVM User Guide to understand the bootup modes of the EVM and the SOP jumper/switch locations (See "Sense-on-Power (SOP) Jumpers" section in mmWave device's EVM user guide). To put the EVM in flashing mode, power off the board and either place jumpers on pins marked as SOP2 and SOP0 or toggle SOP0 and SOP2 switches to ON .

SOP2 jumper /switch	SOP1 jumper /switch	SOP0 jumper /switch	Bootloader mode & operation
0	0	1	**Functional Mode** Device Bootloader loads user application from QSPI Serial Flash to internal RAM and switches the control to it
1	0	1	**Flash Programming Mode** Device Bootloader spins in loop to allow flashing of user application to the serial flash.

2. **Procure the Images**
   For flashing xWR1xxx devices, TI Uniflash tool should be used. Users can either use the cloud version available at https://dev.ti.com/uniflash/ or download the desktop version available at http://www.ti.com/tool/UNIFLASH. Detailed instructions on how to use the GUI are described in the Uniflash document " **UniFlash User Guide for mmWave Devices** " located at http://processors.wiki.ti.com/index.php/Category:CCS_UniFlash. This document talks about the steps from the perspective of desktop GUI but the flashing steps (except for installation) should apply for cloud version as well. For the SDK packaged demos and ccsdebug utility, there is a bin file provided in their respective folder: <platform>_<demo|ccsdebug>.bin which is the metaImage to be used for flashing. The metaImage already has the MSS, BSS (RADARSS) and DSS (as applicable) application combined into one file. These bin files can be selected in Uniflash based on the working mode. Users can use these instructions to flash the metaImage of their custom demo as well.
   1. For demo mode, mmwave_sdk_<ver>\ti\demo\<platform>\mmw\<platform>_mmw_demo.bin should be selected.
   2. For CCS development mode, mmwave_sdk_<ver>\ti\ utils\ccsdebug\<platform>_ccsdebug.bin should be selected.
3. **Flashing procedure**

   Power up the EVM and check the Device Manager in your windows PC. Note the number for the serial port marked as "**XDS110 Class Application/User UART**" for the EVM. Lets say for this example, it showed up as COM25. Use this COM port in the TI Uniflash tool. Follow the remaining instructions in the " **UniFlash v4 User Guide for mmWave Devices** " to complete the flashing.
4. **Switch back to Functional Mode**

   Refer to the EVM User Guide to understand the bootup modes of the EVM and the SOP jumpers (See "Sense-on-Power (SOP) Jumpers" section in mmWave device's EVM user guide). To put the EVM in functional mode, power off the board and remove jumpers from "SOP2" pin and leave the jumper on "SOP0" pin or toggle SOP0 switch to ON and SOP2 switch to OFF.

4. 3. How to erase flash on mmWave EVM

1. Setup the Booster Pack EVM for flashing as mentioned in step 1 of the section: How to flash an image onto mmWave EVM
2. Follow the instructions in " **UniFlash v4 User Guide for mmWave Devices** " section "**Format SFLASH Button**".
3. Switch back to Functional Mode as mentioned in step 4 of the section: How to flash an image onto mmWave EVM

4. 4. How to connect mmWave EVM to CCS using JTAG

Debug/JTAG capability is available via the same XDS110 micro-USB port/cable on the EVM. TI Code composer studio would be required for accessing the debug capability of the device. Refer to the release notes for TI code composer studio and emulation pack version that would be needed.

4. 4. 1. Emulation Pack Update

Refer to the mmWave SDK release notes for the emulation pack version that would be needed within CCS to connect to the EVM. Check if that particular or its later version of "TI Emulators" is available within your CCS installation. If you have an older version on your system, refer to CCS help on how to update software packages within CCS.

4. 4. 2. Device support package Update

To create the ccxml file for connecting to the EVM, you will need to first update the device support package within CCS. Refer to the mmWave SDK release notes for the device support package version that would be needed within CCS to connect to the EVM. Check if that particular or its later version of "mmWave Radar Device Support" is available within your CCS installation. If you have an older version on your system, refer to CCS help on how to update software packages within CCS.

4. 4. 3. Target Configuration file for CCS (CCXML)

4. 4. 3. 1. Creating a CCXML file

Assuming you have updated the device support package and Emulation pack as mentioned in the section above, follow the steps mentioned below to create a target configuration file in CCS.

1. If your CCS does not already show "Target Configurations" window, do View->Target Configurations
2. This will show the "Target Configurations" window, right click in the window and select "New Target Configuration"
3. Give an appropriate name to the ccxml file you want to create for the EVM
4. Scroll the "Connection" list and select "Texas Instruments XDS110 USB Debug Probe", when this is done, the "Board or Device" list will be filtered to show the possible candidates, find and choose the mmWave device (AWR or IWR) of interest and check the box. Click Save and the file will be created.

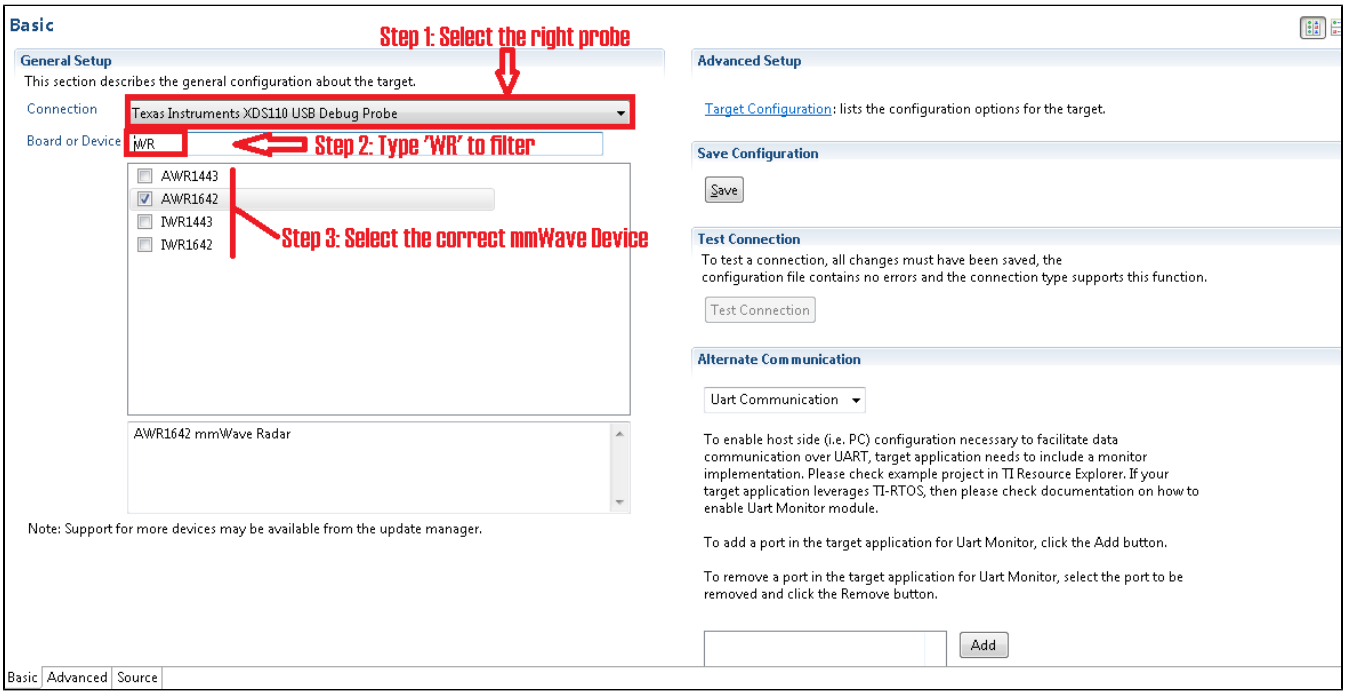


Figure 4: Creating a mmWave device CCXML in CCS

4. 4. 3. 2. Connecting to mmWave EVM using CCXML in CCS

Follow steps in above section to create a ccxml file. Once created, the target configuration file will be seen in the "Target Configurations" list and you can launch the target by selecting it and with right-click select the "Launch Selected Configuration" option. This will launch the target and the Debug window will show all the cores present on the device. You can connect to the target with right-click and doing "Connect Target".

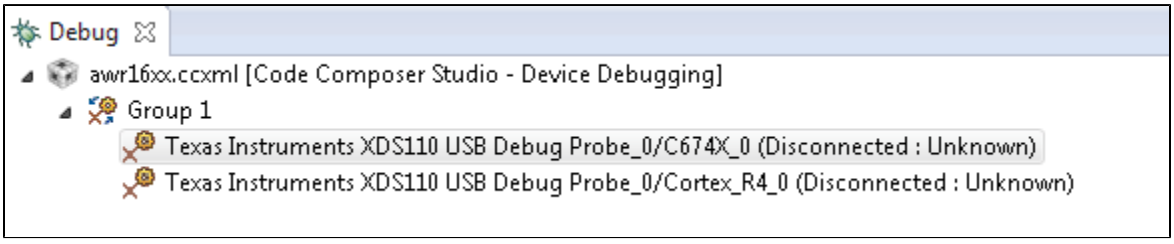


Figure 5: Connecting to mmWave Device in CCS

4. 5. Developing using SDK

4. 5. 1. Build Instructions

Follow the mmwave_sdk_release_notes instructions to install the mmwave_sdk in your development environment (windows or linux). All the tools needed for mmwave sdk build are installed as part of mmwave sdk installer.

4. 5. 2. Setting up build environment

4. 5. 2. 1. Windows

1. Create command prompt at <mmwave_sdk_<ver> install path>\packages\scripts\windows folder. Under this folder you should see a setenv.bat file that has all the tools environment variables set automatically based on the installation folder. Review this file and change the few build variables shown below (if needed) and save the file. Please note that the rest of the environment variables should not be modified if the standard installation process was followed.

Build variables that can be modified (if needed) in setenv.bat

```
@REM ###########################################################################################
@REM # Build variables (to be modified based on build need)
@REM ###########################################################################################
@REM Select your device. Options (case sensitive) are: awr14xx, iwr14xx, awr16xx, iwr16xx,
awr18xx, iwr18xx, iwr68xx
set MMWAVE_SDK_DEVICE=iwr68xx

@REM If download via CCS is needed, set below define to yes else no
@REM yes: Out file created can be loaded using CCS.
@REM Binary file created can be used to flash
@REM no: Out file created cannot be loaded using CCS.
@REM Binary file created can be used to flash
@REM (additional features: write-protect of TCMA, etc)
set DOWNLOAD_FROM_CCS=yes

@REM If using a secure device this variable needs to be updated with the path to
mmwave_secdev_<ver> folder
set MMWAVE_SECDEV_INSTALL_PATH=

@REM If using a secure device, this variable needs to be updated with the path to hsimage.cfg file
that
@REM has customer specific certificate/key information. A sample hsimage.cfg file is in the secdev
package
set MMWAVE_SECDEV_HSIMAGE_CFG=%MMWAVE_SECDEV_INSTALL_PATH%/hs_image_creator/hsimage.cfg
```

Refer to the MMWAVE-SECDEV User Guide to setup environment needed for builds for high secure (HS) devices. For non secure devices the MMWAVE_SECDEV_INSTALL_PATH environment variable should be empty.

If you see the following line in the setenv.bat file then most probably the wrong installer was used (Linux installation being compiled under Windows)

set MMWAVE_SDK_TOOLS_INSTALL_PATH=__MMWAVE_SDK_TOOLS_INSTALL_PATH__

In a proper installation the __MMWAVE_SDK_TOOLS_INSTALL_PATH__ would have been replaced with the actual installation folder path

2. Run **setenv.bat** as shown below.

Run setenv.bat

```
setenv.bat
```

This should not give errors and should print the message **"mmWave Build Environment Configured"**. The build environment is now setup.

4. 5. 2. 2. Linux

1. Open a terminal and cd to <mmwave_sdk_<ver> install path>/packages/scripts/unix. Under this folder you should see a setenv.sh file that has all the tools environment variables set automatically based on the installation folder. Review this file and change the few build variables shown below (if needed) and save the file. Please note that the rest of the environment variables should not be modified if the standard installation process was followed.

Build variables that can be modified (if needed) in setenv.sh

```
##############################################################################
# Build variables (to be modified based on build need)
##############################################################################
@REM Select your device. Options (case sensitive) are: awr14xx, iwr14xx, awr16xx, iwr16xx,
awr18xx, iwr18xx, iwr68xx
set MMWAVE_SDK_DEVICE=iwr68xx

# If download via CCS is needed, set below define to yes else no
# yes: Out file created can be loaded using CCS.
# Binary file created can be used to flash
# no: Out file created cannot be loaded using CCS.
# Binary file created can be used to flash
# (additional features: write-protect of TCMA, etc)
export DOWNLOAD_FROM_CCS=yes

# If using a secure device, this variable needs to be updated with the path to mmwave_secdev_<ver>
folder
export MMWAVE_SECDEV_INSTALL_PATH=

# If using a secure device, this variable needs to be updated with the path to hsimage.cfg file
that
# has customer specific certificate/key information. A sample hsimage.cfg file is in the secdev
package
export MMWAVE_SECDEV_HSIMAGE_CFG=${MMWAVE_SECDEV_INSTALL_PATH}/hs_image_creator/hsimage.cfg
```

Refer to the MMWAVE-SECDEV User Guide to setup environment needed for builds for high secure (HS) devices. For non secure devices the MMWAVE_SECDEV_INSTALL_PATH environment variable should be empty.

If you see the following line in the setenv.sh file then most probably the wrong installer was used (Windows installation being compiled under Linux)

export MMWAVE_SDK_TOOLS_INSTALL_PATH=__MMWAVE_SDK_TOOLS_INSTALL_PATH__

In a proper installation the __MMWAVE_SDK_TOOLS_INSTALL_PATH__ would have been replaced with the actual installation folder path

2. Assuming build is on a Linux 64bit machine, install modules that allows Linux 32bit binaries to run. This is needed for Image Creator binaries

```
sudo dpkg --add-architecture i386
```

3. Install mono. One of the Image Creator binaries (out2rprc.exe) is a windows executable that needs mono to run in Linux environment

```
sudo apt-get --assume-yes install mono-complete
```

4. Run **setenv.sh** as shown below.

Run setenv.sh

```
source ./setenv.sh
```

This should not give errors and should print the message **"mmWave Build Environment Configured"**. The build environment is now setup.

4. 5. 3. Building demo

To clean build a demo, first make sure that the environment is setup as detailed in earlier section. Then run the following commands. On successful execution of the commands, the output is <demo>.xe* which can be used to load the image via CCS and <demo>.bin which can be used as the binary in the steps mentioned in section "How to flash an image onto mmWave EVM".

4. 5. 3. 1. Building demo in Windows

Building demo in windows

```
REM Fill <device type> with appropriate device that supports demo in a particular release
cd %MMWAVE_SDK_INSTALL_PATH%/ti/demo/<device type>/mmw

REM Clean and build
gmake clean
gmake all

REM Incremental build
gmake all

REM For example to build the mmw demo for iwr68xx
cd %MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr68xx/mmw
gmake clean
gmake all
REM This will create xwr68xx_mmw_demo_mss.xer4f & xwr68xx_mmw_demo.bin binaries
REM under %MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr68xx/mmw folder
```

4. 5. 3. 2. Building demo in Linux

Building demo in linux

```
# Fill <device type> with appropriate device that supports demo in a particular release
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/<device type>/mmw

# Clean and build
gmake clean
gmake all

# Incremental build
gmake all

# For example to build the mmw demo for iwr68xx
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr68xx/mmw
gmake clean
gmake all
# This will create xwr68xx_mmw_demo_mss.xer4f & xwr68xx_mmw_demo.bin binaries
# under ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr68xx/mmw folder
```

Each demo has dependency on various drivers and control components. The libraries for those components need to be available in their respective lib folders for the demo to build successfully.

4. 5. 4. Advanced build

The mmwave sdk package includes all the necessary libraries and hence there should be no need to rebuild the driver, algorithms or control component libraries. In case a modification has been made to any of these modules then the following section details how to build these components.

4. 5. 4. 1. Building drivers/control/alg components

To clean build driver, control, datapath or alg component and its unit test, first make sure that the environment is setup as detailed in earlier section. Then run the following commands

Building component in windows

```
cd %MMWAVE_SDK_INSTALL_PATH%/ti/<component_path_under_ti>
gmake clean
gmake all


REM The command will create the following file
REM     lib<component>_<device_type>.aer4f library under ti/<component_path_under_ti>/lib folder
REM If the module has unit test, it will also create
REM     <device_type>_<component>_mss.xer4f unit test binary under ti/<component_path_under_ti>/test
/<device_type> folder
REM If the device has a DSP and the driver supports DSP then the command will also create
REM     lib<component>_<device_type>.ae674 library for DSS under ti/<component_path_under_ti>/lib folder
REM If the module has unit test, it will also create
REM     <device_type>_<component>_dss.xe674 unit test binary for DSS under ti/<component_path_under_ti>
/test/<device_type> folder
REM Above paths are relative to %MMWAVE_SDK_INSTALL_PATH%/

REM For example to build the adcbuf lib and unit test
cd %MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf
gmake clean
gmake all

REM For example to build the mmwavelink lib and unit test
cd %MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink
gmake clean
gmake all

REM For example to build the aoaproc dpu lib
cd %MMWAVE_SDK_INSTALL_PATH%/ti/datapath/dpc/dpu/aoaproc
gmake clean
gmake all

REM Additional build options for each component can be found by invoking make help
gmake help
```

Building component in linux

```
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/<component_path_under_ti>
gmake clean
gmake all

# The command will create the following file
#     lib<component>_<device_type>.aer4f library under ti/<component_path_under_ti>/lib folder
# If the module has unit test, it will also create
#     <device_type>_<component>_mss.xer4f unit test binary under ti/<component_path_under_ti>/test
/<device_type> folder
# If the device has a DSP and the driver supports DSP then the command will also create
#     lib<component>_<device_type>.ae674 library for DSS under ti/<component_path_under_ti>/lib folder
# If the module has unit test, it will also create
#     <device_type>_<component>_dss.xe674 unit test binary for DSS under ti/<component_path_under_ti>/test
/<device_type> folder
# Above paths are relative to ${MMWAVE_SDK_INSTALL_PATH}/

# For example to build the adcbuf lib and unit test
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf
gmake clean
gmake all

# For example to build the mmwavelink lib and unit test
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink
gmake clean
gmake all

# For example to build the aoaproc dpu lib
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/datapath/dpc/dpu/aoaproc
gmake clean
```

```
gmake all
# Additional build options for each component can be found by invoking make help
gmake help
```

example output of make help for drivers and mmwavelink

```
**********************************************************************
* Makefile Targets for the ADCBUF
clean                -> Clean out all the objects
drv                  -> Build the Driver only
drvClean             -> Clean the Driver Library only
test                 -> Builds all the unit tests for the SOC
testClean            -> Cleans the unit tests for the SOC
**********************************************************************
```

example output of make help for mmwave control and alg component

```
**********************************************************************
* Makefile Targets for the mmWave Control
clean                -> Clean out all the objects
lib                  -> Build the Core Library only
libClean             -> Clean the Core Library only
test                 -> Builds all the Unit Test
testClean            -> Cleans all the Unit Tests
**********************************************************************
```

Please note that not all components are supported for all devices and not all components have unit tests. List of supported components for each device is listed in the Release Notes.

4. 5. 4. 2. "Error on warning" compiler and linker setting

By default, the SDK build uses "–emit_warnings_as_errors" option to help users identify certain common mistakes in code that are flagged as warning but could lead to unexpected results. If user desires to disable this feature, then please set the flag MMWAVE_DISABLE_WARNINGS_AS_ERRORS to 1 in the above mentioned setenv.bat or setenv.sh and invoke that file again to update the build environment.

5. MMWAVE SDK deep dive

5. 1. System Deployment

A typical mmWave application would perform these operations:

- Control and monitoring of RF front-end through mmaveLink
- Transport of external communications through standard peripherals
- Some radar data processing using DSP

Typical customer deployment for mmWave sensor is shown in Figure 6:

1. Application code for MSS and DSP-SS is downloaded from the serial flash memory **attached** to the mmWave device (via QSPI)
2. **Optional** high level control from remote entity
3. Sends **low speed data** output (objects detected) to remote entity
4. **Optional** high speed data (debug) sent out of device over LVDS

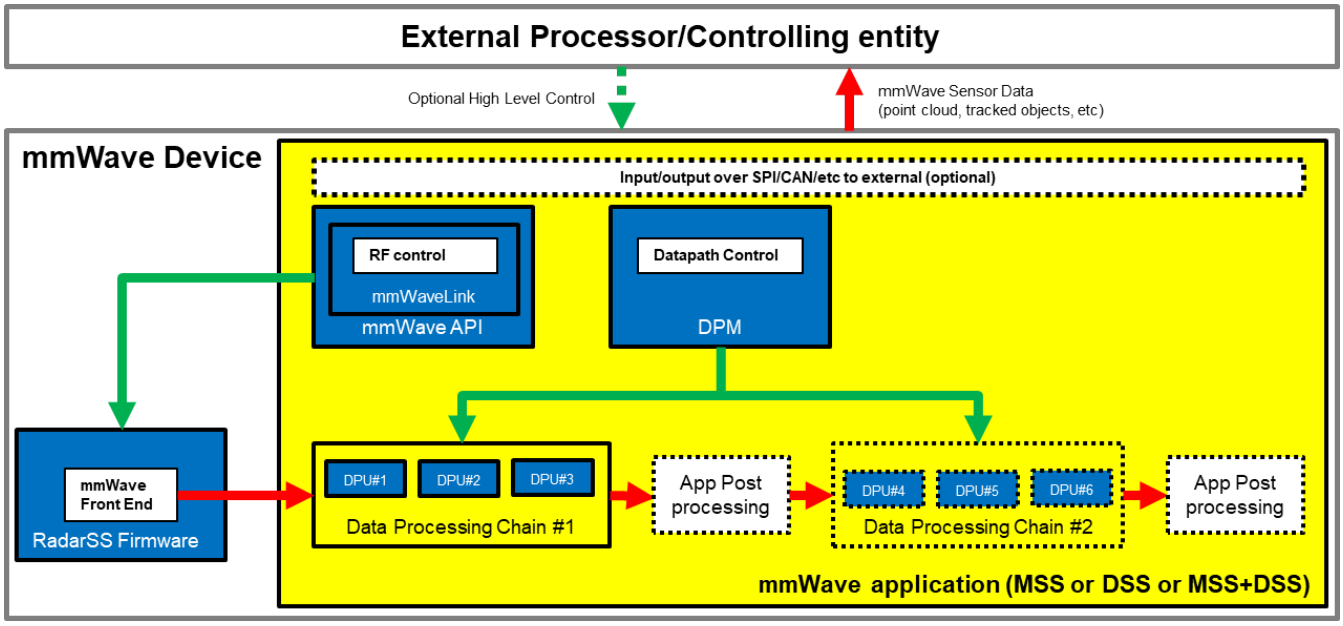


Figure 6: Autonomous mmWave sensor (Standalone mode)

The above deployment can be realized using the mmWave SDK and it components in a layered structure as shown below:

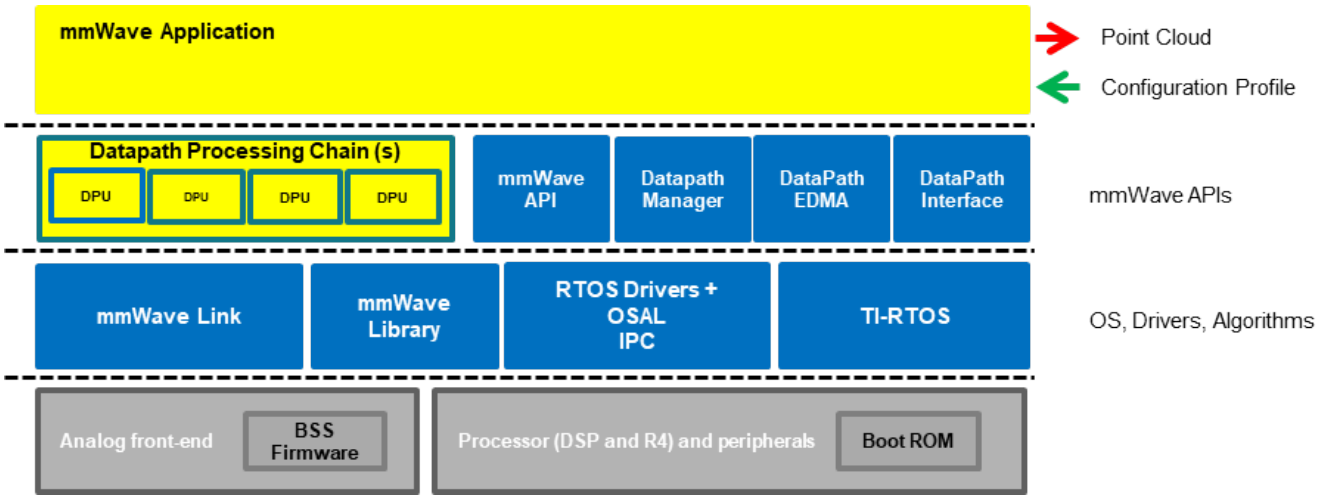


Figure 7: SDK Layered block diagram

5. 2. Typical mmWave Radar Processing Chain

Following figure shows a typical mmWave Radar processing chain that accepts ADC data as input from mmWave Front End and then performs Range and Doppler FFT followed by non-coherent detection using CFAR. Finally angle is estimated using 3D FFT and the detected points represent the point cloud data. The point cloud data can then be post processed using higher layer algorithms such as Clustering, Tracking, Classification to represent real world targets.

