MMWAVE SDK User Guide

Product Release 1.0.0

Release Date: May 2, 2017

Document Version: 1.0

COPYRIGHT

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. System Overview

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

1. 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
- RADARSS Firmware
- Board Setup and Flash Utilities

1. 2. mmWave Demos

SDK provides a suite of 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"
- mmWave Data Capture/Streaming demo

1. 3. External Dependencies

The SDK depends on the following external components which are not part of the SDK package but will be needed to integrate the mmWave SDK.

- TI RTOS (or Custom RTOS)
- XDC Tools (if building for TI RTOS)
- CCS (for debugging)
- TI ARM and C674X compiler
- DSPLib and Mathlib

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

1. 4. Terms used in this document

Terms used	Comment
xWR14xx	This is used throughout the document where that section/component/module applies to both AWR14xx and IWR14xx
xWR16xx	This is used throughout the document where that section/component/module applies to both AWR16xx and IWR16xx
xWR1xxx	This is used throughout the document where that section/component/module applies to all the part: AWR14xx, IWR14xx, AWR16xx and IWR16xx
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. System Deployment

2. 1. xWR14xx

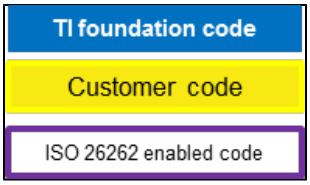
A typical mmWave application using xWR14xx would perform these operations:

- Control and monitoring of RF front-end through mmwaveLink
- External communications through standard peripherals
- Some radar data processing using FFT HW accelerator

Typical xWR14xx system deployments could be envisioned as follows:

1. **Autonomous xWR14xx sensor (aka Standalone mode)**
   a. xWR14xx program code is downloaded from the serial flash memory attached to xWR14xx (via QSPI)
   b. Optional high level control from remote entity
   c. Sends low speed data output (objects detected) to remote entity
2. **Hybrid xWR14xx sensor + Controller**
   a. Serial flash is attached/in-built to external controller and SPI interface exists between xWR14xx and controller
   b. High level control from controller (code download, GPIO toggling, etc)
   c. Sends low speed data output (objects detected) to controller.
3. **Satellite xWR14xx sensor + DSP**
   a. Program code is either in serial flash memory attached to xWR14xx (via QSPI) or downloaded via the control interface between xWR14xx and DSP (ex: via SPI)
   b. High level control from DSP
   c. Sends high speed data output (1D/2D FFT output) to DSP

These deployments are depicted in the Figure 1 and Figure 2.