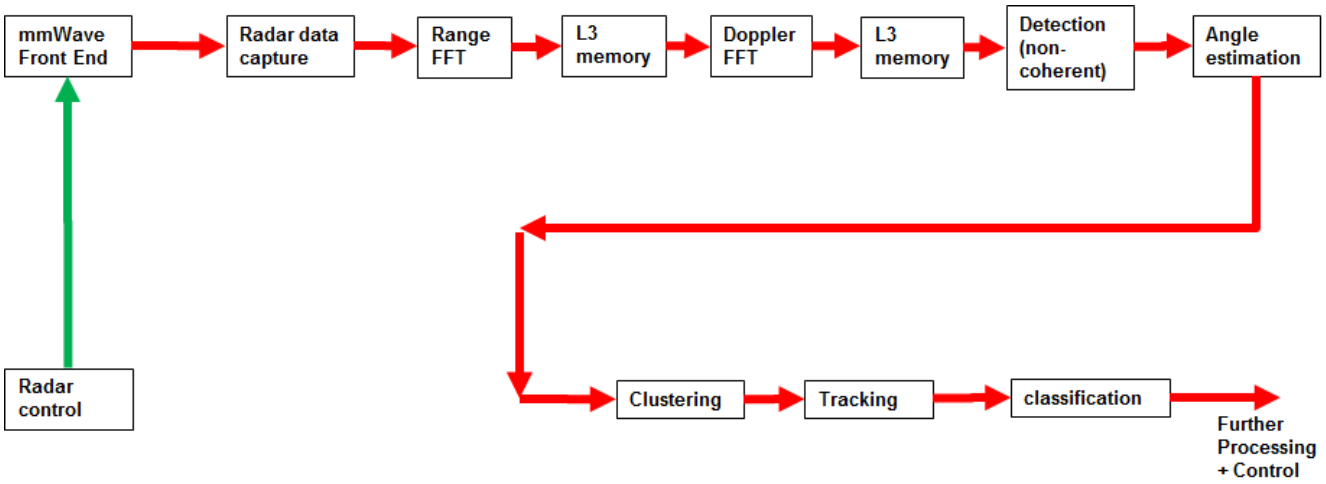


Figure 8: Typical mmWave radar processing chain

Using mmWave SDK the above chain could be realized as shown in the following figure for devices with HWA or DSP as processing nodes. In the following figure, green arrow shows the control path and red arrow shows the data path. Blue blocks are mmWave SDK components and yellow blocks are custom application code. The hierarchy of software flow/calls is shown with embedding boxes. Depending on the complexity of the higher algorithms (such as clustering, tracking, etc) and their memory/mips consumption, they can either be partially realized inside the mmWave device or would run entirely on the external processor.

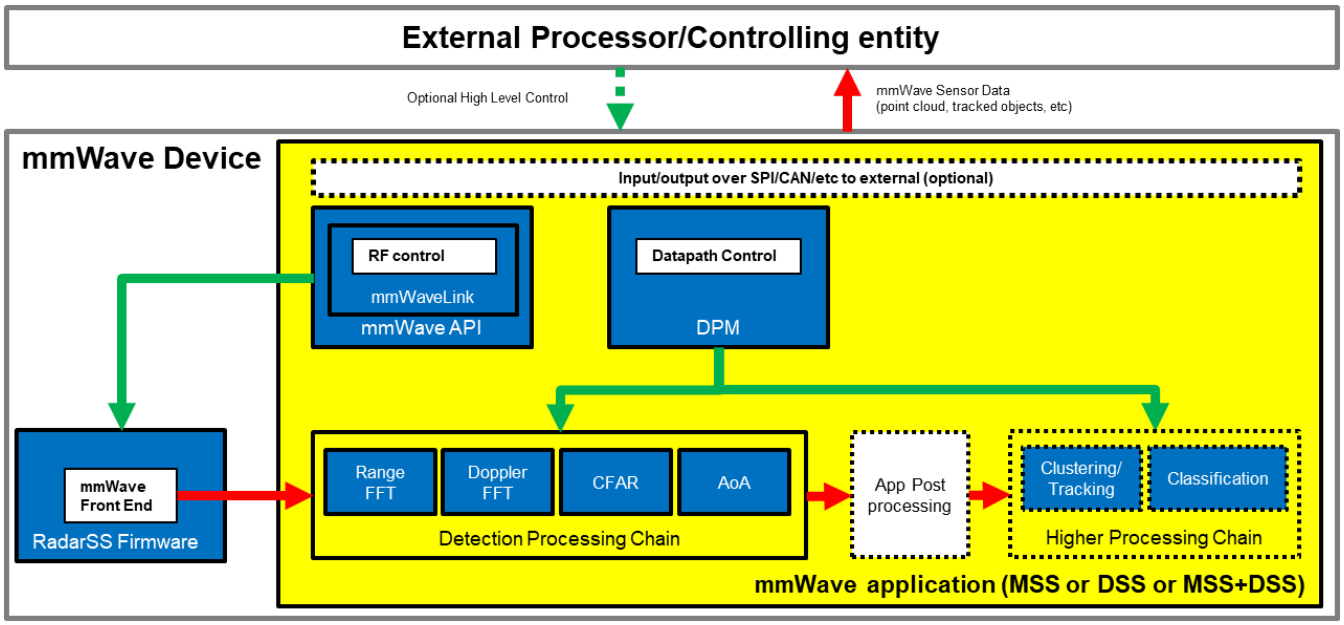


Figure 9: Typical mmWave radar processing chain using mmWave SDK components

Each of the mmWave device offers different processing nodes to realize the mmwave processing. xWR14xx has HWA engine, xWR16xx has DSP C674x core, xWR68xx and xWR18xx has HWA+DSP(C674x). For devices with multiple processing nodes, the mmWave detection processing chain can exploit them as needed for performance and scalable reasons. Shown below is an example of detection processing

chain that uses various data processing units (DPUs) to perform the typical mmwave processing upto the point cloud. The mmwave data representation in mmWave device memory forms an interface layer between the various DPUs. Each DPU can be realized independently using HWA or DSP processing node - the choice is either driven by usecase or availability of that processing node on a given mmWave device. The various mmWave SDK components shown below are described in the section "mmWave SDK - TI components" below.

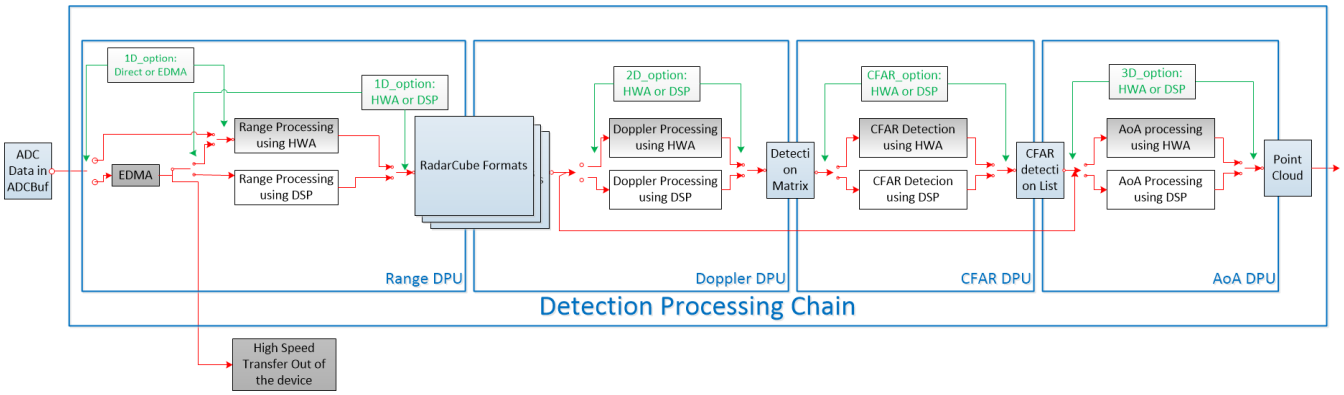


Figure 10: Scalable data processing chain using mmWave SDK

Please refer to the code and documentation inside the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw folder for more details and example code on how this chain is realized using mmWave SDK components.

5. 3. Typical Programming Sequence

The above processing chain can be split into two distinct blocks: RF control path and data path.

5. 3. 1. RF Control Path

The control path in the above processing chain is depicted by the following blocks.

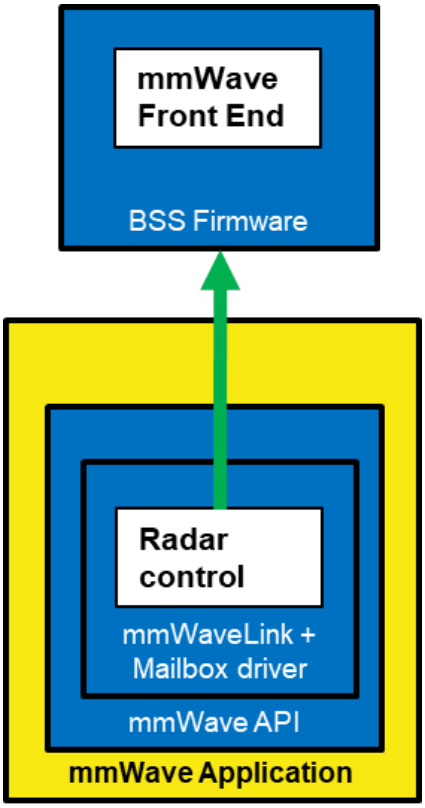


Figure 11: Typical mmWave radar control flow

Following set of figures shows how an application programming sequence would look like for setting up the typical control path - init, config, start. This is a high level diagram simplified to highlight the main software APIs and may not show all the processing elements and call flow. For an example implementation of this call flow, please refer to the code and documentation inside the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw folder.

5. 3. 1. 1. Single RF Control (MSSRADARSS or DSSRADARSS)

In this scenario, the RF control path runs on either Master subsystem (Cortex-R4F) or DSP subsytem (C674x) and the application can simply call the mmwave APIs in the SDK in isolation mode to realize most of the functionality.

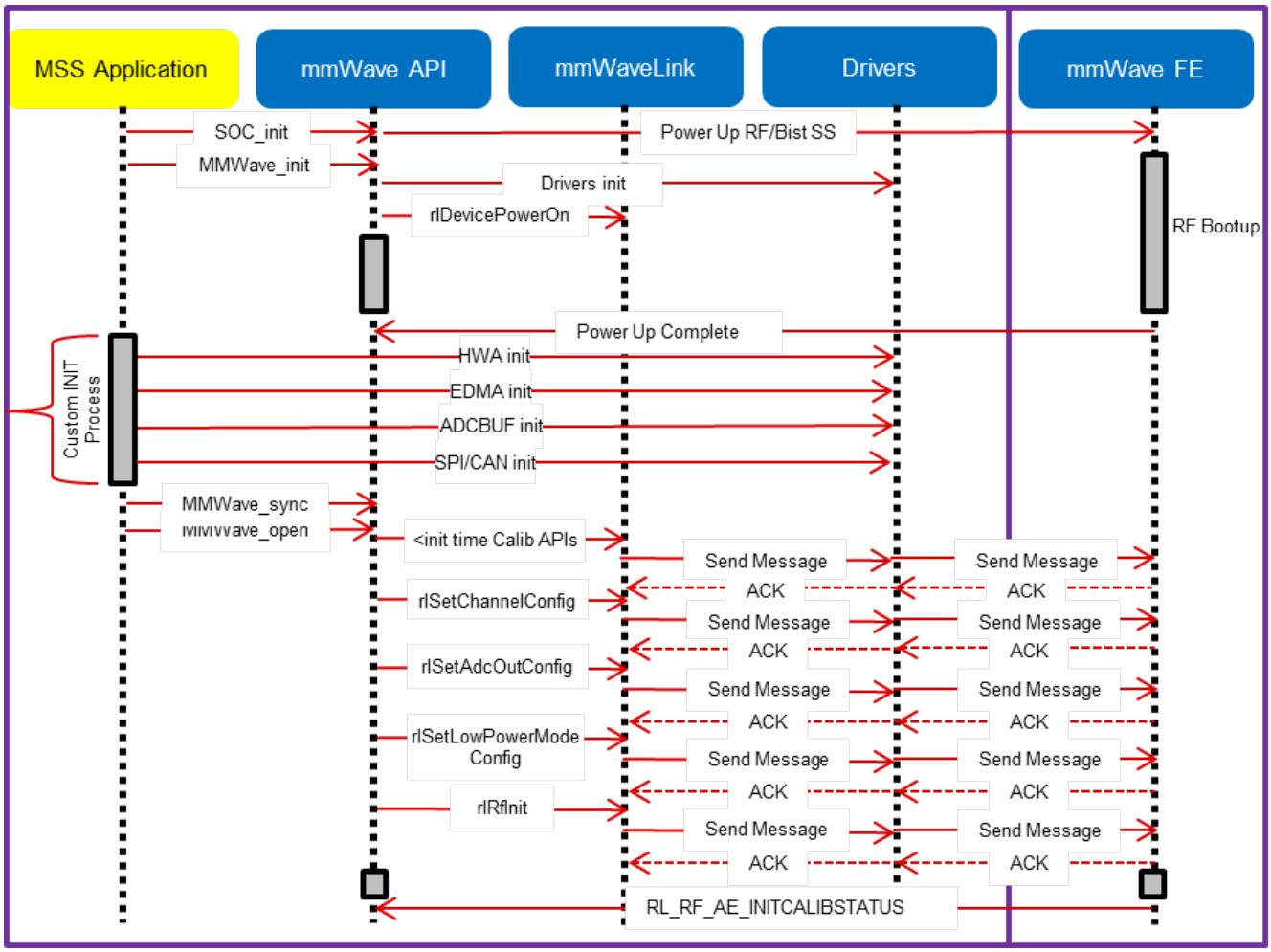


Figure 12: mmWave Isolation mode: Detailed Control Flow (Init sequence)

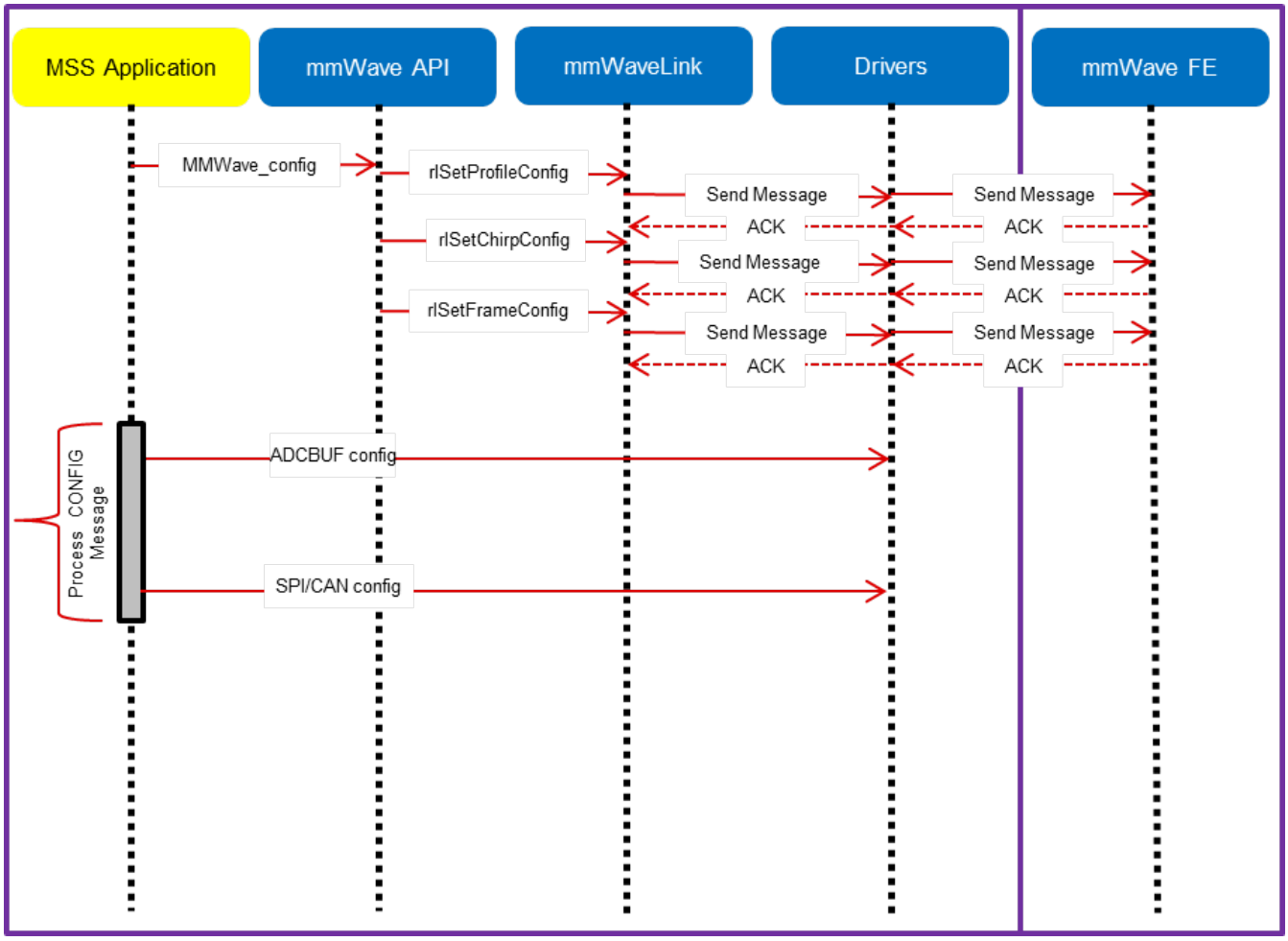


Figure 13: mmWave Isolation mode: Detailed Control Flow (Config sequence)

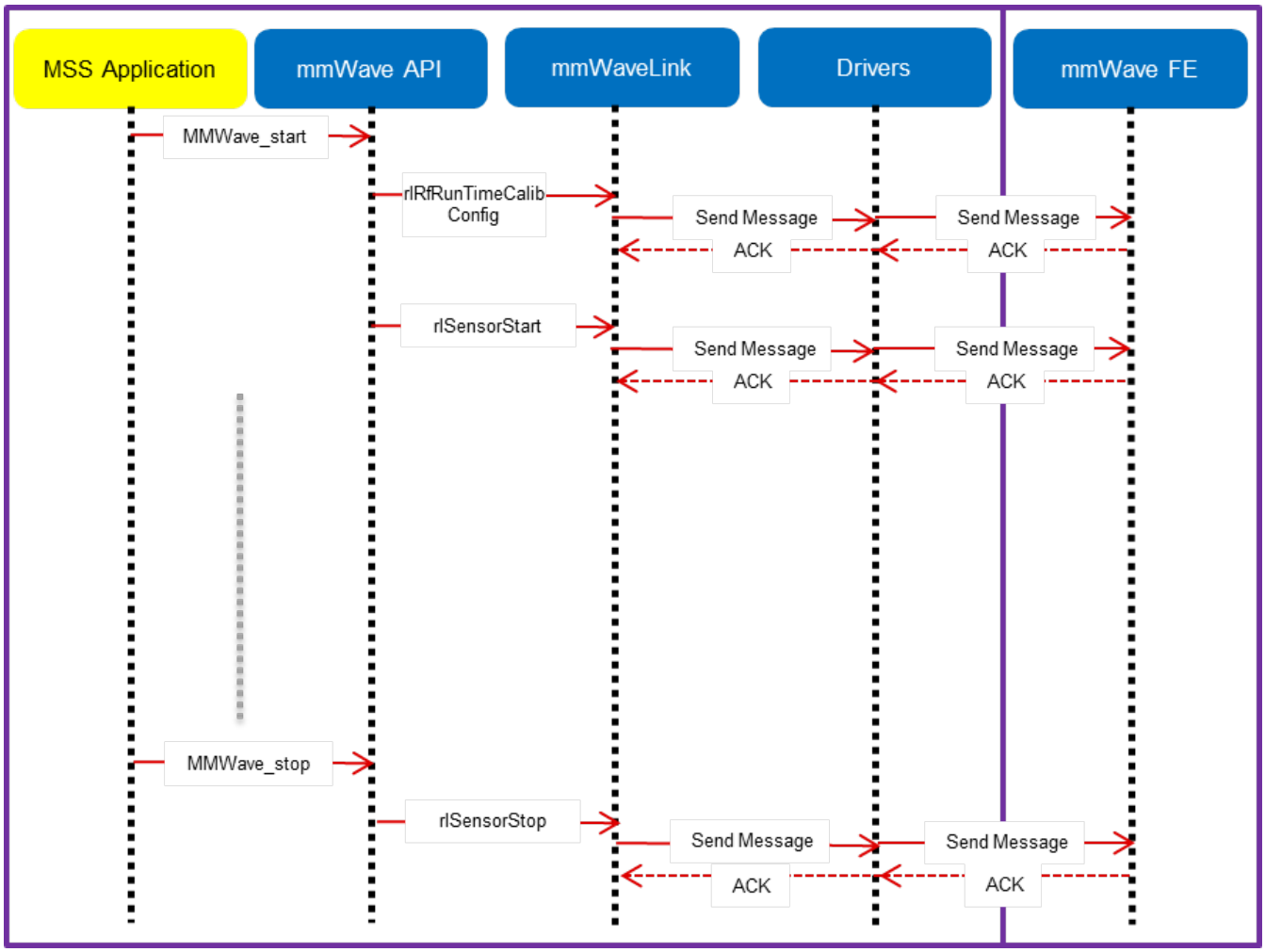


Figure 14: mmWave Isolation mode: Detailed Control Flow (start sequence)

5. 3. 1. 2. Co-operative RF control ((MSS+DSS)<->RADARSS)

In this scenario the control path can runs in "co-operative" mode where RF control APIs can be interchangeably called by either domains (but the sequence of API needs to be maintained). One such deployment could have the RF init and config initiated by the MSS and the start is initiated by the DSS after the data path configuration is complete. In the figures below, control path runs on MSS entirely and MSS is responsible for properly configuring the RADARSS (RF) and DSS (data processing). The mmWave unit tests provide a sample implementation of this co-operative mode.

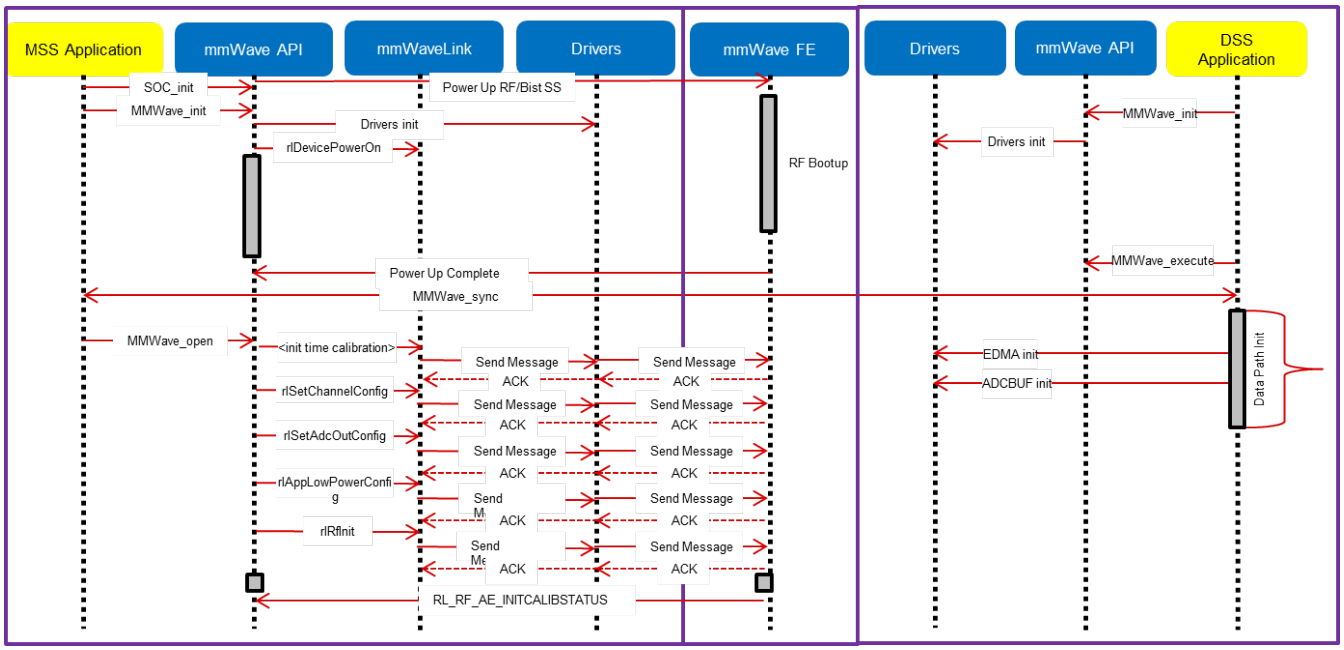


Figure 15: mmWave Co-operative Mode: Detailed Control Flow (Init sequence)

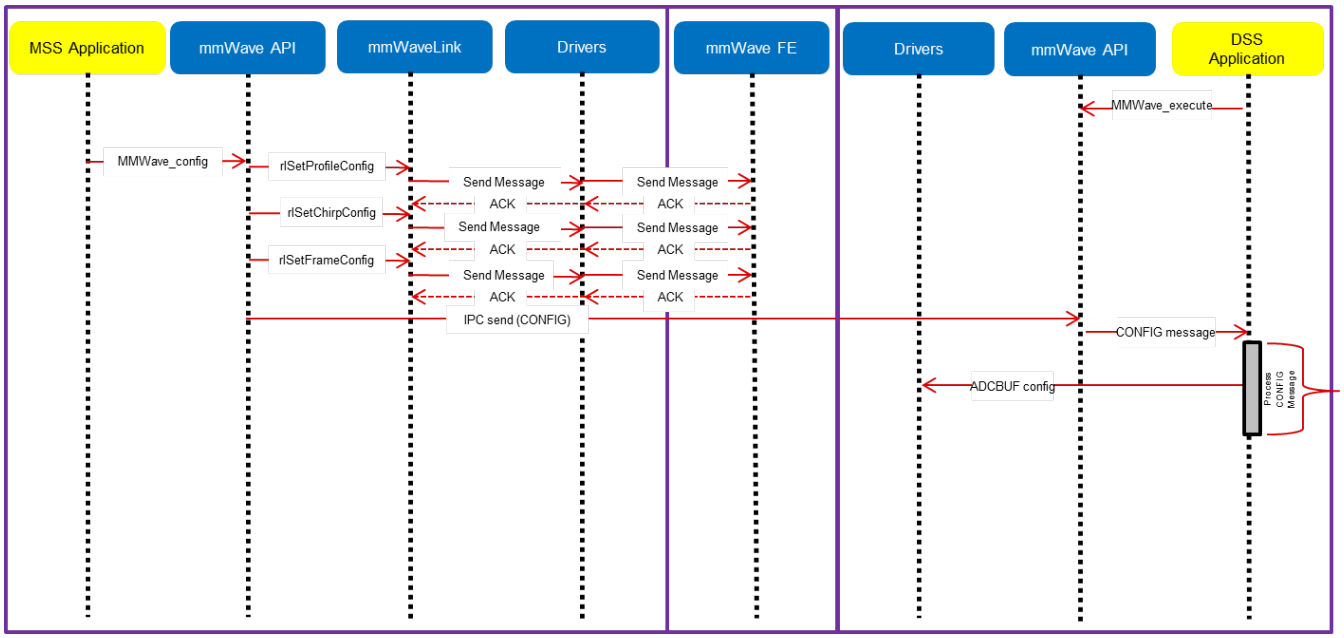


Figure 16: mmWave Co-operative Mode: Detailed Control Flow (Config sequence)

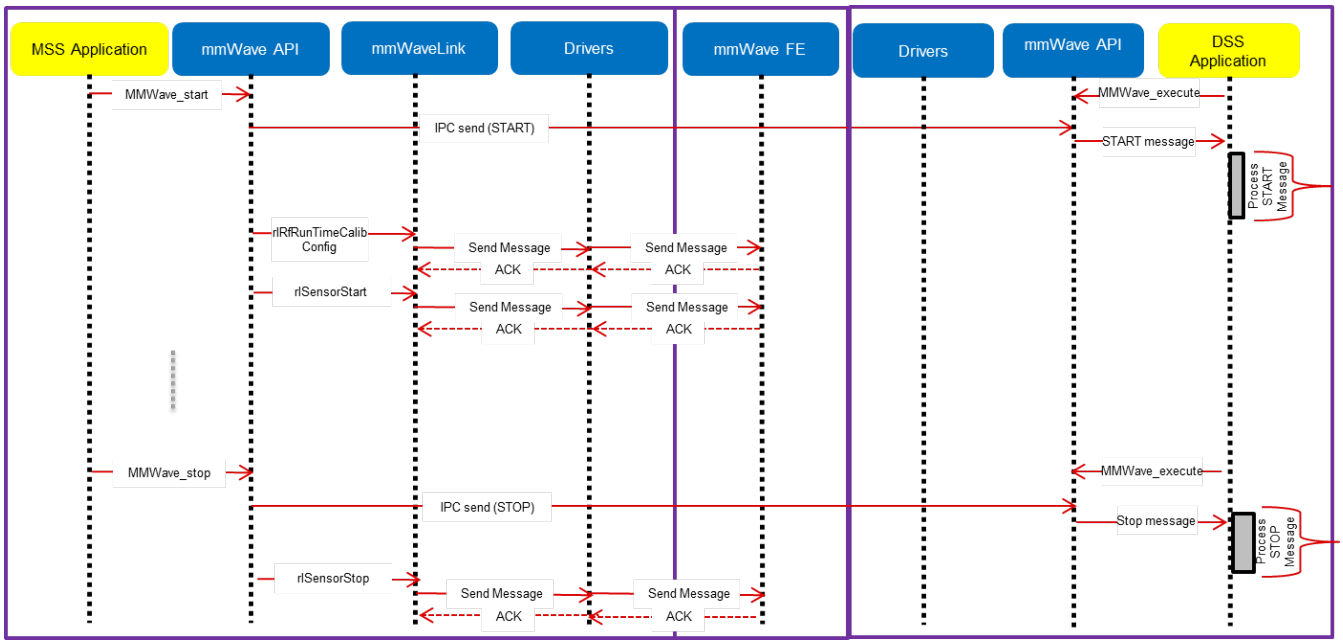


Figure 17: mmWave Co-operative Mode: Detailed Control Flow (Start sequence)

5. 3. 2. Data Path

The mmwave detection processing can be split into following layers of application code, control/management layer to manipulate the data processing elements, processing chain that ties up individual modules to create a data flow and the low level data processing modules and interfaces.

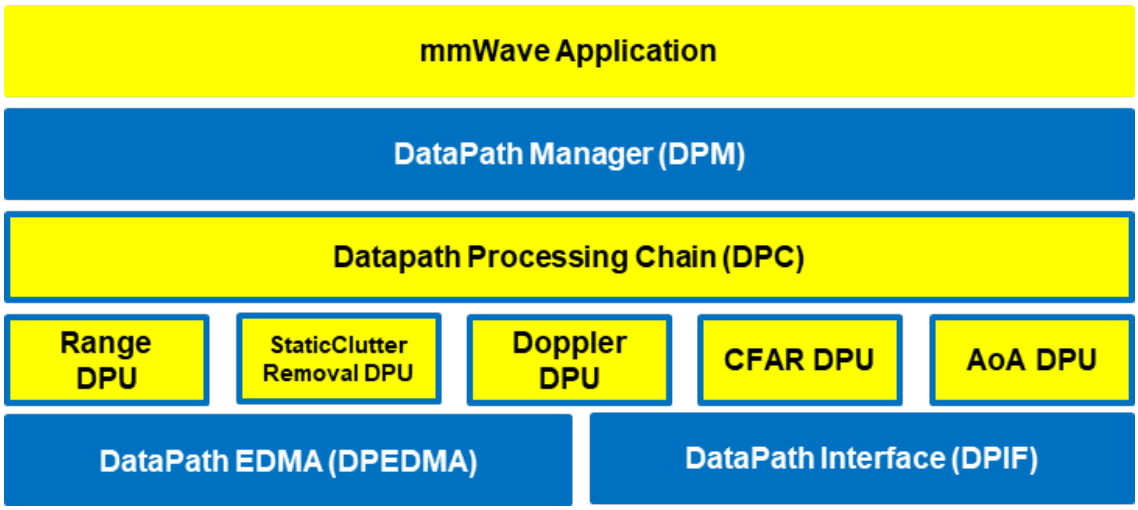


Figure 18: Typical mmWave Detection Processing Layers

mmWave devices present a few options on how the data processing layers can be realized using the various control/processing nodes within the device. To allow ease of programming across these deployment types, data path manager (DPM) presents a simplified API structure to the application while hiding the complexity of inter task and inter processor communications. As can be seen from the following figures, application would just need to call the various DPM APIs to control the processing chain (seen as function calls in 'blue' in the ladder diagrams below) and re-act to the outcome of these APIs in the report callback. Data processing chains (DPCs) also present a standardized API structure to the application via DPM and encapsulate the realization of the data flow using data processing units (DPUs) within while presenting simple IOCTL based interface to configure and trigger the data flow. Based on the usecase and the mmWave device hardware capabilities, application can choose from one of the following deployments:

- DPC runs on the same core as control core and the application can control the DPC via DPM in local mode. (See local domain config and processing figures below)

- DPC runs on another core which is different from the controlling core and the application can control the DPC via DPM in remote mode. (See remote domain config and processing figures below)
- DPC is split between two cores and the application can control the DPC via DPM in distributed mode. (See distributed domain config and processing figures below)

The following ladder diagrams show the flow for init, two different forms of config (one initiated on local core and other on remote core), start trigger, chirps/frame events and stop trigger. The choice of MSS and DSS responsibilities are shown as one of the possible examples - their roles can be interchanged as per application needs. These ladder diagrams dont show the corresponding MMWAVE/RF control calls to show independence between RF control and datapath control. Having said that, typical application would follow the following flow for these two form of controls:

- mmWave init and DPM init (order doesn't matter)
- mmWave config and DPM IOCTL for DPC config (order doesn't matter)
- DPM start and then mmWave start (note this is recommended as DPC should be in started state before the real time frame/chirp H /W events occur due to mmWave start)
- mmWave stop and then DPM start (note this is recommended as DPC should be stopped after the real time frame/chirp H/W events stop due to mmWave stop)

5. 3. 2. 1. Data processing flow with local domain control

In this deployment, the core (MSS or DSS) that runs the actual data processing chain (DPC) also controls it. Application calls DPM APIs for init, data processing IOCTL for configuration, start and stop. DPM reports back status from DPC using the application registered report callback function. Application provides an execution context (task) for the DPM/DPC to run. DPC provides back the processing results (point cloud, tracked objects, etc) to the application in this execution context.

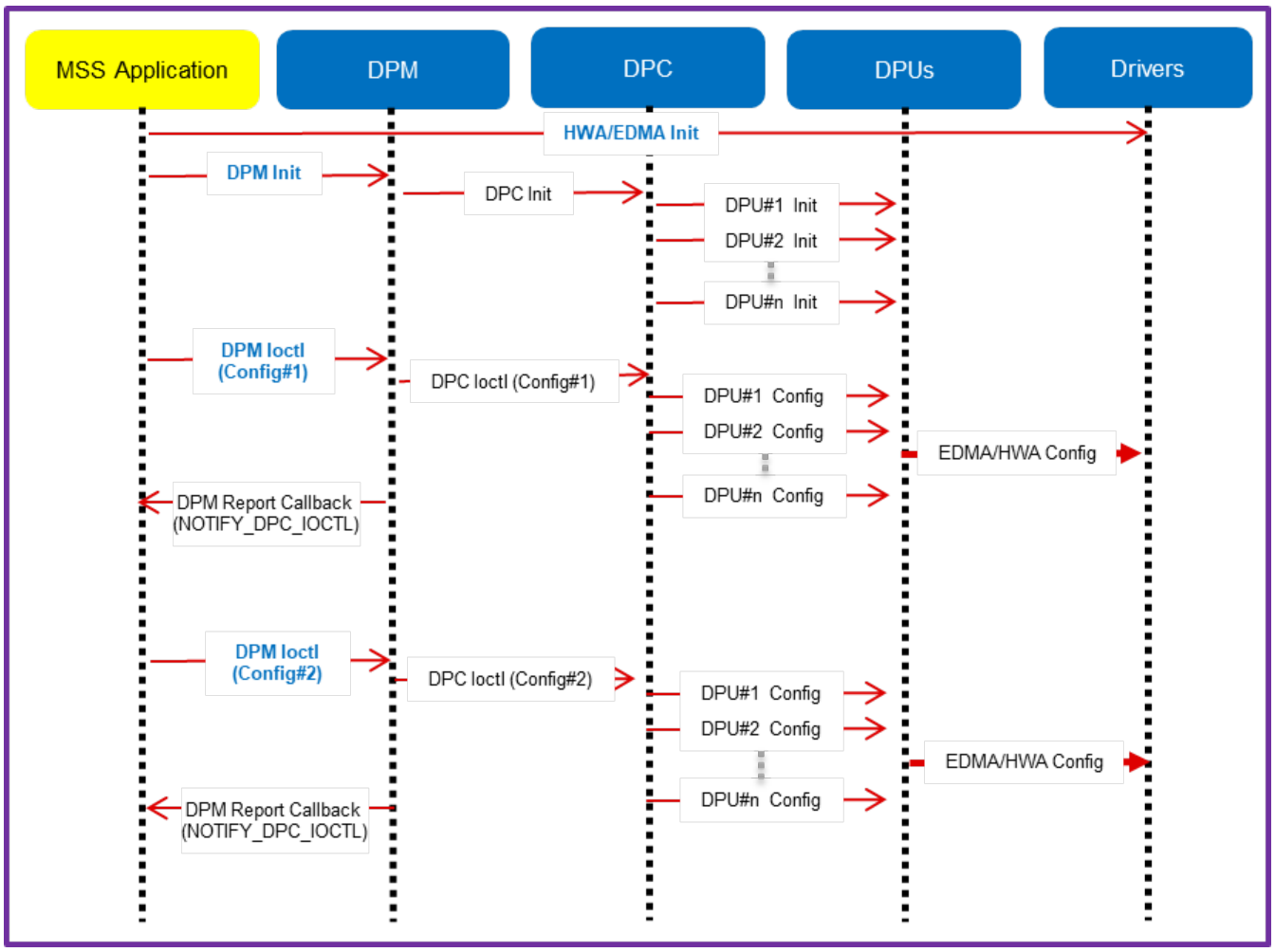


Figure 19: Data processing flow with local domain control (init/config)

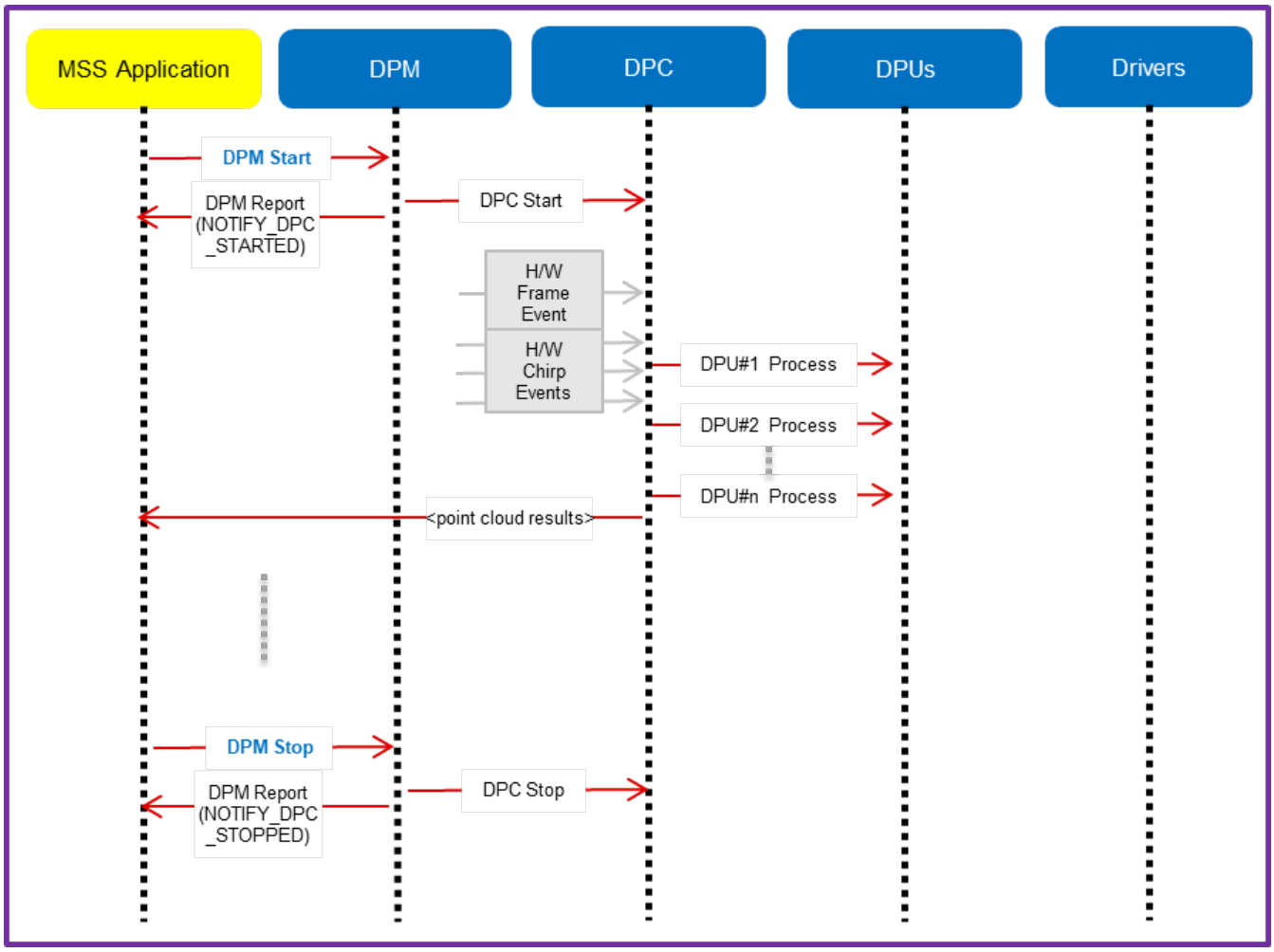


Figure 20: Data processing flow with local domain control (start/chirp/frame/stop)

5. 3. 2. 2. Data processing flow with remote domain control

In this deployment, the data processing chain runs on a chosen data core while the control for it exists on the other core. Application code on control core and data core calls DPM APIs for init and sync'ing with each other. The control core calls data processing IOCTL for configuration, start and stop APIs. The H/W events are received on the data core. DPM reports back status from DPC using the application registered report callback function on both control and data cores. DPC provides back the processing results (point cloud, tracked objects, etc) to the data core application code which can send the result to the control core using DPM send result API.