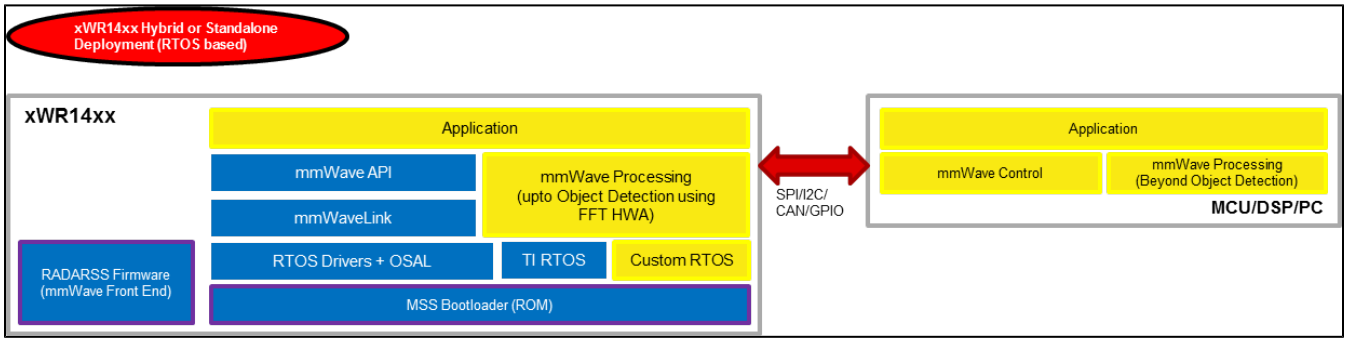


Figure 1: xWR14xx Deployment in Hybrid or Standalone mode

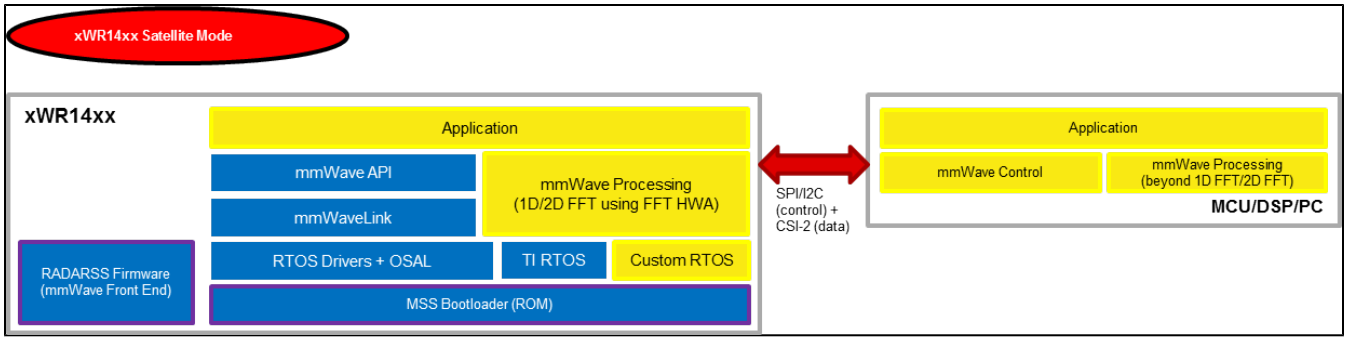


Figure 2: xWR14xx Deployment in Satellite mode

Note that the software architecture presented above demonstrates only the mmWave SDK components running on the external devices – MCU, DSP, PC. There are other software components running on those external devices which are part of the ecosystem of those devices and out of scope for this document. The mmWave SDK package would provide, in future, sample code for the mmWave API running on these external devices but the porting of this layer onto these external device ecosystem is the responsibility of system integrator/application code provider.

2. 2. xWR16xx

A typical xWR16xx 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 xWR16xx customer deployment is shown in Figure 3:

a. xWR16xx program code for MSS and DSP-SS is downloaded from the serial flash memory **attached** to xWR16xx (via QSPI)
b. **Optional** high level control from remote entity
c. Sends **low speed data** output (objects detected) to remote entity
d. **Optional** high speed data (debug) sent out of device over LVDS

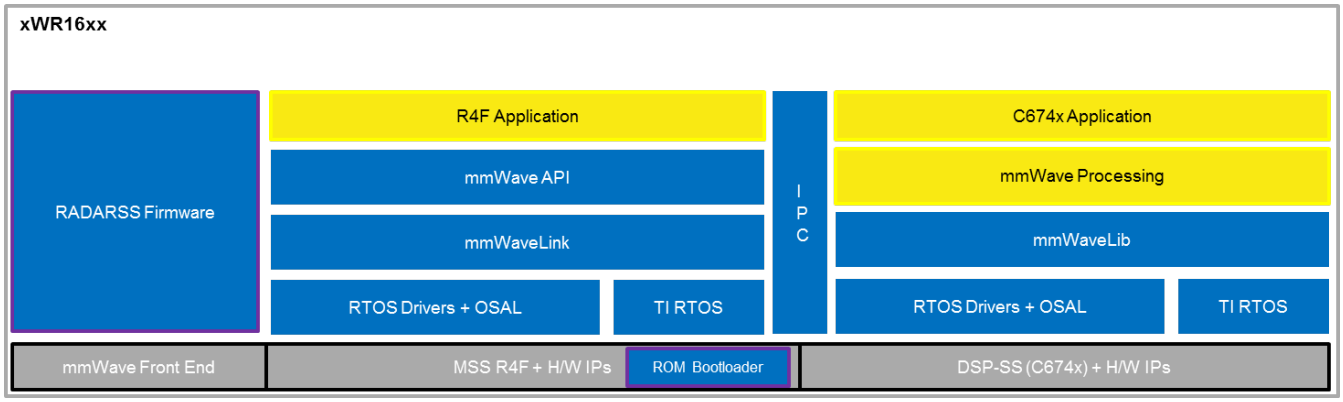


Figure 3: Autonomous xWR16xx sensor (Standalone mode)