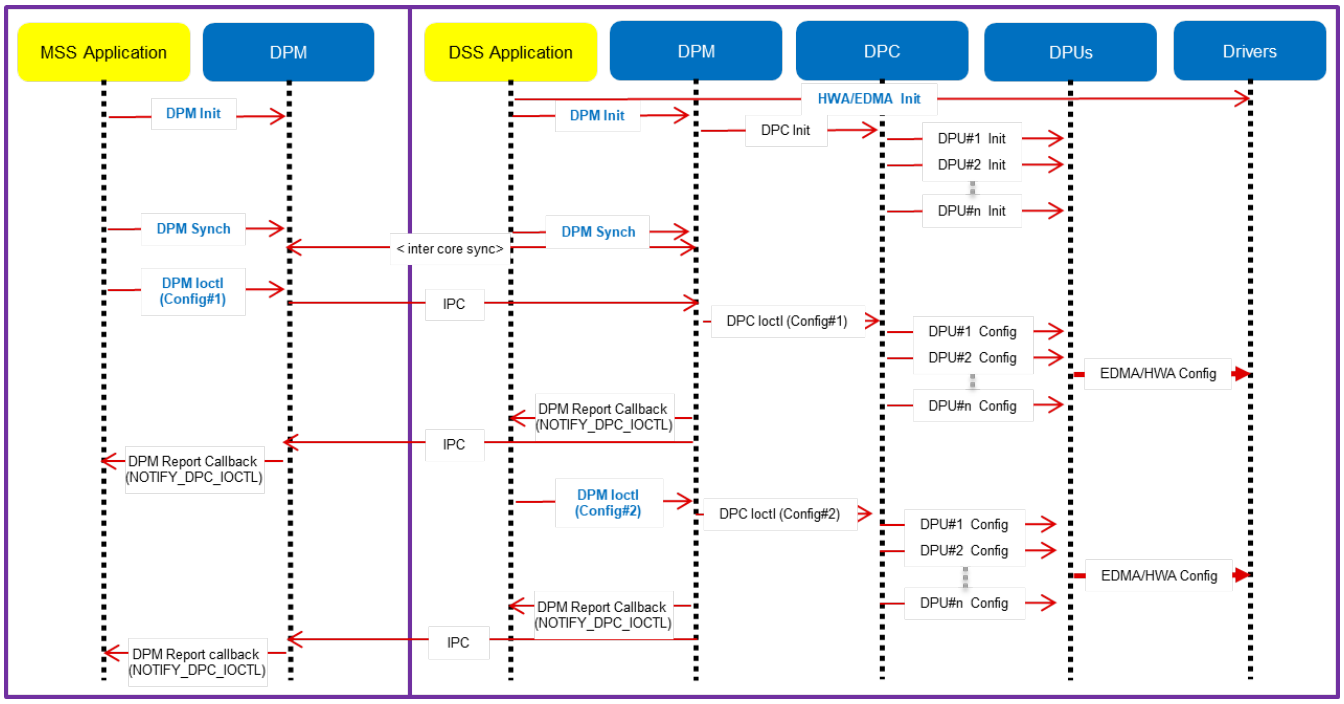


Figure 21: Data processing flow with remote domain control (init/config)

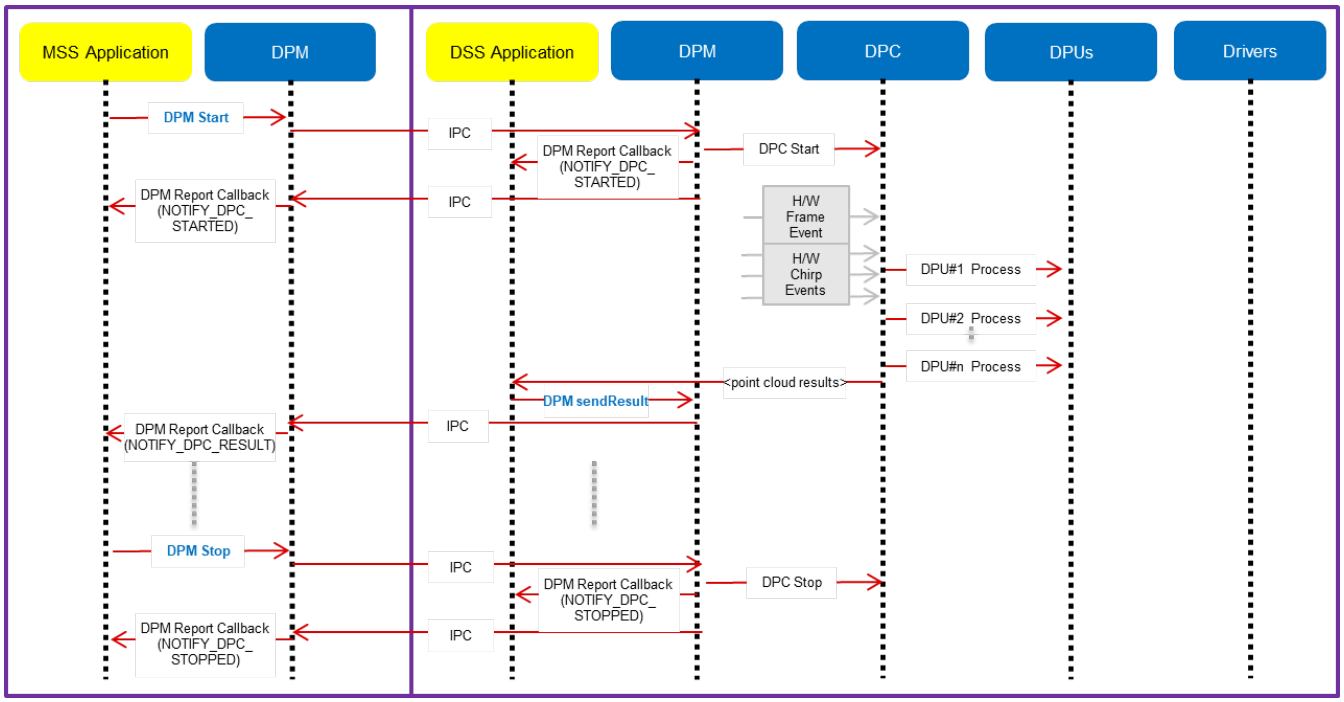


Figure 22: Data processing flow with remote domain control (start/chirp/frame/stop)

5. 3. 2. 3. Distributed Data processing flow and control

In this deployment, the data processing chain is split across cores along with the control. Application code on both cores call DPM APIs for init and sync'ing with each other. Either core can call data processing IOCTL for configuration, start and stop APIs. DPM reports back status from DPC using the application registered report callback function on both cores. Partial results from the DPC running on one core can be

passed onto the DPC running on other core using the DPM relay result API. DPC can provide back the final processing results (point cloud, tracked objects, etc) to the same core's application code which can then send the result to the application running on other core using DPM send result API. Following ladder diagrams shows just one of the many ways of splitting the DPC across two cores.

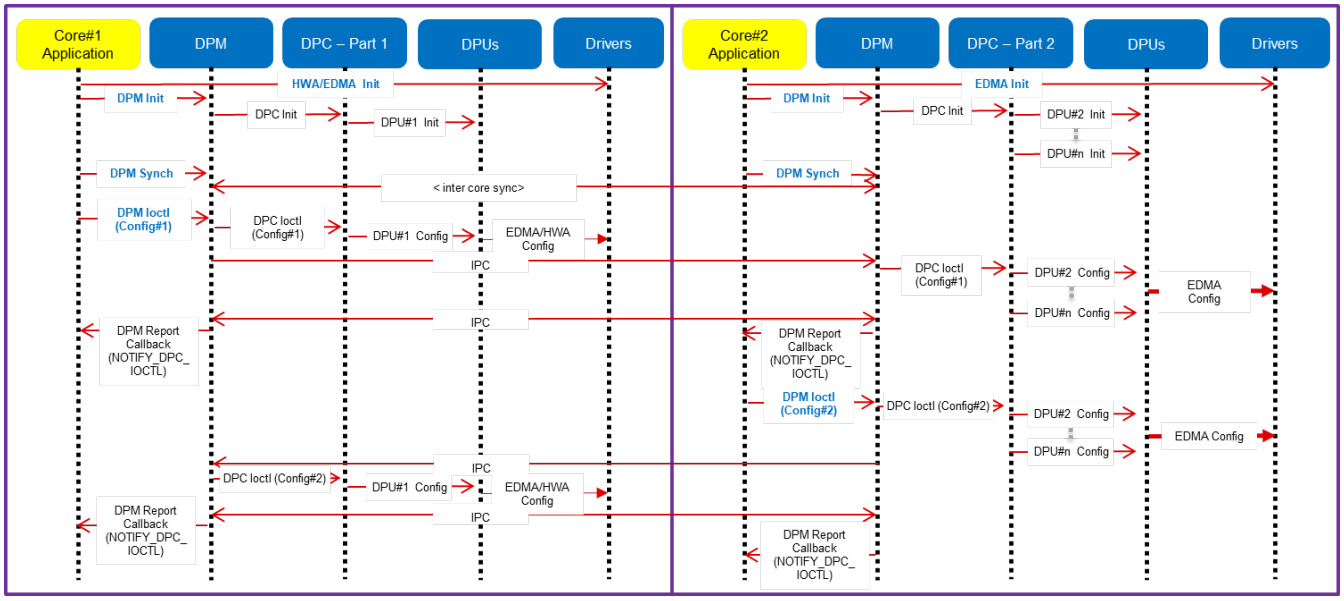


Figure 23: Distributed Data processing flow and control (init/config)

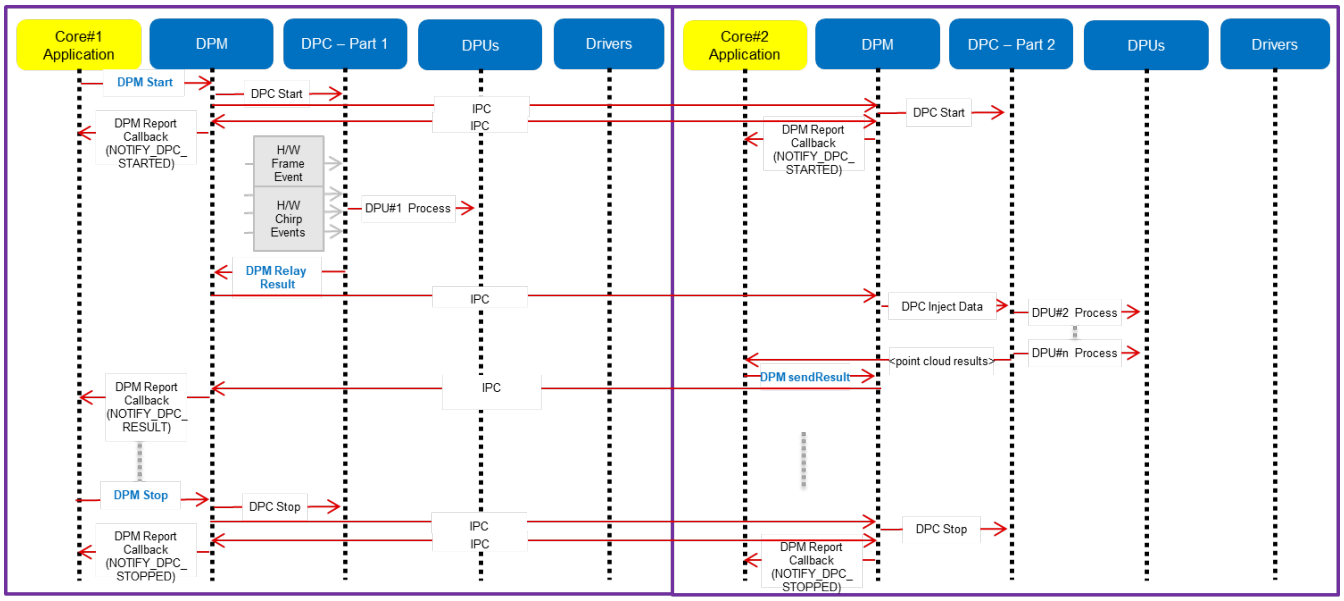


Figure 24: Distributed Data processing flow and control (start/chirp/frame/stop)

5. 4. mmWave SDK - TI components

The mmWave SDK functionality broken down into components are explained in next few subsections.

5. 4. 1. Drivers

Drivers encapsulate the functionality of the various hardware IPs in the system and provide a well defined API to the higher layers. The drivers are designed to be OS-agnostic via the use of OSAL layer. Following figure shows typical internal software blocks present in the SDK drivers. The source code for the SDK drivers are present in the mmwave_sdk_<ver>\packages\ti\drivers\<ip> folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\drivers\<ip>\docs\doxygen\html\index.html . The driver's unit test

code, running on top of SYSBIOS is also provided as part of the package mmwave_sdk_<ver>\packages\ti\drivers\<ip>\test\. The library for the drivers are placed in the mmwave_sdk_<ver>\packages\ti\drivers\<ip>\lib directory and the file is named lib<ip>_<platform>.aer4f for MSS and lib<ip>_<platform>.ae 674 for DSP.

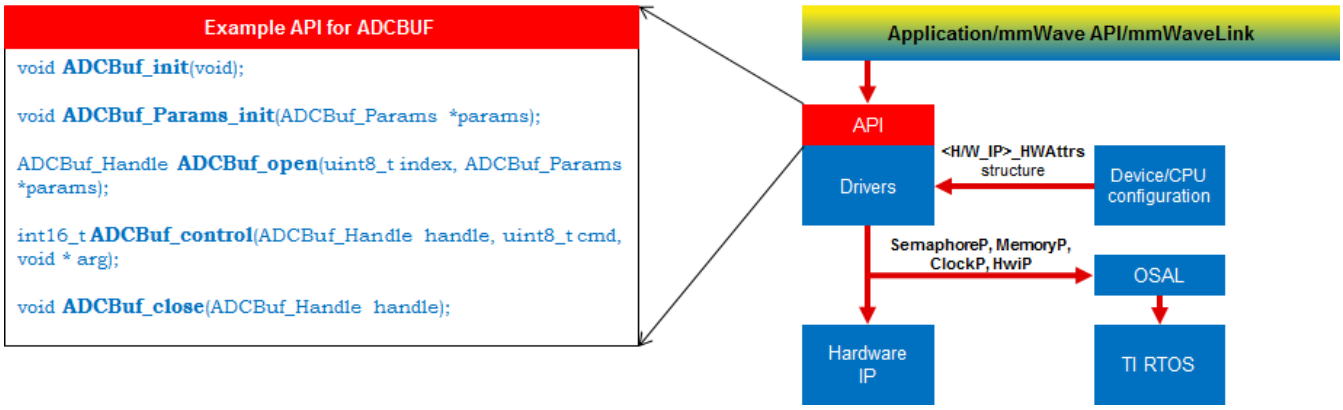


Figure 25: mmWave SDK Drivers - Internal software design

Drivers /Hardware IP	Platform supported for R4F target	Platform supported for DSP C674x target ('all' here excludes xwr14xx)	Driver Functionality Implemented in mmWave SDK
ADCBUF	all	all	All features of IP (ADCBUF, CQ) are implemented in the driver
CAN	all except iwr68xx	-	Following features of IP are implemented in the driver: Configure Rx and Tx I/O Control registers Configure DCAN mode of operation Configure DCAN controller, interrupts, ECC, parity Set bit time parameters Configure Rx and Tx message objects Receive and Transmit a CAN message Retrieve Tx message object transmission status and Rx message object reception status Check the validity of the received message
CANFD	awr16xx awr18xx iwr68xx	-	Following features of IP are implemented in the driver: • Reset MCAN driver • Initialize MCAN clock stop controls, auto wakeup, MCAN mode - classic versus FD mode, transceiver delay conpensation • Configure MCAN controller and global filters • Configure MCAN mode of operation • Set bit time parameters • Configure message filters, Rx/Tx FIFOs • Add and cancel Tx requests • Transmits a CAN message • Receive a CAN message • Check the validity of the received message • Configure interrupt multiplexer to service message objects • Retrieve interrupt line status, interrupt pending status, parity error status, bit error status, ECC diagnostics status, ECC error status and MCAN error status • Clear interrupt pending status, ECC diagnostics error status, ECC error status and MCAN error status • Configure MCAN parity function, self test mode, ECC Diagnostic mode
CBUFF	all	all	Following features of IP are implemented in the driver: Supports Platform defined HSI: LVDS or CSI (IWR14xx only). Initialize and setup the CBUFF Driver Configure the Linked List and EDMA Channels to support the data transfer Supports Interleaved and Non-Interleaved data mode Supports the data formats: ADC, ADC_CP, CP_ADC, CP_ADC_CQ Supports CRC
CBUFF (LVDS)	all	all	Following features of IP are implemented in the driver: LVDS driver supports the chirp and continuous mode of data transmission. Supports only 2 and 4 lane configuration in Format0. Supports transfer of S/W triggered user data over CBUFF/LVDS interface

CRC	all	all	All features of IP are implemented in the driver including: CRC-16 CRC-32 CRC-64
CRYPTO	xwr16xx (HS) awr18xx (HS)	-	The driver supports following AES mode of encryption: ■ Electronic codebook mode (ECB) ■ Cipher-block chaining mode (CBC) ■ Cipher feedback mode (CFB) ■ Counter mode (CTR) ■ Integer counter mode (ICM) ■ Galios/counter mode (GCM) ■ Counter with CBC-MAC mode (CCM) The driver supports following HMAC modes: ■ MD5 ■ SHA-1 ■ SHA-224 ■ SHA-256
CSI-2	iwr14xx	-	Following features of IP are implemented in the driver: Initialization and Setup of the Protocol Engine Initialization and configuration of the DSI PHY DSI Phy Parameters can be customized by the application
DMA	all	-	Following features of IP are implemented in the driver: software and hardware triggered transfer frame based transfer block based transfer Addressing mode (Constant, Indexed, Post Increment) FTC, BTC, LFS, HBC interrupts channel chaining auto-initiation mode interrupt based and polling based channel completion APIs
EDMA	all	all	All features of IP are implemented in the driver except "privilege" feature
ESM	all	all	Default ESM FIQ Interrupt handler for R4F and hook function for DSP's NMI Provide application to register callback functions on specific ESM errors.
GPIO	all	-	All features of IP are implemented in the driver
HWA	all except xwr16xx	all except xwr16xx	All features of IP are implemented in the driver
I2C	all	-	All features of IP are implemented in the driver
MAILBOX	all	all	All features of IP are implemented in the driver.
OSAL	all	all	Provides an abstraction layer for some of the common OS services: Semaphore Mutex Debug Interrupts Clock Memory
PINMUX	all	-	All Pinmux fields can be set and all device pad definitions are available
QSPI	all	-	All features of IP are implemented in the driver.
QSPIFLASH	all	-	All features of IP are implemented in the driver.
RTI	all	all	Part of TI RTOS offering
SOC	all	all	Provides abstracted APIs for system-level initialization. See section "mmWave SDK - System Initialization" for more details.
SPI (MIBSPI)	all	-	All features of IP are implemented in the driver including: 4-wire Slave and master mode 3-wire Slave and Master mode both polling mode and DMA mode for read/write char length 8-bit and 16-bit.
VIM	all	-	Part of TI RTOS offering
UART	all	all	All features of IP are implemented in the driver including: Standard Baud Rates: 9600, 14400, 19200 till 921600 Data Bits: 7 and 8 Bits Parity: None, Odd and Even Stop Bits: 1 and 2 bits

			Blocking and Polling API for reading and writing to the UART instance DMA support for read/write APIs
WATCHDOG	all	all	All features of IP are implemented in the driver.

Table 2: Supported drivers and their functionality

5. 4. 2. OSAL

An OSAL layer is present within the mmWave SDK to provide the OS-agnostic feature of the foundational components (drivers, mmWaveLink, mmWaveAPI). This OSAL provides an abstraction layer for some of the common OS services: Semaphore, Mutex, Debug, Interrupts, Clock, Memory, CycleProfiler. The source code for the OSAL layer is present in the mmwave_sdk_<ver>\packages\ti\drivers\osal folder. Documentation of the APIs are available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\drivers\osal\docs\doxygen\html\index.html. A sample porting of this OSAL for TI RTOS is provided as part of the mmWave SDK. System integrators could port the OSAL for their custom OS or customize the same TI RTOS port for their custom application, as per their requirements.

Examples of what integrators may want to customize:

- MemoryP module - for example, choosing from among a variety of heaps available in TI RTOS (SYSBIOS), or use own allocator.
- Hardware interrupt mappings. This case is more pronounced for the C674 DSP which has only 16 interrupts (of which 12 are available under user control) whereas the events in the SOC are much more than 16. These events go to the C674 through an interrupt controller (INTC) and Event Combiner (for more information see the C674x megamodule user guide at http://www.ti.com/lit/ug/sprufk5a/sprufk5a.pdf). The default OSAL implementation provided in the release routes all events used by the drivers through the event combiner. If a user chooses to route differently (e.g for performance reasons), they may add conditional code in OSAL implementation to route specific events through the INTC and event combiner blocks. User can conveniently use event defines in ti/common/sys_common_*.h to acheive this.

5. 4. 3. mmWaveLink

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 Master subsystem (MSS R4F) and/or DSP subsystem (DSS C674x). 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. It provides a well defined interface for the application to plug in the correct communication driver APIs to communicate with the RADARSS. it acts as driver for Radar SS and exposes services of Radar SS. It includes APIs to configure HW blocks of Radar SS and provides communication protocol for message transfer between MSS/DSS and RADAR SS.

- Link between application and Radar SS
- Handles communication errors, Notifies exceptions
- Platform and OS independent
- Can work in single threaded (non OS) environment

Following figure shows the various interfaces/APIs of the mmWaveLink component. The source code for mmWaveLink is present in the mmwave_sdk_<ver>\packages\ti\control\mmwavelink folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\control\mmwavelink\docs\doxygen\html\index.html . The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\control\mmwavelink\test\ .

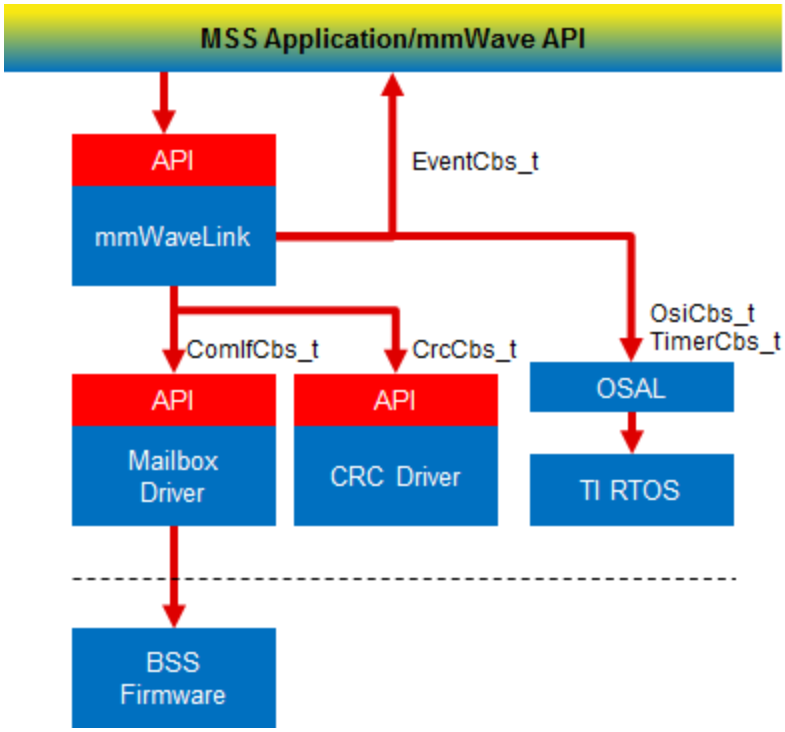


Figure 26: mmWaveLink - Internal software design

5. 4. 4. mmWave API

mmWaveAPI is a higher layer control running on top of mmWaveLink and LLD API (drivers API). It is designed to provide a layer of abstraction in the form of simpler and fewer set of APIs for application to perform the task of mmWave radar sensing. In mmwave devices with dual cores, it also provides a layer of abstraction over IPC to synchronize and pass configuration between the MSS and DSS domains. The source code for mmWave API layer is present in the mmwave_sdk_<ver>\packages\ti\control\mmwave folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\control\mmwave\docs\doxygen\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\control\mmwave\test\.

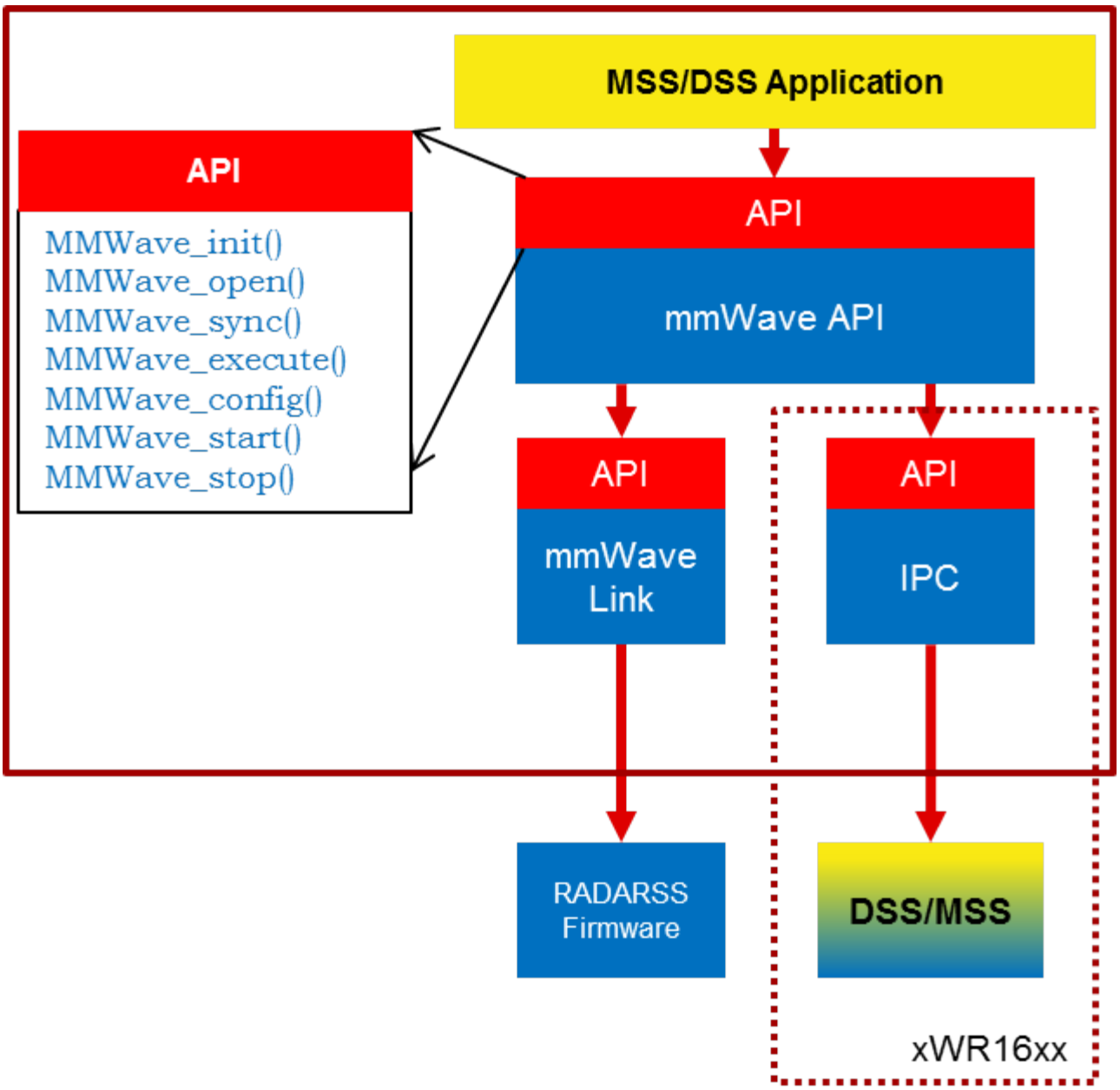


Figure 27: mmWave API - Internal software design

There are two modes of configurations which are provided by the mmWave module.

5. 4. 4. 1. Full configuration

The "full" configuration mode implements the basic chirp/frame sequence of mmWave Front end and is the recommended mode for application to use when using the basic chirp/frame configuration. In this mode the application will use the entire set of services provided by the mmWave control module. These features includes:-

- Initialization of the mmWave Link
- Synchronization services between the MSS and DSS
- Asynchronous Event Management
- Start & Stop services
- Configuration of the RADARSS for Frame, advanced frame & Continuous mode
- Configuration synchronization between the MSS and DSS

In the full configuration mode; it is possible to create multiple profiles with multiple chirps. The following APIs have been added for this purpose:-

Chirp Management:

- MMWave_addChirp
- MMWave_delChirp

Profile Management:

- MMWave_addProfile
- MMWave_delProfile

5. 4. 4. 2. Minimal configuration

For advanced users, that either need to use advanced frame config of mmWave Front End or need to perform extra sequence of commands in the CONFIG routine, the minimal mode is recommended. In this mode the application has access to only a subset of services provided by the mmWave control module. These features includes:-

- Initialization of the mmWave Link
- Synchronization services between the MSS and DSS on the dual core devices
- Asynchronous Event Management
- Start & Stop services

In this mode the application is responsible for directly invoking the mmWave Link API in the correct order as per their configuration requirements. The configuration services are not available to the application; so in mmwave devices with multiple cores (ex: xwr16xx, iwr68xx, etc), the application is responsible for passing the configuration between the MSS and DSS if required.

See sample call flow below:

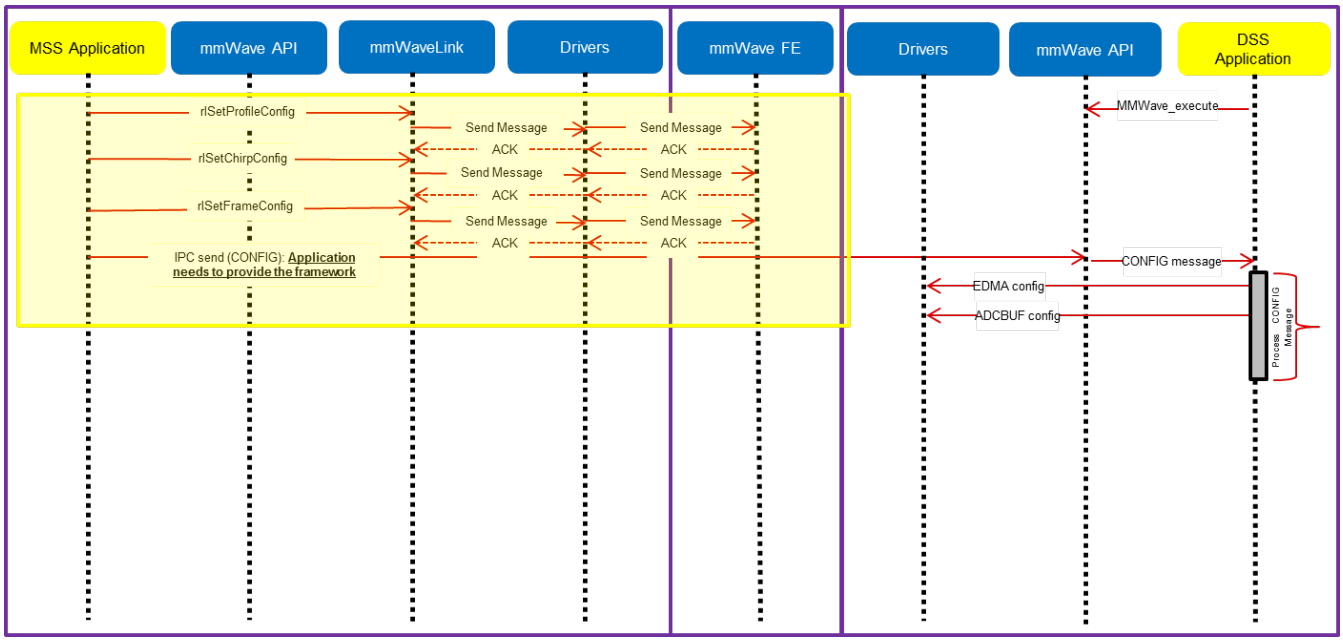


Figure 28: mmWave API - 'Minimal' Config - Sample flow (mmWave devices with MSS and DSS cores and module in co-operative mode)

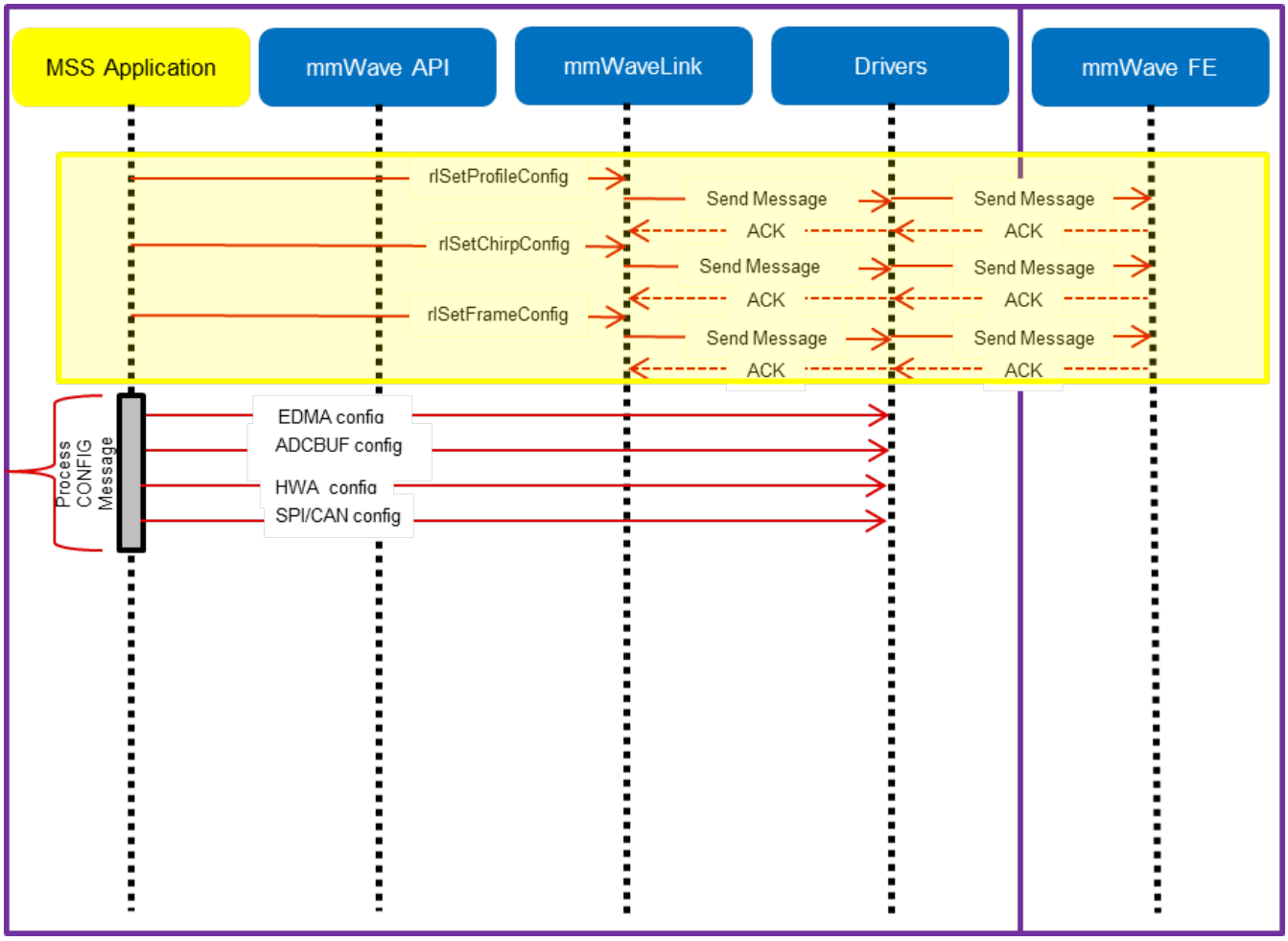


Figure 29: mmWave API - 'Minimal' Config - Sample flow (mmWave devices with single core or when module is used in isolation mode)

mmWave Front End Calibrations

mmWave API, by default, enables all init/boot time time calibrations for mmWave Front End. There is a provision for user to provide custom calibration mask in MMWave_open API and/or to provide a buffer that has pre-stored calibration data.

When application requests the one-time and periodic calibrations in MMWave_start API call, mmWave API enables all the available one-time and periodic calibrations for mmWave Front End.

mmWave API doesn't expose the mmwavelink's LDO bypass API (rlRfSetLdoBypassConfig/rlRfLdoBypassCfg_t) via any of its API. If this functionality is needed by the application (either because of the voltage of RF supply used on the TI EVM/custom board or from monitoring point of view), user should refer to mmwavelink doxygen (mmwave_sdk_<ver>\packages\ti\control\mmwavelink\docs\doxygen\html\index.html) on the usage of this API and call this API from their application before calling MMWave_open().

mmWave_open

Although mmWave_close API is provided, it is recommended to perform mmWave_open only once per power-cycle of the sensor.

5. 4. 5. Datapath Interface (DPIF)

DPIF defines the standard interface points in the detection processing chain that will correspond to the "blue" boxes in the scalable chain shown in the figure above. Key interfaces defined in this layer are:

- Input ADC data (contents of ADCbuf memory)
- Radar Cube

- Detection Matrix
- Point cloud and its side info

The source code for DPIF is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpif folder.

5. 4. 6. Data Processing Units (DPUs)

Data Translating function(s) from one interface point to the other are called “Data Processing Units”. Splitting the data processing chain into processing units promote re-use of certain processing blocks across multiple chains.

- Range Processing (ADC data to Radar Cube): This processing unit performs (1D FFT+ optional DC Range Calib) processing on the chirp (RF) data during the active frame time and produces the processed data in the radarCube. This processing unit is standardized in the mmWave SDK. It provides implementation based on both HWA and DSP. HWA based implementation can be instantiated either on R4F or C674x. The source code for Range DPU is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpu\rangeproc folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpu\rangeproc\docs\doxygen\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\ datapath\dpu\rangeproc\ test\ .
- Static Clutter Processing (Radar Cube to Radar Cube): When enabled, this processing unit reads Range FFT out data from the radar cube and performs static clutter removal before writing the data back to the radar cube during the interframe time. This processing unit is offered as reference implementation and users of SDK could either re-use these as is in their application /processing chain or create variations of these units based on their specific needs. It provides S/W based implementation and can be instantiated either on R4F or C674x. The source code for StaticClutter DPU is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\staticclutterproc folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\staticclutterproc\docs\doxygen\html\index.html.
- Doppler Processing (Radar Cube to Detection Matrix): This processing unit performs (2D FFT + Energy Sum) processing on the radar Cube during the inter frame and produced detection matrix. This processing unit is offered as reference implementation and users of SDK could either re-use these as is in their application/processing chain or create variations of these units based on their specific needs. It provides HWA based implementation and can be instantiated either on R4F or C674x. The source code for Doppler DPU is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\dopplerproc folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\dopplerproc\docs\doxygen\html\index.html.
- CFAR + AoA (Detection Matrix to Point Cloud): They are offered as two independent DPUs and collectively run CFAR-CA algorithm, peak grouping, field-of-view filtering, doppler compensation and angle (azimuth+elevation) estimation on the detection matrix during inter frame to produce the point cloud. These processing units are offered as reference implementation and users of SDK could either re-use these as is in their application/processing chain or create variations of these units based on their specific needs. They provide HWA based implementation and can be instantiated either on R4F or C674x. The source code for CFAR-CA DPU is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\cfarcaproc folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\cfarcaproc\docs\doxygen\html\index.html. The source code for AoA DPU is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\aoaproc folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpc\dpu\aoaproc\docs\doxygen\html\index.html.

Each DPU presents the following high level design:

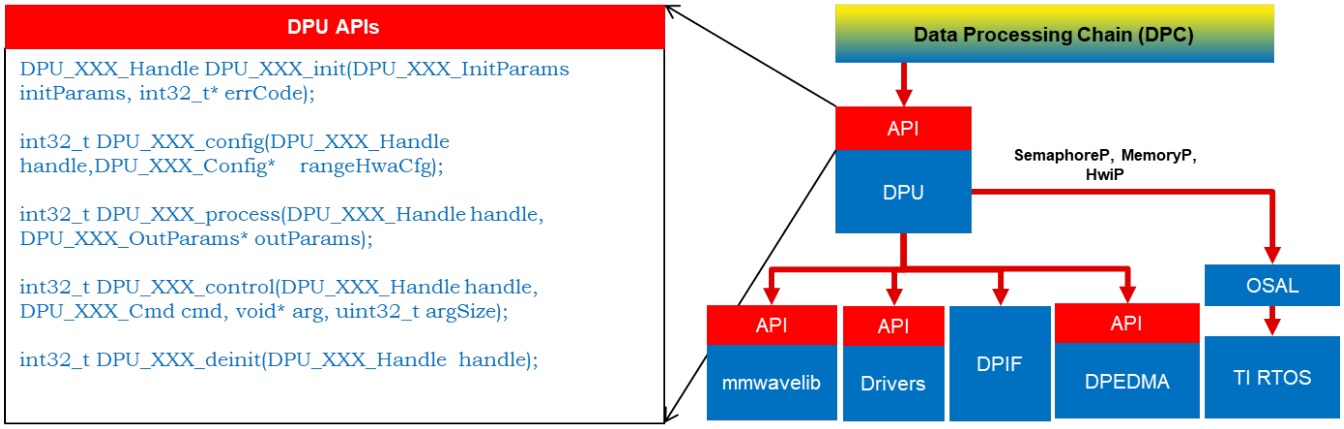


Figure 30: DPU - Internal software design

- All external DPU APIs start with the prefix DPU_. DPU unique name follows next.
  - Ex: DPU_RangeProcHWA_init
- Standard external APIs: init, config, process, ioctl, deinit are provided by each DPU.
  - Init: one time initialization of DPU
  - Config: complete configuration of the DPU: hardware resources, static and dynamic (if supported by DPU)
    - static config: config that is static during ongoing frames
    - dynamic config: config that can be changed from frame to frame but only when process is not ongoing - ideally interframe time after DPC has exported the results for the frame
  - Process: the actual processing function of the DPU

    - Ioctl: control interface that allows higher layer to switch dynamic configuration during interframe time
    - De-init: de-initialization of DPU
- All memory allocations for I/O buffers and scratch buffers are outside the DPU since mmWave applications rely on memory overlay technique for optimization and that is best handled at application level
- All H/W resources must be allocated by application and passed to the DPU. This helps in keeping DPU platform agnostic as well as allows application to share the resources across DPU when DPU processing doesn't overlap in time.
- DPUs are OS agnostic and use OSAL APIs for needed OS services.

A typical call flow for DPUs could be represented as follows. The timing of config and process API calls with respect to chirp/frame would vary depending on the DPU functionality, its usage in the chain, DPC implementation and overlap of hardware resources.

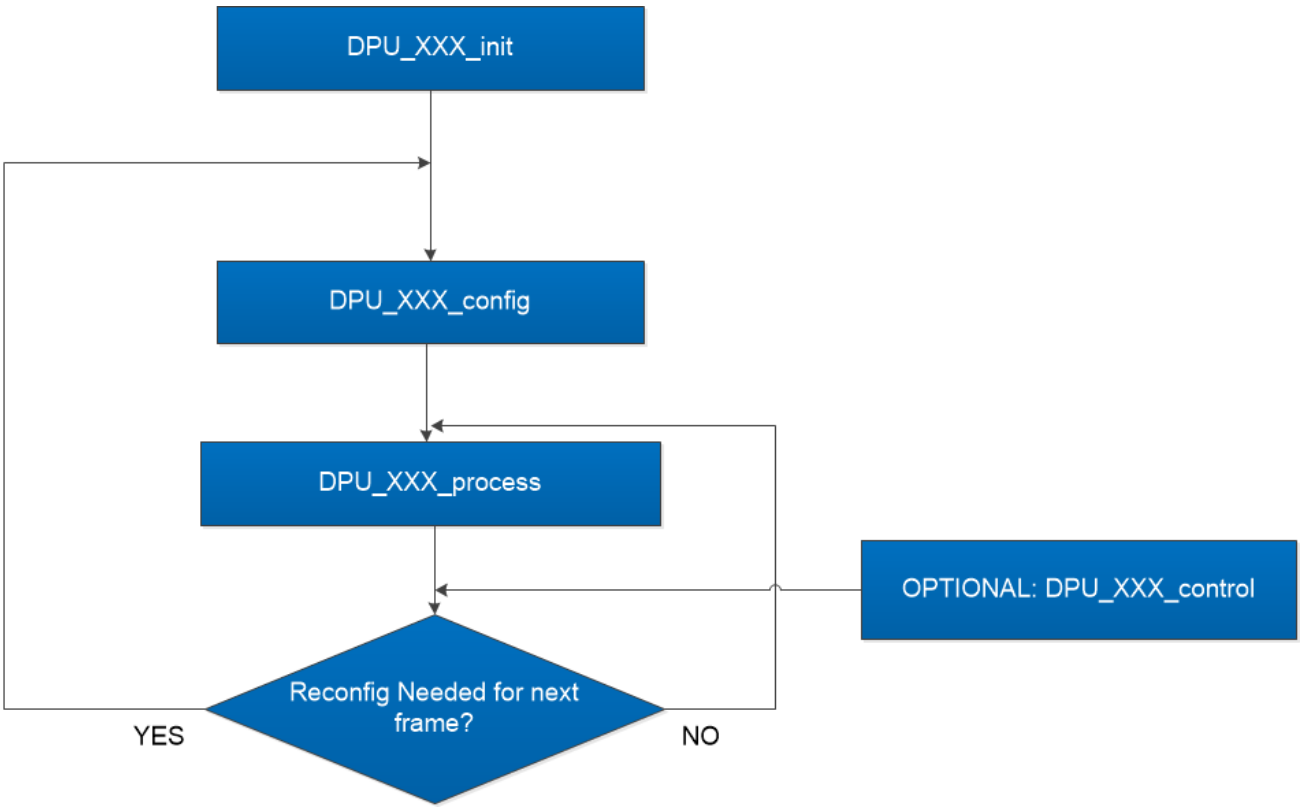


Figure 31: DPU - typical call flow

5. 4. 7. Data Path Manager (DPM)

DPM is the foundation layer that enables the "scalability" aspect of the architecture. This layer absorbs all the messaging complexities (cross core and intra core) and provide standard APIs for integration at the application level and also for integrating any "data processing chain". Application layer will be able to call the DPM APIs from any domain (MSS or DSS) and control the configuration and execution of the "data processing chain". The APIs offered by DPM will be available on both MSS and DSS. The various deployments that it can cater to (but not limited to) are:

- Datapath control on R4F and datapath execution is split between R4F/HWA and DSP (Distributed)
- Datapath control on R4F and datapath execution is on R4F using HWA (Local)
- Datapath control on R4F and datapath execution is on DSP (with and without HWA) (Remote)
- Datapath control on DSP and datapath execution is on DSP+HWA (Local)
- Datapath control on DSP and datapath execution is on DSP (Local)

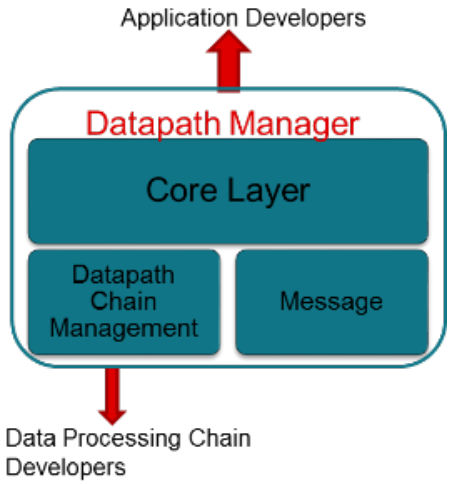


Figure 32: Datapath manager (DPM) - internal software design

The source code for DPM is present in the mmwave_sdk_<ver>\packages\ti\control\dpm folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\ control\dpm \docs\doxygen\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\ control\dpm\test\ .