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. The demos are placed at mmwave_sdk_<ver>/packages/ti/demo/<platform> folder. Currently following demos are supported within the SDK:

1. **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 xWR14xx/xWR16xx SoC using the drivers in the mmWave SDK (Software Development Kit). It allows user to specify the chirping profile and displays the detected objects and other information in real-time. 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. DS3 LED on TI EVM is turned on when the sensor is started successfully and turned off when the sensor is stopped successfully. SW1 switch press on TI EVM will start/stop the demo (sensor needs to be configured atleast once using the CLI).
2. **Capture Demo**: This demo is located at mmwave_sdk_<ver>/packages/ti/demo/<platform>/capture folder. The capture demo shows some of the radar sensing capabilities of the xWR14xx/xWR16xx SoC using the drivers in the mmWave SDK (Software Development Kit). It allows user to specify the chirping profile and on successful execution captures ADC data in the device's L3 memory. The data captured depends on the frame configuration provided and the amount of L3 memory that is available on the device. In xWR14xx, the data buffer in L3 is implemented as a circular buffer and memory will be overwritten if the frame configuration produces more data than that can fit in L3 memory. In xWR16xx, the data buffer in L3 is a linear buffer and capture stops when the buffer is full. Typically this demo should be used to capture only one frame worth of data by specifying number of frame parameter to be 1 in the cli's frameCfg command. Another constraint is that the interchirp time should be more than 10 usec to allow for DMA to copy the data from ADC buffer to L3 at every chirp available interrupt.
   This demo has additional commands to enable the streaming of ADC data over high speed interface (LVDS for xWR14xx/xWR16xx and CSI-2 for IWR14xx). In this streaming mode, the data is also copied (via DMA) into the L3 memory to verify that the data transferred on the LVDS/CSI2 interface matches the ADC samples generated on the mmWave sensor device. Note that this demo needs mmWave sensor device to be interfaced with another device that has a compatible LVDS/CSI2 interface.
   A detailed explanation of this demo is available in the demo's docs folder: mmwave_sdk_<ver>\packages\ti\demo\<platform>\capture\docs\doxygen\html\index.html

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

3. 1. Connecting the xWR14xx/xWR16xx EVM to PC

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 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 TI code composer studio is not installed on that PC, then XDS110 drivers need to be installed.
> 2. If TI code composer studio is installed, then version of CCS and emulation package need to be checked 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

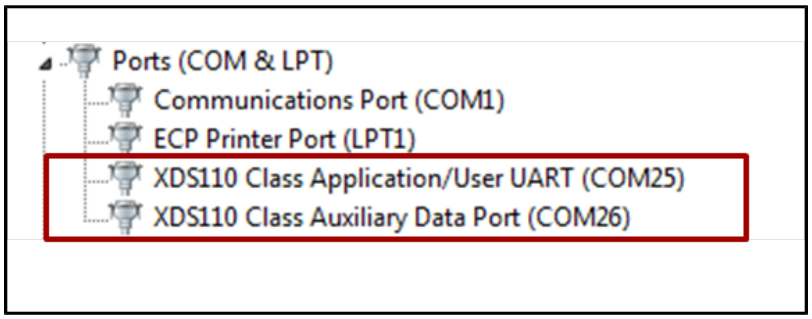


Figure 4: xWR14xx/xWR16xx PC Connectivity - Device Manager - COM Ports

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.

3. 2. Programming xWR14xx/xWR16xx

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

xWR16xx has 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 execuables 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 only exception to this is the Mailbox unit test named "test_mss_dss_msg_exchange" and mmWave unit tests under full and minimal.