5. 4. 8. Data processing chain (DPC)

DPC is a separate layer within the datapath that encapsulates all the data processing needs of a mmwave application and provides a well defined interface for integration with the application. In the SDK, there is a reference implementation that corresponds to the generic "object detection" chain which was already a part of the OOB demo in past releases. This chain will conform to the standard DPM dictated API definitions. Internally this layer will use the functionality exposed by Data processing units (DPUs), datapath interface and datapath manager (DPM) to realize the data flow needed for the "object detection" chain. The source code for objectdetection DPC is present in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\objectedetection folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\datapath\dpc\objectedetection\objdethwa\docs\doxygen\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package mmwave_sdk_<ver>\packages\ti\datapath\dpc\objectedetection\objdethwa\test\. See section on Data Path tests using Test vector method for more details on this test.

5. 4. 9. mmWaveLib

mmWaveLib is a collection of algorithms that provide basic functionality needed for FMCW radar-cube processing. This component is available only for those mmWave devices that have DSP/C674 cores, It contains optimized library routines for C674 DSP architecture only. This component is not available for cortex R4F (MSS). These routines do not encapsulate any data movement/data placement functionality and it is the responsibility of the application code to place the input and output buffers in the right memory (ex: L2) and use EDMA as needed for the data movement. The source code for mmWaveLib is present in the mmwave_sdk_<ver>\packages\ti\alg\mmwavelib folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\alg\mmwavelib\docs\doxygen\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\alg\mmwavelib\test\ .

Functionality supported by the library:

- Collection of algorithms that provide basic functionality needed for FMCW radar-cube processing.
  - Windowing (16-bit complex input, 16 bit complex output, 16-bit windowing real array)
  - Windowing (16-bit complex input, 32 bit complex output, 32-bit windowing real array)
  - log2 of absolute value of 32-bit complex numbers
  - vector arithmetic (accumulation)
  - CFAR-CA, CFAR-CASO, CFAR-CAGO (logarithmic input samples)
  - 16-point FFT of input vectors of length 8 (other FFT routines are provided as part of DSPLib)
  - single DFT value for the input sequences at one specific index
  - Twiddle table generation for 32x32 and 16x16 FFTs: optimized equivalent functions of dsplib for generating twiddle factor
  - FFT Window coefficients generation
  - DFT sine/cosine table generation for DFT single bin calculation
  - Single bin DFT with windowing.
  - Variation of the windowing functions with I/Q swap since most of the fixed point FFT functions in DSPLib only support one format of complex types.
- CFAR algorithms
  - Floating-point CFAR-CA:
    - mmwavelib_cfarfloat_caall supports CFAR cell average, cell accumulation, SO, GO algorithms, with input signals in floating point formats;

        - mmwavelib_cfarfloat_caall_opt implements the same functionality as mmwavelib_cfarfloat_caall except with less cycles, but the detected objects will not be in the ascending order.
        - mmwavelib_cfarfloat_wrap implements the same functionality as mmwavelib_cfarfloat_caall except the noise samples for the samples at the edges are the circular rounds samples at the other edge.
        - mmwavelib_cfarfloat_wrap_opt implements the same functionality as mmwavelib_cfarfloat_wrap except with less cycles, but the detected objects will not be in the ascending order.
    - CFAR-OS: Ordered-Statistic CFAR algorithm
        - mmwavelib_cfarOS accepts fixed-point input data (16-bit log-magnitude accumulated over antennae). Search window size is defined at compile time.
    - Peak pruning for CFAR post-processing
        - mmwavelib_cfarPeakPruning: Accepts detection matrix and groups neighboring peaks into one.
        - mmwavelib_cfarPeakQualifiedInOrderPruning: Accepts the list of CFAR detected objects and groups neighboring peaks into one.
        - mmwavelib_cfarPeakQualifiedPruning: Same as mmwavelib_cfarPeakQualifiedInOrderPruning, but with no assumption for the order of cfar detected peaks
- Floating-point AOA estimation:
    - mmwavelib_aoaEstBFSinglePeak implements Bartlett beamformer algorithm for AOA estimation with single object detected, it also outputs the variance of the detected angle.
    - mmwavelib_aoaEstBFSinglePeakDet implements the save functionality as mmwavelib_aoaEstBFSinglePeak without the variance of detected angle calculation.
    - mmwavelib_aoaEstBFMultiPeak also implements the Bartlett beamformer algorithm but with multiple detected angles, it also outputs the variances for every detected angles.
    - mmwavelib_aoaEstBFMultiPeakDet implements the same functionality as mmwavelib_aoaEstBFMultiPeak but with no variances output for every detected angles.
- DBscan Clustering:
    - mmwavelib_dbscan implements density-based spatial clustering of applications with noise (DBSCAN) data clustering algorithm.
    - mmwavelib_dbscan_skipFoundNeiB also implements the DBSCAN clustering algorithm but when expanding the cluster, it skips the already found neighbors.
- Clutter Removal:
    - mmwavelib_vecsum: Sum the elements in 16-bit complex vector.
    - mmwavelib_vecsubc: Subtract const value from each element in 16-bit complex vector.
- Windowing:
    - mmwavelib_windowing16xl6_evenlen: Supports multiple-of-2 length(number of input complex elements), and mmwavelib_windowing16x16 supports multiple-of-8 length.
    - mmwavelib_windowing16x32: This is updated to support multiple-of-4 length(number of input complex elements). It was multiple-of-8 previously.
- Floating-point windowing:
    - mmwavelib_windowing1DFltp: support fixed-point signal in, and floating point signal out windowing, prepare the floating point data for 1D FFT.
    - mmwavelib_chirpProcWin2DFxdpinFltOut, mmwavelib_dopplerProcWin2DFxdpinFltOut: prepare the floating point data for 2D FFT, with fixed point input. The difference is mmwavelib_chirpProcWin2DFxdpinFltOut is done per chip bin, while mmwavelib_dopplerProcWin2DFxdpinFltOut is done per Doppler bin.
    - mmwavelib_windowing2DFltp: floating point signal in, floating point signal out windowing to prepare the floating point data for 2D FFT.
- Vector arithmetic
    - Floating-point and fixed point power accumulation: accumulates signal power.
    - Histogram: mmwavelib_histogram right-shifts unsigned 16-bit vector and calculates histogram.
    - Right shift operation on signed 16-bit vector or signed 32-bit vector
        - mmwavelib_shiftright16 shifts each signed 16-bit element in the input vector right by k bits.
        - mmwavelib_shiftright32 shifts each signed 32-bit element in the input vector right by k bits.
        - mmwavelib_shiftright32to16 right shifts 32-bit vector to 16-bit vector
    - Complex vector element-wise multiplication.
        - mmwavelib_vecmul16x16: multiplies two 16-bit complex vectors element by element. 16-bit complex output written in place to first input vector.
        - mmwavelib_vecmul16x32, mmwave_vecmul16x32_anylen : multiplies a 16-bit complex vector and a 32-bit complex vector element by element, and outputs to the 32-bit complex output vector.
        - mmwave_vecmul32x16c: multiplies 32bit complex vector with 16bit complex constant.
    - Sum of absolute value of 16-bit vector elements
        - mmwavelib_vecsumabs returns the 32-bit sum.
    - Max power search on 32-bit complex data
        - mmwavelib_maxpow outputs the maximum power found and returns the corresponding index of the complex sample
        - mmwavelib_powerAndMax : Power computation combined with max power search
    - Peak search for Azimuth estimation on 32-bit float vector
        - mmwavelib_multiPeakSearch : Multiple peak search in the azimuth FFT output
        - mmwavelib_secondPeakSearch : Second peak search in the azimuth FFT output
    - DC (antenna coupling signature) Removal on 32-bit float complex vector
    - Vector subtraction for 16-bit vectors
- Matrix utilities
    - Matrix transpose for 32-bit matrix: Similar to DSPLib function but optimized for matrix with rows larger than columns

5. 4. 10. Group Tracker

The algorithm is designed to track multiple targets, where each target is represented by a set of measurement points (point cloud output of CFAR detection layer). Each measurement point carries detection information, for example, range, angle, and radial velocity. Instead of tracking individual reflections, the algorithm predicts and updates the location and dispersion properties of the group. The group is defined as the set of measurements (typically, few tens; sometimes few hundreds) associated with a real life target. This algorithm is provided for all mmWave devices except xwr14xx but is supported for both R4F and C674x. The source code for gtrack is present in the mmwave_sdk_<ver>\packages\ti\alg\gtrack folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\alg\gtrack \docs\doxygen<2d|3D>\html\index.html. The component's unit test code, running on top of SYSBIOS is also provided as part of the package: mmwave_sdk_<ver>\packages\ti\alg\gtrack\test\ .

5. 4. 11. RADARSS Firmware

This is a binary (under mmwave_sdk_<ver>\firmware\radarss) that runs on Radar subsystem of the mmWave device and realizes the mmWave front end. It exposes configurability via a set of messages over mailbox which is understood by the mmWaveLink component running on the MSS. RADARSS firmware is responsible for configuring RF/analog and digital front-end in real-time, as well as to periodically schedule calibration and functional safety monitoring. This enables the mmWave front-end to be self-contained and capable of adapting itself to handle temperature and ageing effects, and to enable significant ease-of-use from an external host perspective. Features/enhancements information can be found in the platform specific release notes under mmwave_sdk_<ver>\firmware\radarss.

5. 4. 12. CCS Debug Utility

This is a simple binary that can flashed onto the board to facilitate the development phase of mmWave application using TI Code Composer Studio (CCS). See section CCS Development mode for more details. For xWR14xx, this binary is for R4F (MSS) and for other mmWave devices, there is an executable for both R4F (MSS) and C674 (DSS) and is combined into one metaImage for flashing along with RADARSS firmware. Note that the CCS debug application for C674 (DSS) has the L1 and L2 cache turned off so that new application that gets downloaded via CCS can enable it as needed, without any need for cache flush operations, etc during switching of applications. CCS debug for MSS (R4F) has the while loop implemented using ARM instruction set since its purpose is to allow users to load another application using CCS and the first instruction that the application would run will be _c_int00 which is compiled only in ARM mode.

5. 4. 13. HSI Header Utility

An optional utility library is provided for user to create a header that it can attach to the data being shipped over LVDS. This library accepts the CBUFF session configuration and creates a header with appropriate information filled in and passes it back to the calling application. The calling application can then supply this created header to CBUFF APIs. This config inside the header is intended to help user parse the LVDS on the receiving end. The source code for this utility is present in the mmwave_sdk_<ver>\packages\ti\utils\hsiheader folder. Documentation of the API is available via doxygen and placed at mmwave_sdk_<ver>\packages\ti\utils\hsiheader\docs\doxygen\html\index.html.

5. 4. 14. Secondary Bootloader

A reference implementation of secondary bootloader is provided in the SDK to show the usecase of updating the application metaimage in the SFLASH (outside of ROM bootloader) by receiving the image over any serial interface. Subsequent to successful flashing, it shows how to read the individual core images from the flash/metaImage, load them onto respective core memories and then execute the application. In addition to the flash read/write and metaImage parser functionality, it provides reference implementation for image validity and failsafe mechanisms. The source code for this utility is present in the mmwave_sdk_<ver>\packages\ti\utils\sbl folder. Documentation of the SBL is available at mmwave_sdk_<ver>\packages\ti\utils\sbl\docs.

5. 4. 15. mmWave SDK - System Initialization

Application should call init APIs for the following system modules (ESM, SOC, Pinmux) to enable correct operation of the device

5. 4. 15. 1. ESM

ESM_init should be the first function that is called by the application in its main(). Refer to the doxygen for this function at mmwave_sdk_<ver>\packages\ti\drivers\esm \docs\doxygen\html\index.html to understand the API specification.

5. 4. 15. 2. SOC

SOC_init should be the next function that should be called after ESM_init. Refer to the doxygen for this function at mmwave_sdk_<ver>\packages\ti\drivers\soc \docs\doxygen\html\index.html to understand the API specification. It primarily takes care of following things:

DSP un-halt

This applies for mmWave devices with DSP core. Bootloader loads the DSP application from the flash onto DSP's L2/L3 memory but doesnt un-halt the C674x core. It is the responsibility of the MSS application to un-halt the DSP. SOC_init for MSS provides this optional functionality under its hood. It is recommended to always unhalt the DSP when application needs to use the DSP for realizing its functionality. For applications that doesnt need DSP completely can choose to leave the DSP in its original state i.e. either in halt state when this application is running from the flash and booted by the bootloader OR in unhalted/while 1 loop state when this application is running from CCS in development mode.

RADARSS un-halt/System Clock

To enable selection of system frequency to use "closed loop APLL", the SOC_init function unhalts the RADARSS and then spins around waiting for acknowledgement from the RADARSS that the APLL clock close loop calibration is completed successfully.

> Note that this function assumes that the crystal frequency is 40MHz.

MPU (Cortex R4F)

MPU or Memory Protection Unit needs to be configured on the Cortex R4F of mmWave device for the following purposes:

- Protection of memories and peripheral (I/O) space e.g not allowing execution in I/O space or writes to program (.text) space.
- Controlling properties like cacheability, buferability and orderability for correctness and performance (execution time, memory bandwidth). Note that since there is no cache on R4F, cacheability is not enabled for any region.

Default MPU settings has been implemented in the SOC module as a private function SOC_mpu_config() that is called by public API SOC_init() when SOC_MPUCfg_CONFIG option is passed by the application. Doxygen of SOC (mmwave_sdk_<ver>\packages\ti\drivers\soc\docs\doxygen\html\index.html) has SOC_mpu_config() documented with details of choice of memory regions etc. When MPU violation happens, BIOS will automatically trap and produce a dump of registers that indicate which address access caused violation (e.g DFAR which indicates what data address access caused violation). Note: The SOC function uses as many MPU regions as possible to cover all the memory space available on the respective device. There may be some free MPU regions available for certain devcies (ex: xWR14xx) for the application to use and program as per their requirement. See the function implementation/doxygen for more details on the usage and availability of the MPU regions. If the application needs for the MPU are different than the default settings, it can pass SOC_MPUCfg_BYPASS_CONFIG to SOC_init function and then it can either pre-configure or post configure the MPU using exported SOC_MPUxxx() functions. Application is responsible for the correct MPU settings when SOC_MPUCfg_BYPASS_CONFIG mode is chosen.

> A build time option called DOWNLOAD_FROM_CCS has been added which when set to yes prevents program space from being protected in case of SOC owned default MPU settings. This option should be set to yes when debugging using CCS because CCS, by default, attempts to put software break-point at main() on program load which requires it to change (temporarily) the instruction at beginning main to software breakpoint and this will fail if program space is read-only. Hence the benefit of code space protection is not there when using CCS for download. It is however recommended to set this option to no when building the application for production so that program space is protected.

MARs (DSP/C674x)

The cacheability property of the various regions as seen by the DSP (C674x) is controlled by the MAR registers. These registers are programmed as per driver needs in in the SOC module as a private function SOC_configMARs() that is called by public API SOC_init(). See the doxygen documentation of this function to get more details. Note that the drivers do not operate on L3 RAM and HS-RAM, hence L3/HS-RAM cacheability is left to the application/demo code writers to set and do appropriate cache (writeback/invalidate etc) operations from the application as necessary, depending on the use cases. The L3 MAR is MAR32 -> 2000_0000h - 20FF_FFFFh and HS-RAM MAR is MAR33 -> 2100_0000h - 21FF_FFFFh.

5. 4. 15. 3. Pinmux

Pinmux module is provided under mmwave_sdk_<ver>\packages\ti\drivers\pinmux with API documentation and available device pads located at mmwave_sdk_<ver>\packages\ti\drivers\pinmux\docs\doxygen\html\index.html. Application should call these pinmux APIs in the main() to correctly configure the device pads as per their hardware design.

> **TI Pinmux Utility**
>
> TI Pinmux Tool available at https://dev.ti.com/pinmux supports mmWave devices and can be used for designing the pinmux configuration for custom board. It also generates code that can be included by the application and compiled on top of mmWave SDK and its Pinmux driver.

5. 4. 16. Usecases

5. 4. 16. 1. Data Path tests using Test vector method

The data path processing on mmWave device for 1D, 2D and 3D processing consists of a coordinated execution between the MSS, HWA /DSS and EDMA. This is demonstrated as part of the object detection processing chain and millimeter wave demo. The demo runs in real-time and has all the associated framework for RADARSS control etc with it.

The unit tests located at mmwave_sdk_<ver>\packages\ti\datapath\dpc \objectdetection\<chain_type>\test) are stand-alone tests that allow data path processing chain to be executed in non real-time. This allows developer to use it as a debug/development aid towards eventually making the data path processing real-time with real chirping. Developer can easily step into the code and test against knowns input signals. The core data path processing source code in object detection chain and the processing modules (DPUs) is shared between this test and the

mmw demo. Most of the documentation is therefore shared as well and can be looked up in the object detection DPC and mmw demo documentation.

The tests also provide a test generator, which allows user to set objects artificially at desired range, doppler and azimuth bins, and noise level so that output can be checked against these settings. It can generate one frame of data. The test generation and verification are integrated into the tests, allowing developer to run a single executable that contains the input vector and also verifies the output (after the data path processing chain), thereby declaring pass or fail at the end of the test. The details of test generator can be seen in the doxygen documentation of these tests located at mmwave_sdk_<ver>\packages\ti\datapath\dpc \objectdetection\<chain_type>\test\docs\doxygen\html\index.html.

5. 4. 16. 2. CSI-2 based streaming of ADC data

IWR14xx device has a high speed CSI-2 transmit interface that can be used to ship ADC data or 1D/2D processed data out of the device. An example usecase on how to program the front end to generate the ADC samples and tie it up to CBUFF/CSI-2 interface for data shipment is provided under mmwave_sdk_<ver>\packages\ti\drivers\test\csi_stream. Refer to the doxygen documentation located at mmwave_sdk_<ver>\packages\ti\drivers\test\csi_stream\docs\doxygen\html\index.html for more details.

5. 4. 16. 3. Basic configuration of Front end and capturing ADC data in L3 memory

To access ADC data from mmWave sensors, user need to program various basic components within the device in a given sequence. In order to help user understand the programming model needed to configure the device and generate ADC data in device's L3 memory, an example usecase is provided under mmwave_sdk_<ver>\packages\ti\drivers\test\mem_capture. Refer to the doxygen documentation located at mmwave_sdk_<ver>\packages\ti\drivers\test\mem_capture\docs\doxygen\html\index.html for more details.

6. Appendix

6. 1. Memory usage

The map files of demo and driver unit test application captures the memory usage of various components in the system. They are located in the same folder as the corresponding .xer4f/.xe674 and .bin files. Additionally, the doxygen for mmW demo summarizes the usage of various memories available on the device across the demo application and other SDK components. Refer to the section "Memory Usage" in the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\docs\doxygen\html\index.html documentation.

6. 2. Register layout

The register layout of the device is available inside each hardware IP's driver source code. See mmwave_sdk_<ver>\packages\ti\drivers\<ip>\include\reg_*.h. The system level registers (RCM, TOPRCM, etc) are available under the SOC module (mmwave_sdk_<ver>\packages\ti\drivers\soc\include\reg_*.h).

6. 3. Enable DebugP logs

The DebugP_log OSAL APIs in ti/drivers/osal/DebugP.h are used in the drivers and test/app code for debug streaming. These are tied to BIOS's Log_* APIs and are well documented in SYSBIOS documentation. The logs generated by these APIs can be directed to be stored in a circular buffer and observed using ROV in CCS (http://rtsc.eclipse.org/docs-tip/Runtime_Object_Viewer).

Following steps should be followed to enable these logs:

1. Enable the flag DebugP_LOG_ENABLED before the header inclusion as seen below.

```
#define DebugP_LOG_ENABLED 1
#include <ti/drivers/osal/DebugP.h>
```

2. Add the following lines in your SYSBIOS cfg file with appropriate setting of numEntries (number of messages) which will impact memory space:

Application SYSBIOS cfg file

```
var Log       = xdc.useModule('xdc.runtime.Log');
var Main = xdc.useModule('xdc.runtime.Main');
var Diags = xdc.useModule('xdc.runtime.Diags');
var LoggerBuf                   = xdc.useModule('xdc.runtime.LoggerBuf');
LoggerBuf.TimestampProxy        = xdc.useModule('xdc.runtime.Timestamp');

/* Trace Log */
var loggerBufParams = new LoggerBuf.Params();
loggerBufParams.bufType = LoggerBuf.BufType_CIRCULAR; //BufType_FIXED
loggerBufParams.exitFlush = false;
loggerBufParams.instance.name = "_logInfo";
loggerBufParams.numEntries = 100; <-- number of messages this will affect memory consumption
// loggerBufParams.bufSection = ;
_logInfo = LoggerBuf.create(loggerBufParams);
Main.common$.logger = _logInfo;

/* Turn on USER1 logs in Main module (all non-module functions) */
Main.common$.diags_USER1 = Diags.RUNTIME_ON;

/* Turn on USER1 logs in Task module */
Task.common$.diags_USER1 = Diags.RUNTIME_ON;
```

A sample ROV log looks like below after code is re-build and run with above changes :

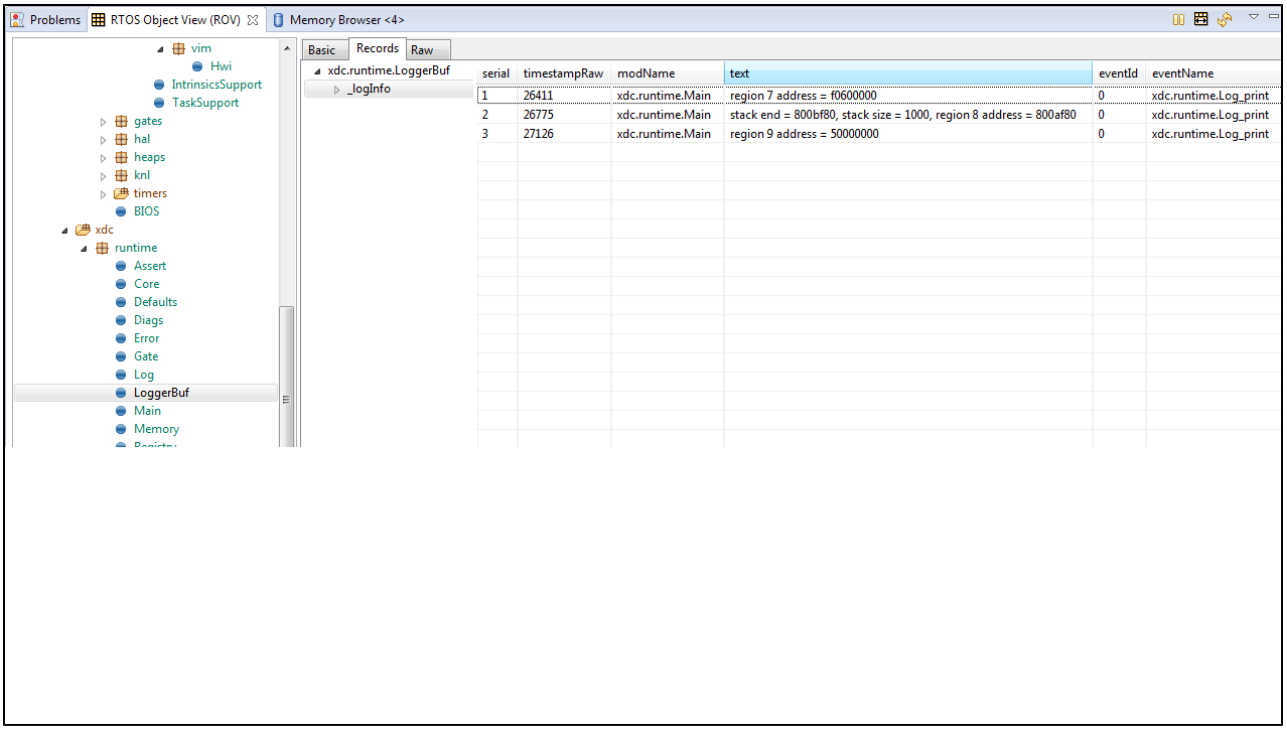


Figure 33: Sample ROV log with debug prints

6. 4. Shared memory usage by SDK demos

Existing SDK demos (mmw) assigns all available banks of shared memory to L3 memory. No additional banks are added to MSS TCMA and TCMB; they remain at the default memory size. See TRM for more details on the L3 memory layout and "xWR1xxx Image Creator User Guide" in SDK for more details on shared memory allocation when creating flash images. Note that the image that is programmed into the flash of the mmWave device determines the shared memory allocation. So in CCS development mode, it is the allocation defined in ccsdebug metaImage that applies and not the application that you download via CCS.

In SDK code, one can change the environment variable MMWAVE_SDK_SHMEM_ALLOC to customize the shmem alloc settings. If this variable is undefined, platform specific SDK common makefile (mmwave_sdk_<ver>\packages\ti\common\mmwave_sdk_<platform>.mak) will define the default values. When this variable is changed, user should do a clean build of the demo or ccsdebug depending on the working mode. This setting will influence

- the size of L3 memory section in linker command files (mmwave_sdk_<ver>\packages\ti\platform\<platform>)
- the sys_common defines for the L3, TCMA and TCMB memory sizes for the application code to use and size the buffers, heaps, etc accordingly. (ex: SOC_XWR16XX_MSS_TCMA_SIZE, SOC_XWR16XX_MSS_L3RAM_SIZE, etc)
- the shmem_alloc input parameter to generateMetaImage script in ccsdebug and mmw demo makefiles.

Since there is a chance for sys_common defines for the memories and metaImage bank allocation to go out of sync (due to user error such as failure to do clean build), SOC module init does a sanity check of the hardware programmed L3 bank allocations (that are fed via metaImage header) and the sys_common defines. If the sys_common defined memory size is greater than hardware programmed bank allocations, the module throws an assert.

6. 5. mmWave Device Image Creator

This section outlines the tools used for image creation needed for flashing the mmWave devices. The application executable generated after the compile and link step needs to be converted into a bin form for the bootloader to understand and burn it onto the serial flash present on the device. The demos inside the mmWave SDK already incorporate the step of bin file generation as part of their makefile and no further steps are required. This section is helpful for application writers that do not have makefiles similar to the SDK demos. Once the compile and link step is done, application image generation is described as follows.

The Application Image interpreted by the bootloader is a consolidated Multicore image file that includes the RPRC image file of individual subsystems along with a Meta header. The Meta Header is a Table of Contents like information that contains the offsets to the individual subsystem RPRC images along with an integrity check information using CRC. In addition, the allocation of the shared memory to the various memories of the subsystems also has to be specified. The bootloader performs the allocation accordingly. It is recommended that the allocation of shared memory is predetermined and not changed dynamically.

Use the generateMetaImage script present under mmwave_sdk_<ver>\packages\ scripts\windows or mmwave_sdk_<ver>\packages\ scripts\linux for merging the MSS .xer4f, DSS .xe674 and RADARSS RPRC binaries into one metaImage and appending correct CRC. The RPRC image for MSS and DSS are generated internally in this script from the input ELF formatted files for those subsystem (i.e. output of linker command - .xer4f, .xe674). Set **MMWAVE_SDK_INSTALL_PATH=mmwave_sdk_<ver>\packages** in your environment before calling this script. This script needs 5 parameters:

- FLASHIMAGE: [output] multicore file that will be generated by this script and should be used for flashing onto the board
- SHMEM_ALLOC: [input] shared memory allocation in 32-bit hex format where each byte (left to right) is the number of banks needed for RADARSS (BSS),TCMB,TCMA and DSS. Refer to the the TRM on details on L3 shared memory layout and "Image Creator User Guide" in the SDK. It is advisable to pass MMWAVE_SDK_SHMEM_ALLOC environment variable here to keep the compiled code and metaImage in sync. See Shared memory usage by SDK demos section for more details.
- MSS_IMAGE_OUT: [input] MSS input image in ELF (.xer4f) format as generated by the linker. Use keyword NULL if not needed
- BSS_IMAGE_BIN: [input] RADARSS (BSS) input image in RPRC (.bin) format, use keyword NULL if not needed. Use mmwave_sdk_<ver>\firmware\radarss\<platform>_radarss_rprc.bin here. For xWR14xx, select xwr12xx_xwr14xx_radarss_rprc.bin.
- DSS_IMAGE_OUT: [input] DSP input image in ELF (,xe674) format as generated by the linker. Use keyword NULL if not needed

The FLASHIMAGE file generated by this script should be used for the METAIMAGE1 during flashing step (How to flash an image onto mmWave EVM). Refer to "Image Creator User Guide" in the SDK docs directory for details on the internal layout and format of the files generated in these steps.

6. 6. mmw Demo: cryptic message seen on DebugP_assert

In mmw demo, the BIOS cfg file dss_mmw.cfg has below code at the end to optimize BIOS size. Because of some of these changes, exceptions, such as those generated through DebugP_assert() calls may give a cryptic message instead of file name and line number that helps identify easily where the exception is located. To be able to restore this capability, the user can comment out the lines marked with the comment "" below. For more information, refer to the BIOS user guide.

```
/* Some options to reduce BIOS code and data size, see BIOS User Guide section
   "Minimizing the Application Footprint" */
System.maxAtexitHandlers = 0; /* COMMENT THIS FOR FIXING DebugP_Assert PRINTS */
BIOS.swiEnabled = false; /* We don't use SWIs */
BIOS.libType = BIOS.LibType_Custom;
Task.defaultStackSize = 1500;
Task.idleTaskStackSize = 800;
Program.stack = 1048; /* for isr context */
var Text = xdc.useModule('xdc.runtime.Text');
Text.isLoaded = false;
```

6. 7. How to execute Idle instruction in idle task when using SYSBIOS

The idle function hook provided by SYSBIOS can be used to install application specific function which in turn could call the "idle" asm instruction. See code snapshots below or refer to mmW demo for details.

BIOS CFG file

```
var Idle      = xdc.useModule('ti.sysbios.knl.Idle');
Idle.addFunc('&MmwDemo_sleep');
```

WFI instruction for R4F

```
void MmwDemo_sleep(void)
{
    /* issue WFI (Wait For Interrupt) instruction */
    asm(" WFI ");
}
```

IDLE instruction for C674x

```
void MmwDemo_sleep(void)
{
    /* issue IDLE instruction */
```

```
        asm(" IDLE ");
    }
```

6. 8. Range Bias and Rx Channel Gain/Offset Measurement and Compensation

Refer to the section "Range Bias and Rx Channel Gain/Offset Measurement and Compensation" in the mmwave_sdk_<ver>\packages\ti\datapath\dpc\objectdetection\<chain_type>\docs\doxygen\html\index.html documentation for the procedure and internal implementation details. To execute the procedure using Visualizer GUI, here are the steps:

- Set the target as explained in the demo documentation and update the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\profiles\profile_calibration.cfg appropriately.
- Set up Visualizer and mmW demo as mentioned in the section Running the Demos.
- Use the "Load Config From PC and Send" button on plots tab to send the mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\profiles\profile_calibration.cfg.
- The Console messages window on the Configure tab will dump the "compRangeBiasAndRxChanPhase" command to be used for subsequent runs where compensation is desired.
- Copy and save the string for that particular mmWave sensor to your PC. You can use it in the "Advanced config" tab in the Visualizer and tune any running profile in real time. Alternatively, you can add this to your custom profile configs and use it via the "Load Config From PC and Send" button.

6. 9. Guidelines on optimizing memory usage

Depending on requirements of a given application, there may be a need to optimize memory usage, particularly given the fact that the mmWave devices do not have external RAM interfaces to augment on-chip memories. Below is a list of some optimizations techniques, some of which are illustrated in the mmWave SDK demos (mmW demo). It should be noted, however, that the demo application memory requirements are dictated by requirements like ease/flexibility of evaluation of the silicon etc, rather than that of an actual embedded product deployed in the field to meet specific customer user cases.

1. On R4F, compile your application with ARM thumb option (depending on the compiler use). If using the TI ARM compiler, the option to do thumb is `code_state=16` . Another relevant compiler option (when using TI compiler) to play with to trade-off code size versus speed is `--opt_for_speed=0-5` . For more information, refer to ARM Compiler Optimizations and ARM Optimizing Compiler User's Guide. The pre-built drivers in the SDK are already built with the thumb option. The demo code and BIOS libraries are also built with thumb option. Note the code_state=16 flag and the ti.targets.arm.elf.**R4Ft** target in the mmwave_sdk_<ver>\packages\ti\ common\mmwave_sdk.mak.
2. On C674X, compile portions of code that are not in compute critical path with appropriate -msX option. The -ms0 options is presently used in the SDK drivers, demos and BIOS cfg file. This option does cause compiler to favor code size over performance and hence some cycles impact are to be expected. However, on mmWave device, using ms0 option only caused about 1% change in the CPU load during active and interframe time and around 3-5% increase in config cycles when benchmarked using driver unit tests. For more details on the "ms" options, refer to The TI C6000 compiler user guide at C6000 Optimizing Compiler Users Guide. Another option to consider is -mo (this is used in SDK) and for more information, see section "Generating Function Subsections (--gen_func_subsections Compiler Option)" in the compiler user guide. A link of references for optimization (both compute and memory) is at Optimization Techniques for the TI C6000 Compiler.
3. Even with aggressive code size reduction options, the C674X tends to have a bigger footprint of control code than the same C code compiled on R4F. So if feasible, partition the software to use C674X mainly for compute intensive signal-processing type code and keep more of the control code on the R4F. An example of this is in the mmw demo, where we show the usage of mmwave API to do configuration (of RADARSS) from R4F instead of the C674X (even though the API allows usage from either domain). In mmw demo, this prevents linking of `mmwave` (in mmwave_sdk_<ver>\packages\ ti\control) and mmwavelink (in mmwave_sdk_<ver>\packages\ti\control) code that is involved in configuration (profile config, chirp config etc) on the C674X side as seen from the .map files of mss and dss located along with application binary.
4. If using TI BIOS as the operating system, depending on the application needs for debug, safety etc, the BIOS footprint in the application may be reduced by using some of the techniques listed in the BIOS User Guide in the section "Minimizing the Application Footprint". Some of these are illustrated in the mmw demo on R4F and C674X. Some common ones are disabling system_printf (printf strings do contribute to significant code size), choosing sysmin and using ROV for debugging, disabling assert (although this should be done only when variability in driver configuration is not expected and existing configuration has been proven to function correctly). The savings from these features could be anywhere from 2KB to 10KB but user would lose some perks of debuggability.
5. If there is no requirement to be able to restart an application without reloading, then following suggestions may be used:
    1. one time/first time only program code can be overlaid with data memory buffers used after such code is executed. This is illustrated in some of the mmw demos where such code is overlaid with (load time uninitialized) radar cube data in L3 RAM. Refer to the demos linker command files for more details (Note: Ability to place code at function granularity requires to use the aforementioned -mo option).
    2. the linker option `--ram_model` may be used to eliminate the `.cinit` section overhead. For more details, see compiler user guide referenced previously. Presently, ram model cannot be used on R4F due to bootloader limitation but can be used on C674X. The SDK uses ram model when building C674X executable images (unit tests and demos).
6. On C674X, smaller L1D/L1P cache sizes may be used to increase static RAM. The L1P and L1D can be used as part SRAM and part cache. Smaller L1 caches can increase compute time due to more cache misses but if appropriate data/code is allocated in the SRAMs, then the loss in compute can be compensated (or in some cases can also result in improvement in performance). In the demos, the caches are sized to be 16 KB, allowing 16 KB of L1D and 16 KB of L1P to be used as SRAM. On the mmw demo, the L1D SRAM is used to allocate some buffers involved in data path processing whereas the L1P SRAM has code that is frequently

and more fully accessed during data path processing. Thus we get overall 32 KB more memory. The caches can be reduced all the way down to 0 to give the full 32 KB as SRAM, how much cache or RAM is a decision each application developer can make depending on the memory and compute (MIPS) needs.

7. When modeling the application code using mmW demo as reference code, it might be useful to trim the heaps in mmW demo to claim back the unused portion of the heaps and use it for code/data memory. Ideally, a user can run their worst case profile that they would like to support using mmW demo, record the heap usage/free metrics for (L1D, L2)/TCMB and L3 heaps on 'sensorStart'. These values can then be used to resize or re-allocate heap globals (example: gDPC_ObjDetTCM, gMmwL3, etc) in mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw. The freed up space in DSS could be used as follows:

    1. Free heap space in L1D could be used to move some of the L2 buffers to L1D. The freed L2 space can be used for code /data.
    2. Free heap space in L2 could be trimmed by changing the heap's global variable (ex: gMmwL2) definition and used for code /data memory (note that code memory by default is L2 so no other change is required to get more code space).
    3. Free heap space in L3 could be trimmed by changing the heap's global variable (ex: gMmwL3) definition and used for code /data space.

When using TI compilers for both R4F and C674x, the map files contain a nice 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. See one sample snapshot below:

Module summary inside application's .map file

```
MODULE SUMMARY

       Module                          code     ro data   rw data
       ------                          ----     -------   -------


    obj_xwr14xx/
       main.oer4f                      5191     0         263980
       data_path.oer4f                 8441     0         65536
       config_hwa_util.oer4f           4049     0         0
       post_processing.oer4f           2480     0         0
       mmw_cli.oer4f                   2308     0         0
       config_edma_util.oer4f          1276     0         0
       sensor_mgmt.oer4f               1144     0         24
    +--+-------------------------------+--------+---------+---------+
       Total:                          24889    0         329540
```

6. 10. How to add a .const (table) beyond L3 heap in mmWave application where overlay is enabled

To achieve L3 heap overlaid with the code to be copied into L1P at init time, L3 heap is in PAGE 1 and code is in Page 0. PAGE 0 is the only loadable page whereas PAGE 1 is just a dummy page to allocate uninitialized sections to implement overlay. As a result the ".const" section (which is loadable section) cannot simply be allocated to PAGE 1 to go after the heap. If the .const is allocated in PAGE 0, then it will overlap the heap and will be overwritten once heap is allocated. To resolve this, the HIGH feature of the linker could be used is used to push the const table to the highest address ensuring no overlap with L3 heap. The suggested changes would be as follows:
1. Shrink the L3 heap by the size of the table (but L3 heap must still be bigger than the size of the L1P cache).
2. Place the table in a named section and allocate the named section in the HIGH space of PAGE 0 of L3RAM.

This ensures that the table will be allocated at the high address and will not be overlapping with L3 heap or the L1P intended code which are located at the low address.

Sample code is shown below.

```
In application C file:

#define TABLE_LENGTH 4
#define TABLE_ALIGNMENT 8 /* bytes */

/*! L3 RAM buffer, shrunk by table */
#pragma DATA_SECTION(gMmwL3, ".l3data");
#pragma DATA_ALIGN(gMmwL3, 8);
uint8_t gMmwL3[SOC_XWR16XX_DSS_L3RAM_SIZE - TABLE_LENGTH*sizeof(float) - TABLE_ALIGNMENT];

#pragma DATA_SECTION(gArray, ".l3data_garray");
#pragma DATA_ALIGN(gArray, TABLE_ALIGNMENT);
const float gArray[TABLE_LENGTH] = {1.5, 3.2, 0.8, -9.6};
```

```
In linker command file:
    .l3data_garray:  load=L3SRAM PAGE 0 (HIGH)
```

6. 11. Enabling L3 cache for DSP/C674x on mmWave devices

In a given usecase for mmWave devices, if L3 RAM is not fully utilized for Radar Cube storage, then the remaining free L3 memory could ideally be used for code and other internal data storages for the application. However, access to L3 memory from DSP/C674x core in mmWave devices is slower than accessing L1/L2. The cache-based memory system of C674x can be efficiently used in such cases. Refer to C674x DSP Cache User Guide (http://www.ti.com/lit/ug/sprug82a/sprug82a.pdf) for more details on the L1P/L1D/L2 two-level hierarchy that exists within the C674x memory architecture. L1P, L1D and L2D can be partitioned into SRAM and cache. L1P, L1D and L2 cache size can be set through linker command file -please refer to mmwave_sdk_<ver>/packages/ti/platform/<platform>/c674x_linker.cmd for more details. L2 SRAM addresses are always cached in L1P and L1D. However, external memory addresses (ex: code/data in L3) by default are configured as non-cacheable in L1D and L2 caches. Cacheability for external memory addressed (ex: L3) must first be explicitly enabled by the user using the MAR registers. Note that L1P cache is not affected by this configuration and always caches external memory addresses.

- **Cache writeback**: To maintain cache coherency between different masters (CPU, DMA, R4F, etc), content in cache needs to be written back to memory after it is changed before triggering the other master to access that memory location.
- **Cache Invalidate**: Before reading the content from the physical memory that was updated by another master, the content in cache needs to be invalidated, so that updated data from memory can be loaded in cache.
- **APIs**: User can use DSPICFGRegs directly from mmwave_sdk_<ver>/packages/ti/drivers/soc/include/reg_dspicfg.h or the TI BIOS cache module APIs to perform these MAR settings, cache invalidates and cache writebacks.
- **Code in L3**: mmWave code can be placed from L2 to L3 (via linker command file) with no explicit need for cache enablement and/or cache operations during real time. The only setting that needs to be adjusted is the size of L1P cache and that should be balanced against the need for L1P SRAM to place real time optimized functions (and avoid any cache misses, etc).
- **Data in L3**: If data cache is enabled for L3 memory via the MAR registers, then at first, one needs to take care of cache invalidates and writebacks for existing data structures in L3 memory. Radarcube and detection matrix are the primary data structures placed in L3 memory in case of a typical mmwave application on our device. Typically Radarcube is accessed (read/write) only via EDMA during the Range and Doppler FFT. Post that, it is more common for the DSP core to access the radarcube directly (i.e. no EDMA) and primarily it is a read access. In such scenario, the Radarcube can be invalidated at the end of current frame but before the start of next frame (i.e. when EDMA master begins to access radarcube). If the Radarcube was modified by the core directly (write operation) during the interframe time, then cache writeback_invalidate is needed at the end of current frame but before the start of next frame. Same consideration would apply for detection matrix. Next, mmWave internal data structures that are accessed purely by DSP can also be moved from L2 to L3 (via linker command file). No explicit cache writeback/invalidations are required for such structures. If user chooses to place the frame results structures in L3 (point cloud, etc) which are shared with MSS (R4F), then cache writeback+invalidate needs to be performed before signaling the MSS about availability of frame results. Note: If the analysis of L3 data access pattern between the DSP, MSS and EDMA shows that cache writeback/invalidate of all L3 data content can be done towards the end of the current frame, then performing writeback+invalidate on entire L1D cache might be a better option than calling such API on individual structures.

6. 12. DSPlib integration in mmWave C674x based application (Using 2 libraries simultaneously)

The TI C674X DSP is a merger of C64x+ (fixed point) and C67x+ (floating point) DSP architectures and DSPlib offers two different flavors of library for each of these DSP architectures. An application on C674X may need functions from both architectures. Normally this would be a straight-forward exercise like integrating other TI components/libraries. However there is a problem during integration of the two DSPLib libraries in the same application since the top level library API header `dsplib.h` has the same name and same relative path from the packages/ directory as seen below in the installation:

```
C:\ti\dsplib_c64Px_3_4_0_0\packages\ti\dsplib\dsplib.h
C:\ti\dsplib_c674x_3_4_0_0\packages\ti\dsplib\dsplib.h
```

Typically when integrating TI components, the build paths are specified up to `packages\` directory and headers are referred as below:

```
#include <ti/dsplib/dsplib.h>
```

However this will create an ambiguity when both libraries are to be integrated because the above path is same for both. There are a couple of ways to resolve this:

6. 12. 1. Integrating individual functions from each library

In this case, the headers individual functions are included in the application source file and the build infrastructure (makefiles for example) refers to the paths to the individual functions. This style of integration is illustrated in the following code snippets:

Sample DSPLib integration using individual functions

In application's makefile:

dss_mmw.mak

```
dssDemo: C674_CFLAGS += --cmd_file=$(BUILD_CONFIGPKG)/compiler.opt \
                                                /* include path for DSP_fft16x16    *
/    \
                        -i$(C64Px_DSPLIB_INSTALL_PATH)/packages/ti/dsplib/src/DSP_fft16x16
/c64P \
                        /* include path for DSP_fft32x32    */    \
                        -i$(C64Px_DSPLIB_INSTALL_PATH)/packages/ti/dsplib/src/DSP_fft32x32
/c64 \
                        -i$(C674x_MATHLIB_INSTALL_PATH)/packages \
```

In application C file:

dss_data_path.c

```
#include "DSP_fft32x32.h"
#include "DSP_fft16x16.h"
```

The C674P library can be integrated in the above code similar to the how the C64P has been done, this will not create any conflict.