3. 3. Loading images onto xWR14xx/xWR16xx EVM

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

3. 3. 1. Demonstration Mode

This mode should be used when either experimenting with 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 xWR14xx/xWR16xx EVM).
    a. For xWR16xx: use the mmwave_sdk_<ver>/packages/ti/demo/xwr16xx/<demo> /xwr16xx_<demo>.bin as the METAIMAGE1 file name.
    b. For xWR14xx: use the mmwave_sdk_<ver>/packages/ti/demo/xwr14xx/<demo> /xwr14xx_<demo>_mss.bin as the MSS_BUILD file name.
2. Remove the "SOP2" jumper 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. 3. 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 xWR1xxx 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 xWR1xxx development phase, 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 (xWR16xx only) directly via CCS in the xWR14xx/xWR16xx 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
    a. Follow the procedure mentioned in the section: How to flash an image onto xWR14xx/xWR16xx EVM.
        i. For xWR16xx: use mmwave_sdk_<ver>/packages/ti/utils/ccsdebug /xwr16xx_ccsdebug.bin as the METAIMAGE1 filename for the one-time flash.
        ii. For xWR14xx: use mmwave_sdk_<ver>/packages/ti/utils/ccsdebug /xwr14xx_ccsdebug_mss.bin as the MSS_BUILD filename for the one-time flash.
    b. Follow the steps in How to connect xWR14xx/xWR16xx EVM to CCS using JTAG to setup the environment for CCS connectivity.

2. With "SOP2" jumper removed, after every power cycle/reboot of the EVM, follow these steps to load the application:
    a. Power up the EVM
    b. Launch ccxml file created in step 1.b above.
    c. If the test requires an application to run on MSS
        i. Connect CCS to Cortex_R4_0
        ii. Load the MSS program. (for example: xwr16xx_<module>_mss.xer4f prebuilt executables provided in the SDK release package)
    d. If the test requires an application to run on DSP (xWR16xx only)
        i. Connect CCS to C674X_0
        ii. Load the DSS program. (for example: xwr16xx_<module>_dss.xe674 prebuilt executables provided in the SDK release package)
    e. Run the R4 and/or C674 cores
    f. To reload, disconnect the connected cores, power cycle and connect again

3. 4. Running the Demos

Assuming that you have loaded the right demo binary using the section above, set the EVM to functional mode and power up the device. Connect the EVM to the PC using its XDS110 micro-USB port/cable. When the USB is connected to the PC, the device manager should recognize the following COM ports as shown in the Figure above:

1. **XDS110 Class Auxiliary Data port** -> 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). Lets call this **visualization port** or **Data_port**.
2. **XDS110 Class Application/User UART** -> This is the port where **CLI (command line interface)** runs for all the various demos. Lets call this **CFG_port**.

3. 4. 1. mmWave Demo for xWR14xx/xWR16xx

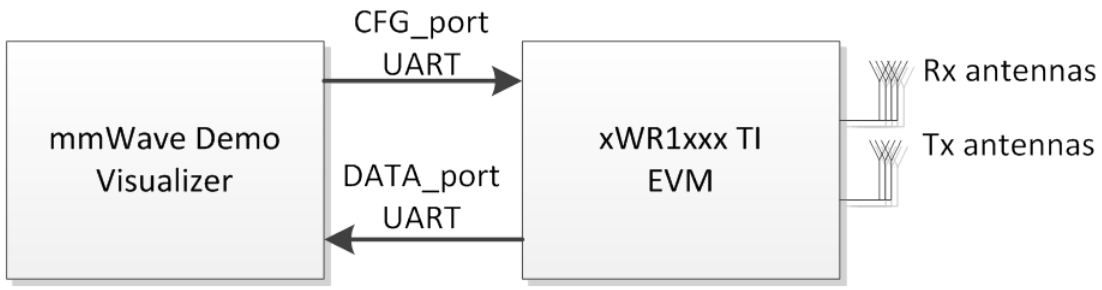


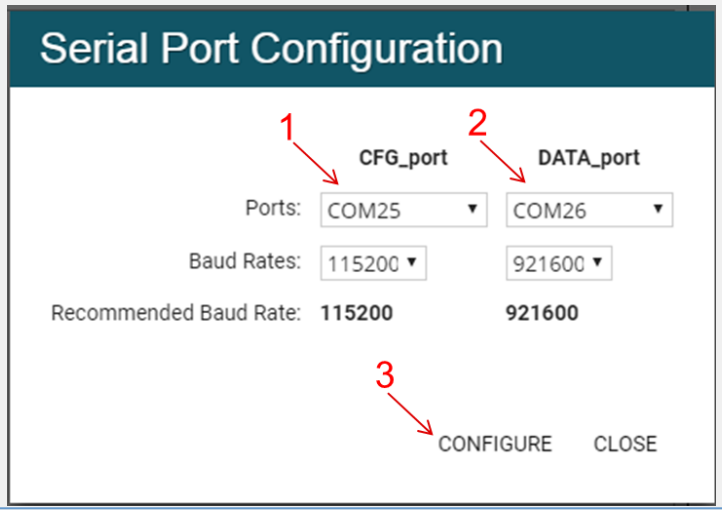
Figure 5: mmWave Demo Visualizer- mmWave Device Connectivity

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.

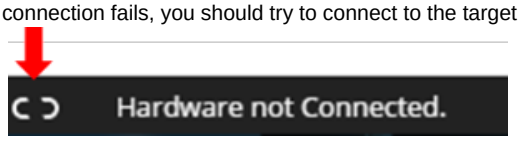
First Time Setup

a. 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.
b. Once the demo is running on the mmWave sensors and the USB is connected from the board to the PC, you need to configure the serial ports in this App. In the App, go to the Menu->Options->Serial Port.
   - **CFG_port**: Use COM port number for "XDS110 Class Application/User UART": Baud: 115200
   - **Data_port**: Use COM port "XDS110 Class Auxiliary Data port": Baud: 921600

> **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.



a. 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.



b. After the App is connected to the target, you can select the configuration parameters in this App (Frequency Band, Platform, etc) in the "Scene Selection" and "Object Detection" area of the **CONFIGURE** tab.
c. 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 1 fps. For selecting frame rate to be 25-30fps, for better GUI performance, select only the scatter plot and statistics plot.
d. Once the configuration is selected, you can send the configuration to the device (use "SEND CONFIG TO MMWAVE DEVICE" button).
e. 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.
f. 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.

3. If board is rebooted, follow the steps starting from 1 above.

> **COM port after reboot**
> 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. On TI gallery app "mmWave Demo Visualizer", use the bottom left serial port connection icon for disconnecting and reconnecting.

Inner workings of the GUI

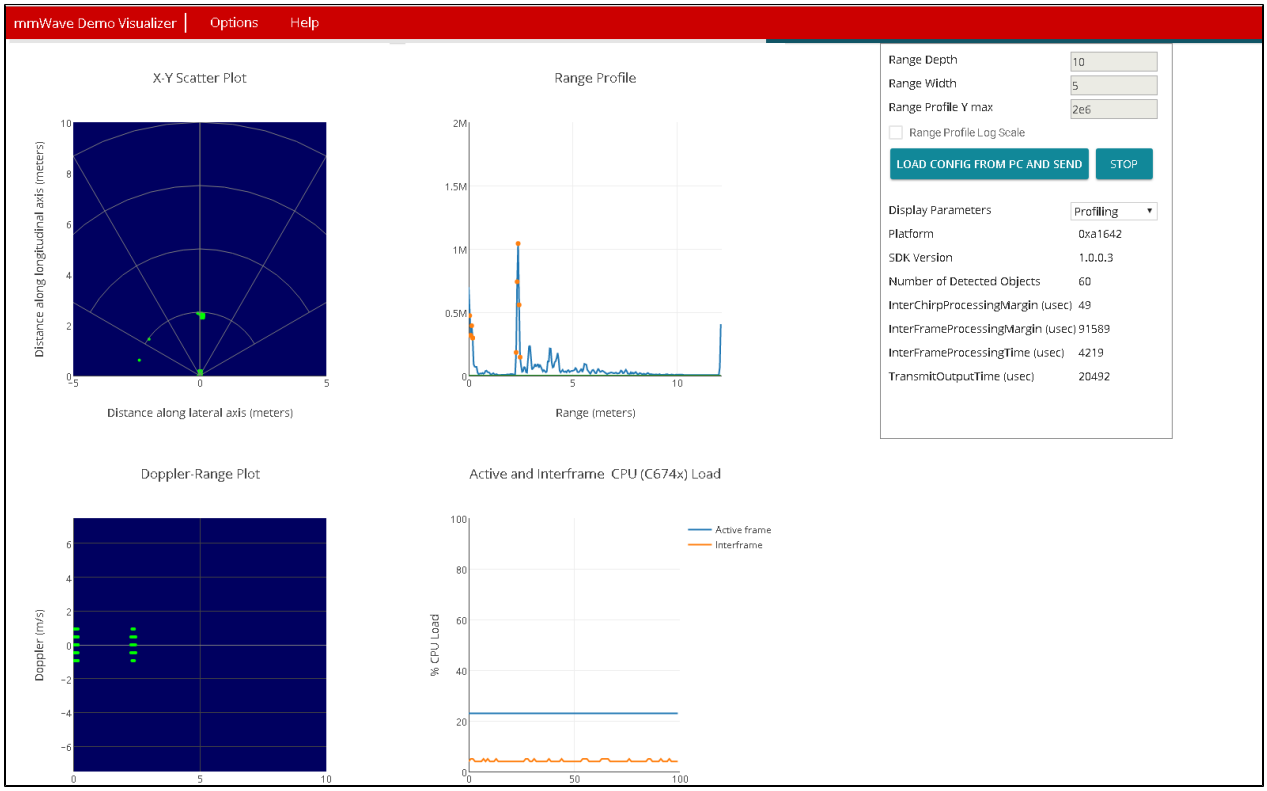
In the background, GUI performs the following steps:

- 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 'stopSensor' 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.

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.

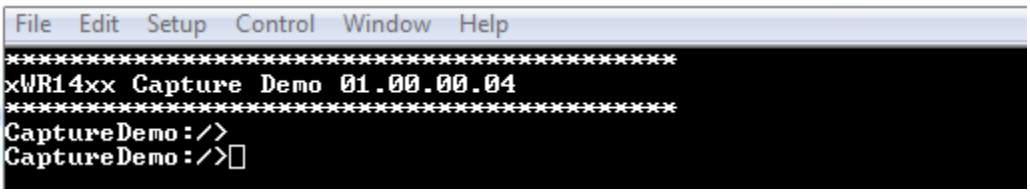
Here is an example of plots that mmWave Demo Visualizer produces based on the config that is passed to the demo application running on mmWave sensor.



3. 4. 2. Capture Demo for xWR14xx

> Note this demo requires connecting TI Code Composer studio to the xWR14xx EVM.
>
> If the demo enables streaming then the DevPack (MMWAVE-DEVPACK) is required to be connected to AWR14xx/IWR14xx EVM to access the LVDS interface and IWR14xx EVM to access the CSI-2 interface. See EVM User guide for details (See section "Connecting BoosterPack™ to LaunchPad™ or MMWAVE-DEVPACK").

1. Power on the xWR14xx EVM in functional mode, connect it to the PC with the USB cable. Open a teraterm or hyperterminal to connect to "User UART" COM port (see figure above).
2. The User UART COM port shows the demo cli as follows on power up. If you don't see this banner, hit 'enter' and you should atleast see the 'CaptureDemo:/>" prompt.



3. Next choose whether you want to run this demo in "**memory capture**" or "**streaming over LVDS/CSI**" mode and follow either step 4 or step 5 below. Note that only one of this mode can be selected per reboot. If streaming mode is selected, make sure a compatible device is connected to the other end of LVDS/CSI cable for capturing the stream of data.
4. **Memory capture procedure**
    a. Send the commands provided in file mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\capture_demo_script.txt to the CLI window. Refer to CFG Section for details on parameters used in this demo.

Capture demo configuration for xWR14xx

```
flushCfg
dfeDataOutputMode 1
channelCfg 2 1 0
adcCfg 2 2
lowPower 0 0
profileCfg 0 77 20 5 80 0 0  40 1 256 8000 0 0 30
chirpCfg 0 0 0 0 0 0 0 1
frameCfg 0 0 128 1 20 1 0
adcbufCfg 0 0 0 1
sensorStart
```

Teraterm window would like as follows:

```
File  Edit  Setup  Control  Window  Help
*******************************************
xWR14xx Capture Demo 01.00.00.04
*******************************************
CaptureDemo:/>
CaptureDemo:/>dfeDataOutputMode 1
Done
CaptureDemo:/>channelCfg 2 1 0
Done
CaptureDemo:/>adcCfg 2 2
Done
CaptureDemo:/>lowPower 0 0
Done
CaptureDemo:/>profileCfg 0 77 20 5 80 0 0  40 1 256 8000 0 0 30
Done
CaptureDemo:/>chirpCfg 0 0 0 0 0 0 0 1
Done
CaptureDemo:/>frameCfg 0 0 128 1 20 1 0
Done
CaptureDemo:/>adcbufCfg 0 0 0 1
Done
CaptureDemo:/>sensorStart
Done
CaptureDemo:/>
```

b. Without powering off the board, connect the board to CCS. (See section How to connect xWR14xx/xWR16xx EVM to CCS using JTAG for details on how to connect to CCS). Load the symbols for the capture demo application using executable file provided at mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\xwr14xx_capture_demo_mss.xer4f.

c. In CCS Expressions window, view the global variable gCaptureMCB and check DMA interrupt counter(dmaIntCounter) and chirp interrupt counter(chirpIntCounter) after sensorStart command. The chirpIntCounter should match the configuration. For the above configuration, the counter should be 128. The dmaIntCounter will be the actual number of DMA transfers triggered.

◢ gCaptureMCB	struct CaptureDemo_MCB_t	{cfg={cpuClockFrequency=200000000,loggingBaudR...	0x0800A598
▷ cfg	struct CaptureDemo_Cfg_t	{cpuClockFrequency=200000000,loggingBaudRate=...	0x0800A598
▷ socHandle	void *	0x08000038	0x0800A5F0
▷ loggingUartHandle	struct UART_Config *	0x0800AC7C {fxnTablePtr=0x0800AC24 {closeFxn=0x...	0x0800A5F4
▷ commandUartHandle	struct UART_Config *	0x0800AC70 {fxnTablePtr=0x0800AC24 {closeFxn=0x0...	0x0800A5F8
▷ adcbufHandle	struct ADCBuf_Config_t *	0x0800AD50 {fxnTablePtr=0x0001DAB0 {closeFxn=0x...	0x0800A5FC
▷ dmaHandle	void *	0x08000D80	0x0800A600
▷ edmaHandle	void *	0x0001DB0C	0x0800A604
▷ csiHandle	void *	0x08001758	0x0800A608
▷ cbuffHandle	void *	0x00000000	0x0800A60C
▷ ctrlHandle	void *	0x08009D3C	0x0800A610
▷ chirpSemHandle	void *	0x08000D60	0x0800A614
▷ chirpAvailable	void *	0x08000074	0x0800A618
(x)= hsi	enum Capture_HSI_e	Capture_HSI_NONE	0x0800A61C
(x)= numRxChannel	unsigned int	1	0x0800A620
(x)= dmaElemSize	unsigned short	1024	0x0800A624
(x)= dmaFrameCnt	unsigned char	1 '\x01'	0x0800A626
(x)= dmaState	unsigned short	0	0x0800A628
(x)= currDstAddr	unsigned int	1359216640	0x0800A62C
◢ stats	struct CaptureDemo_Stats_t	{chirpInterruptCounter=128,dmaIntCounter=128,cb...	0x0800A630
(x)= chirpInterruptCounter	unsigned int	128	0x0800A630
(x)= dmaIntCounter	unsigned int	128	0x0800A634
▷ cbuffStats	struct CBUFF_Stats_t	{numFrameStart=0,numFrameDone=0,numChirpDo...	0x0800A638

d. Data generated can be saved from CCS and analyzed offline.

i. Data is saved from L3 memory using global variable gDataCube. The size (CAPTURE_L3RAM_DATA_MEM_SIZE) is defined in mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\capture.h
Select Tools -> save memory in file ccs_data.dat with format 16-bit Hex TI style, select memory location as gDataCube and size according to capture.h definition.

ii. Run mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\ gui\capture_demo.m in Matlab. The script is just an example for offline analyzing. The users are encouraged to re-use or create their own processing algorithms for this data.

5. **Streaming over LVDS /CSI-2**

a. Send the commands provided in file mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\lvds_stream _demo_script.txt or mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\csi_stream _demo_script.txt to the CLI window. Refer to CFG Section for details on parameters used in this demo.

i. See sample commands below:

stream_demo_script.txt

```
/* CLI command script to have 1 frame with 128 chirps */
flushCfg
dfeDataOutputMode 1
channelCfg 2 1 0
adcCfg 2 2
adcbufCfg 0 0 0 1
lowPower 0 0
profileCfg 0 77 20 5 80 0 0  40 1 256 8000 0 0 30
chirpCfg 0 0 0 0 0 0 0 1
frameCfg 0 0 128 1024 20 1 0

setHSI   LVDS|CSI

sensorStart
```

ii. By default the demo does **"not stream"** data. The CLI command "`setHSI`" needs to be specified for selecting the right HSI interface.

Use the following command to select the LVDS as the High speed interface

Select LVDS xWR14xx CLI command

```
setHSI   LVDS
```

Use the following command to select CSI as the High speed interface

Select CSI IWR14xx CLI command

```
setHSI   CSI
```

iii. After "sensorStart" command is issued, the mmWave ADC samples are streamed out over the LVDS/CSI-2 interface onto the external device.

b. Without powering off the board, connect the board to CCS. (See section How to connect xWR14xx/xWR16xx EVM to CCS using JTAG for details on how to connect to CCS). Load the symbols for the capture demo application using executable file provided at mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\xwr14xx_capture_demo_mss.xer4f.

c. The demo maintains a global variable '`gStreamMCB`' which has all the configuration and run time information stored for the demo. The structure has a stats field which holds the Chirp Interrupt & DMA Interrupt counter. This should match the configuration which as per the above sample configuration should be set to 128. In addition to streaming it out, the demo copies all the ADC data to the L3 memory as well to enable matching the data generated on the xWR14xx with the streamed data captured on the external device. See table below for location of each chirp buffer in xWR14xx L3 memory for the profile/chirp config shown above .

Chirps	xWR14xx L3 buffer location
1	gDataCube
2	gDataCube +0x400
3	gDataCube +0x800
4	gDataCube +0xC00
5	gDataCube +0x1000

...	

Test configurations
On CSI: The demo has only been verified in Raw12 and Raw14 mode with the Data format set to CBUFF_DataFmt_ADC_DATA.

On LVDS: The demo has only been verified in the 16bit output format in DDR mode with the Data format set to CBUFF_DataFmt_ADC_DATA.

This demo has been tested against the configuration supplied with the release package. User may update the configuration (such as RX channels, TX ant, real/complex samples, chirp threshold, etc) but they may have to update the DMA configuration in the demo code as well to copy the right amount of samples to L3 or over LVDS/CSI-2 and/or update their post processing algorithm to match the format of samples collected.

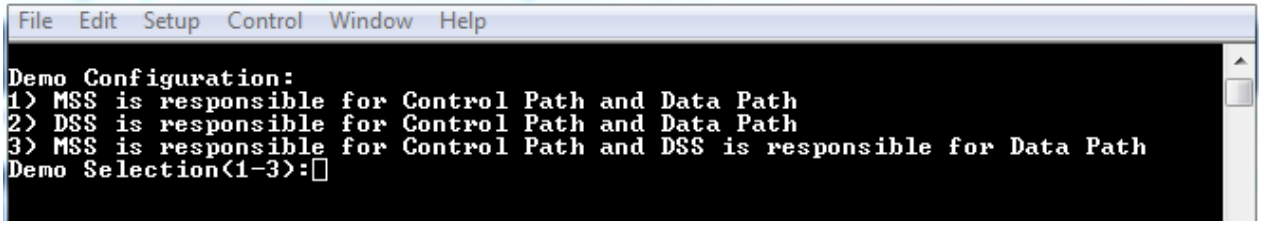
Streaming more than 1 channel
By default the streaming assumes that there is only 1 active receive channel. If the configuration has been modified then the CBUFF configuration needs to be updated to reflect all the channels which are active.

3. 4. 3. Capture demo for xWR16xx

Note this demo requires connecting TI Code Composer studio to the xWR16xx EVM.

If the demo enables streaming then the DevPack (MMWAVE-DEVPACK) is required to be connected to xWR16xx EVM to access the LVDS interface. See xWR16xx EVM User guide for details (See section "Connecting BoosterPack™ to LaunchPad™ or MMWAVE-DEVPACK").

1. Power on the xWR16xx EVM in functional mode, connect it to the PC with the USB cable. Open a teraterm or Hyperterminal to connect to "User UART" COM port (see figure above).
2. The User UART COM port shows Capture demo mode selection window as follows. If you don't see this menu, hit 'enter' :