A variant of the above could be as follows where the paths are now in the .c and .mak only refers to the installation:

dss_mmw.mak

```
dssDemo: C674_CFLAGS += --cmd_file=$(BUILD_CONFIGPKG)/compiler.opt \
                        -i$(C64Px_DSPLIB_INSTALL_PATH)/packages \
                        -i$(C674x_MATHLIB_INSTALL_PATH)/packages \
```

dss_data_path.c

```
#include <ti/dsplib/src/DSP_fft16x16/c64P/DSP_fft32x32.h>
#include <ti/dsplib/src/DSP_fft16x16/c64P/DSP_fft16x16.h>
```

6. 12. 2. Patching the installation

The previous method can get cumbersome if there are many functions to be integrated from both libraries. Patching the installation to rename /duplicate the top level API header `dsplib.h` allows a straight-forward integration. This prevents the name conflict of the two headers. So the installation after patching would look like below for example:

```
C:\ti\dsplib_c64Px_3_4_0_0\packages\ti\dsplib\dsplib_c64P.h [one can retain the older dsplib.h if one
wants to]
C:\ti\dsplib_c674x_3_4_0_0\packages\ti\dsplib\dsplib_c674x.h [one can retain the older dsplib.h if one
wants to]
```

And the .mak and code will look like below:

Sample DSPLib integration after renaming header files

In application makefile:

dss_mmw.mak

```
dssDemo: C674_CFLAGS += --cmd_file=$(BUILD_CONFIGPKG)/compiler.opt \
                        -i$(C64Px_DSPLIB_INSTALL_PATH)/packages \ <-- C64P dsplib
                        -i$(C674x_DSPLIB_INSTALL_PATH)/packages \ <-- C674x dsplib
                        -i$(C674x_MATHLIB_INSTALL_PATH)/packages \
```

In application C file:

dss_data_path.c

```
#include <ti/dsplib/dsplib_c64P.h>
#include <ti/dsplib/dsplib_c674x.h>
```

The present dsplibs do not have name conflicts among their functions so they can both be integrated in the above manner.

6. 13. SDK Demos: miscellaneous information

A detailed explanation of the mmW demo is available in the demo's docs folder: mmwave_sdk_<ver>\packages\ti\demo\<platform>\mmw\docs\doxygen\html\index.html. Some miscellaneous details are captured here:

- In demos that use HWA as the only processing node and elevation is enabled during run-time via configuration file, the number of detected objects are limited by the amount of HWA memory that is available for post processing.
- Demo's rov.xs file is provided in the SDK package to facilitate the CCS debugging of pre-built binaries when demo is directly flashed onto the device (instead of loading via CCS).
- When using non-interleaved mode for ADCBuf, the ADCBuf offsets for every RX antenna/channel enabled need to be multiple of 16 bytes.
- Output packet of mmW demo data over UART is in TLV format and its length is a multiple of 32 bytes. This enables post processing elements on the remote side (PC, etc) to process TLV format with header efficiently.

6. 14. Data size restriction for a given session when sending data over LVDS

For the current implementation of the CBUFF/LVDS driver and its intended usage, the CBUFF data size for a given session needs to be multiple of 8.

User should take care of this restriction when writing their custom application using the SDK LVDS driver. This alignment is taken care by the HSI header library if the application uses the headers for LVDS streaming. If no header are used while streaming data over LVDS lanes, user should calculate the total data size in bytes for the hardware triggered session (i.e. per chirp) and make sure it follows the rules mentioned above. Similar rules apply for the user data sent during the software triggered session.

6. 15. CCS Debugging of real time application

It is relatively easier to debug code before real-time starts because single-stepping or adding break-points does not affect the debugging since there is no real-time data and deadline to process the data. But once real-time starts, which is after sensor is started, such debugging can be intrusive and problematic. Below are some tips that may be helpful in real-time debugging, some of them are relevant to the out of box demos but may be applied in user applications if relevant.

6. 15. 1. Inter-chirp debugging

In out of box demos and many application specialized demos based on the SDK provided by TI (through the TI resource explorer), the inter-chirp processing is based on either HWA or DSP but not a mix of the two. In the case of HWA, the CPU/CPUs are idling with respect to inter-chirp processing so there is no need to halt. If one intends to stop and examine the state of HWA-EDMA during any of the intermediate processing steps, the design would have to be changed to issue a HWA or EDMA interrupt to the CPU that configured these (typically MSS CPU) at this intermediate state and the interrupt could read out some state and store in global variables that could be examined later. If code is halted using a break-point in the interrupt, the EDMA will automatically halt but HWA will not unless HWA is waiting on EDMA, so HWA could continue to run even if the CPU is halted. The current radar SoCs do not have the feature to halt the HWA when any of the CPUs are halted.

In case of DSP doing the inter-chirp processing, there can be a need to single-step/break the processing. However, (unlike the MSS CPU) when DSP is halted, the RadarSS (front end) doesnt halt and the chirping activity does not stop. Because of this, the DSP will miss the chirp processing deadline and the code is typically written to throw an exception. So basically halted debug is not useful unless a single chirp is configured and problem can be recreated with a single chirp. There might be other limitations in the demo code that may prevent a single chirp configuration (e.g in the AWR1642 demo, minimum number of doppler bins is required to be 16 due to DSPLIB FFT function restrictions and there is an error check for this during config validation - but this check may be disabled temporarily as it wouldn't affect the inter-chirp processing). Other techniques shown in below sections (real-time logging, using non real-time unit test bench) may be more practical but

have their own limitations. In most implementations however, 1D processing uses a hardened component from the SDK - the range DPU - so the need for real-time debugging in the active chirping period is low.

6. 15. 2. Inter-frame debugging

As there is no RadarSS chirping activity when MSS CPU is halted, it is possible to do halted debug in MSS during inter-frame debugging without running out of real-time. But on DSP, the device behavior is the opposite i.e the chirping will continue even if DSP is in halted state, so stepping in the DSP will cause an inter-frame deadline miss exception when running the out of box demo and other special demos that are implemented similarly. One technique that may be helpful in this situation is if the problem can be observed in the first frame itself, configure the chirping profile to do only one frame (frameCfg CLI command). This way after the active frame period, there is no chirp overrun (of the next frame) pressure when single-stepping in the inter-frame processing.

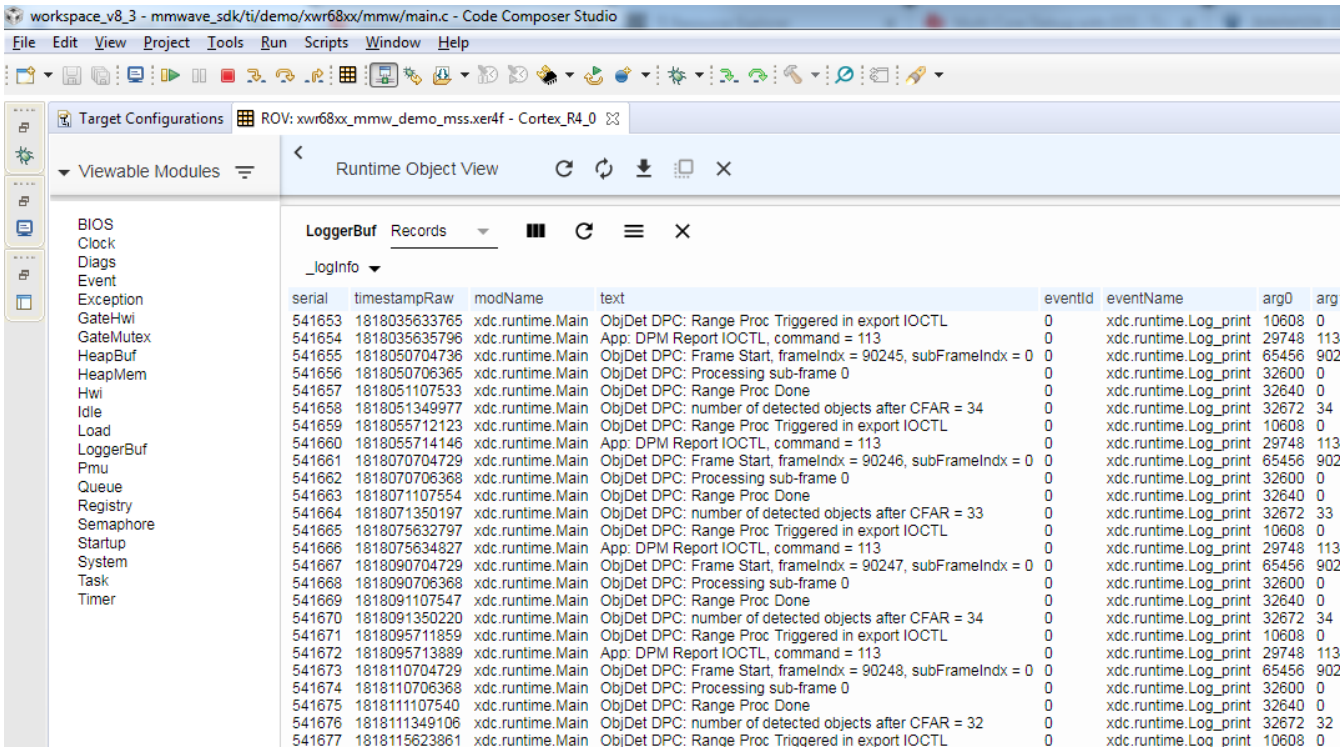
6. 15. 3. Using non-real time chain test code

See section "Data Path tests using Test vector method" on details about the non-real time chain that is provided with the mmWave SDK. Users can use these tests to step through the OOB processing chain in non-real time mode and debug or learn the components of the OOB processing chain.

6. 15. 4. Using printfs in real time

This applies to SYSBIOS and debugging using CCS. Once the application starts real-time processing (i.e. once sensor start is issued), there should ideally be no prints on the console because CCS will halt the processor (unless CIO is disabled) on which such prints are issued for as long as it takes it to transfer the print string data from target to PC over JTAG and print the string on the PC (which can be of the order of seconds). This is true for any real-time application that uses SYSBIOS on any SoC (not just mmWave SDK/devices). For logging in real-time, SYSBIOS offers other options like LOG module, etc - although these will incur some memory overheads. For example, see "Enable DebugP logs" section. It is also possible in cfg file of SYSBIOS based application to direct System_printfs to an internal log buffer (circular or saturate) which will also prevent the hiccup by CCS (See 'xdc.runtime. SysMin' in SYSBIOS/XDC).

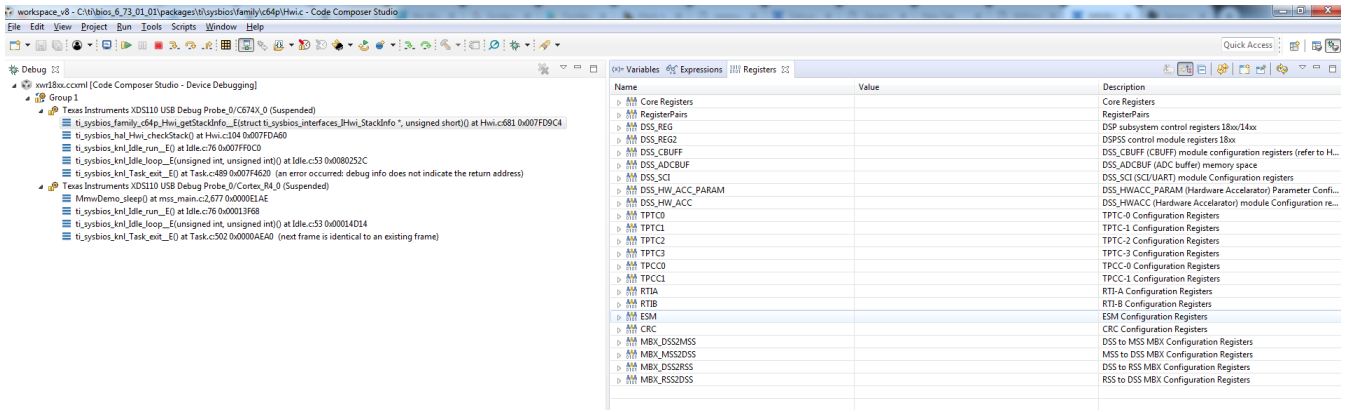
The out of box demos based on the DPC/DPU/DPM architecture have by default the DebugP type real-time logging enabled. In order to visualize the logs, the CCS feature of Run Time Object Viewer (ROV) can be used. Instructions of how to use this feature can be seen at http://processors.wiki.ti.com/index.php/Runtime_Object_View_(ROV). Below is a sample log for the xwr68xx out of box demo.



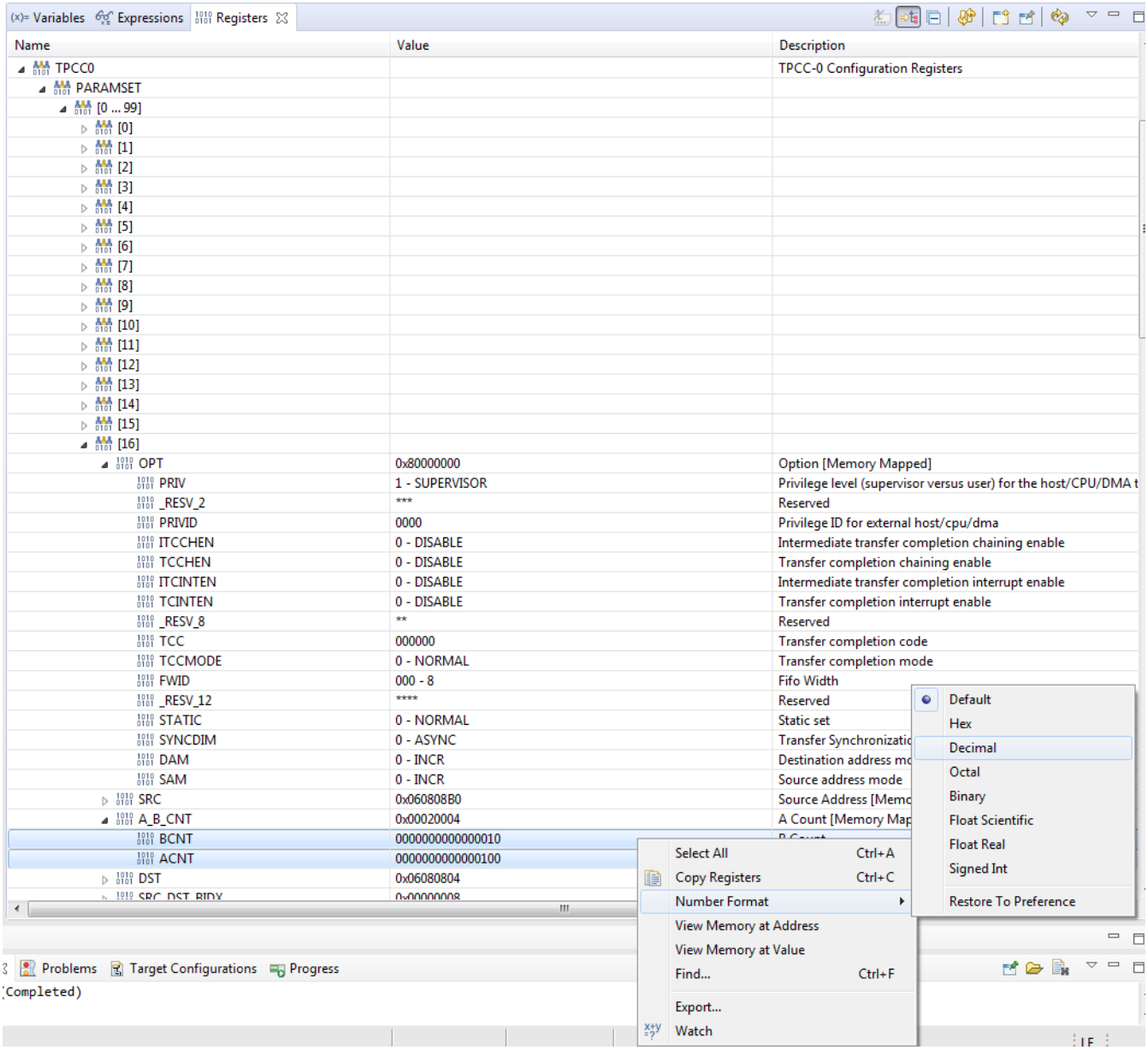
serial	timestampRaw	modName	text	eventId	eventName	arg0	arg
541653	1818035633765	xdc.runtime.Main	ObjDet DPC: Range Proc Triggered in export IOCTL	0	xdc.runtime.Log_print	10608	0
541654	1818035635796	xdc.runtime.Main	App: DPM Report IOCTL, command = 113	0	xdc.runtime.Log_print	29748	113
541655	1818050704736	xdc.runtime.Main	ObjDet DPC: Frame Start, frameIndx = 90245, subFrameIndx = 0	0	xdc.runtime.Log_print	65456	902
541656	1818050706365	xdc.runtime.Main	ObjDet DPC: Processing sub-frame 0	0	xdc.runtime.Log_print	32600	0
541657	1818051107533	xdc.runtime.Main	ObjDet DPC: Range Proc Done	0	xdc.runtime.Log_print	32640	0
541658	1818051349977	xdc.runtime.Main	ObjDet DPC: number of detected objects after CFAR = 34	0	xdc.runtime.Log_print	32672	34
541659	1818055712123	xdc.runtime.Main	ObjDet DPC: Range Proc Triggered in export IOCTL	0	xdc.runtime.Log_print	10608	0
541660	1818055714146	xdc.runtime.Main	App: DPM Report IOCTL, command = 113	0	xdc.runtime.Log_print	29748	113
541661	1818070704729	xdc.runtime.Main	ObjDet DPC: Frame Start, frameIndx = 90246, subFrameIndx = 0	0	xdc.runtime.Log_print	65456	902
541662	1818070706368	xdc.runtime.Main	ObjDet DPC: Processing sub-frame 0	0	xdc.runtime.Log_print	32600	0
541663	1818071107554	xdc.runtime.Main	ObjDet DPC: Range Proc Done	0	xdc.runtime.Log_print	32640	0
541664	1818071350197	xdc.runtime.Main	ObjDet DPC: number of detected objects after CFAR = 33	0	xdc.runtime.Log_print	32672	33
541665	1818075632797	xdc.runtime.Main	ObjDet DPC: Range Proc Triggered in export IOCTL	0	xdc.runtime.Log_print	10608	0
541666	1818075634827	xdc.runtime.Main	App: DPM Report IOCTL, command = 113	0	xdc.runtime.Log_print	29748	113
541667	1818090704729	xdc.runtime.Main	ObjDet DPC: Frame Start, frameIndx = 90247, subFrameIndx = 0	0	xdc.runtime.Log_print	65456	902
541668	1818090706368	xdc.runtime.Main	ObjDet DPC: Processing sub-frame 0	0	xdc.runtime.Log_print	32600	0
541669	1818091107547	xdc.runtime.Main	ObjDet DPC: Range Proc Done	0	xdc.runtime.Log_print	32640	0
541670	1818091350220	xdc.runtime.Main	ObjDet DPC: number of detected objects after CFAR = 34	0	xdc.runtime.Log_print	32672	34
541671	1818095711859	xdc.runtime.Main	ObjDet DPC: Range Proc Triggered in export IOCTL	0	xdc.runtime.Log_print	10608	0
541672	1818095713889	xdc.runtime.Main	App: DPM Report IOCTL, command = 113	0	xdc.runtime.Log_print	29748	113
541673	1818110704729	xdc.runtime.Main	ObjDet DPC: Frame Start, frameIndx = 90248, subFrameIndx = 0	0	xdc.runtime.Log_print	65456	902
541674	1818110706368	xdc.runtime.Main	ObjDet DPC: Processing sub-frame 0	0	xdc.runtime.Log_print	32600	0
541675	1818111107540	xdc.runtime.Main	ObjDet DPC: Range Proc Done	0	xdc.runtime.Log_print	32640	0
541676	1818111349106	xdc.runtime.Main	ObjDet DPC: number of detected objects after CFAR = 32	0	xdc.runtime.Log_print	32672	32
541677	1818115623861	xdc.runtime.Main	ObjDet DPC: Range Proc Triggered in export IOCTL	0	xdc.runtime.Log_print	10608	0

6. 15. 5. Viewing hardware registers

During debug, there may be a need to examine registers of HWA, EDMA, external I/O peripherals etc. These can be done using View->Registers menu and when a core is selected, the register view will display all registers that the core can see organized into various categories. An example is shown below:



Individual hardware entities can be expanded further in the view to see registers specific to the hardware entity. The following picture illustrates viewing a certain PARAM set in instance #0 of the EDMA (TPCC0), note how the bit fields are automatically parsed and displayed in a user friendly manner which saves the burden of manually parsing or developing special parsing tools and facilitates quick debugging. Default number formats of bit fields are binary which is not always convenient, this can be changed by selecting the field/fields and right-clicking to see the number format menu as shown in the example below where the A and B counts of EDMA are about to be chosen for Decimal format. Once chosen, the GUI will remember the user choice for that specific field so user does not have to repeat this action in future debug sessions.



In the above picture, one can also see the "Watch" menu item. If this is selected, then the two fields of interest will appear in the Expressions view, this is a convenient way to see some fields of interest during debug without having to navigate the register structure again (although when a particular structure such as PARAM set #16 above is expanded, if the top level TPCC0 is shrunk and expanded again, thePARAM #16 is shown expanded as before because GUI remembers sub-structure expansion/non-expansion state).

6. 15. 6. Viewing expressions/memory in real time

When debugging real time application (for example: mmw demo) in CCS, if the continuous refresh of variables in the Expression or Memory browser window is enabled without enabling the silicon real-time mode as shown in the picture, the code may crash at a random time showing the message in the console window. To avoid this crash, please put CCS in to "Silicone Real-time" mode after selecting the target core.

Continuous refresh:

Crash in Console window:

```
[C674X_0] Debug: Logging UART Instance @00815560 has been opened successfully
Debug: DSS Mailbox Handle @0080f550
Debug: MMWDemoDSS create event handle succeeded
Debug: MMWDemoDSS mmWave Control Initialization succeeded
Debug: MMWDemoDSS ADCBUF Instance(0) @00815530 has been opened successfully
Debug: MMWDemoDSS Data Path init succeeded
Debug: MMWDemoDSS initTask exit
[CortexR4_0] ******************************************
Debug: Launching the Millimeter Wave Demo
******************************************
Debug: MMWDemoMSS Launched the Initialization Task
Debug: MMWDemoMSS mmWave Control Initialization was successful
Debug: CLI is operational
Sensor has been stopped
Debug: MMWDemoMSS Received CLI sensorStart Event
[C674X_0] Heap L1 : size 16384 (0x4000), free 2816 (0xb00)
Heap L2 : size 49152 (0xc000), free 35368 (0x8a28)
Heap L3 : size 655360 (0xa0000), free 368640 (0x5a000)
A0=0x0 A1=0xffffff2e
A2=0x78 A3=0xffffffa8
A4=0xa7 A5=0x7a
A6=0x5a827999 A7=0x5a827999
A8=0xed A9=0x7fffffff
A10=0x2 A11=0xf00220
A12=0xf002a0 A13=0x0
A14=0x804be0 A15=0xf00200
A16=0x5a827999 A17=0xa57d8667
A18=0xffffff1b A19=0xfffffdf5
A20=0xfffffff0 A21=0xffffffaa
A22=0xa6 A23=0xffffff60
A24=0xfffffea7 A25=0xfffffedf
A26=0x114 A27=0x17
A28=0xffffff34 A29=0x0
A30=0x6 A31=0x0
B0=0x0 B1=0xffffff50
B2=0xffffffd5 B3=0x2
B4=0x0 B5=0x0
B6=0x93 B7=0xfffffecb
B8=0x0 B9=0xf00218
B10=0xffffffeee B11=0xffffffa7
```

C674x CPU Exception

Enable "Silicone Real-time" mode: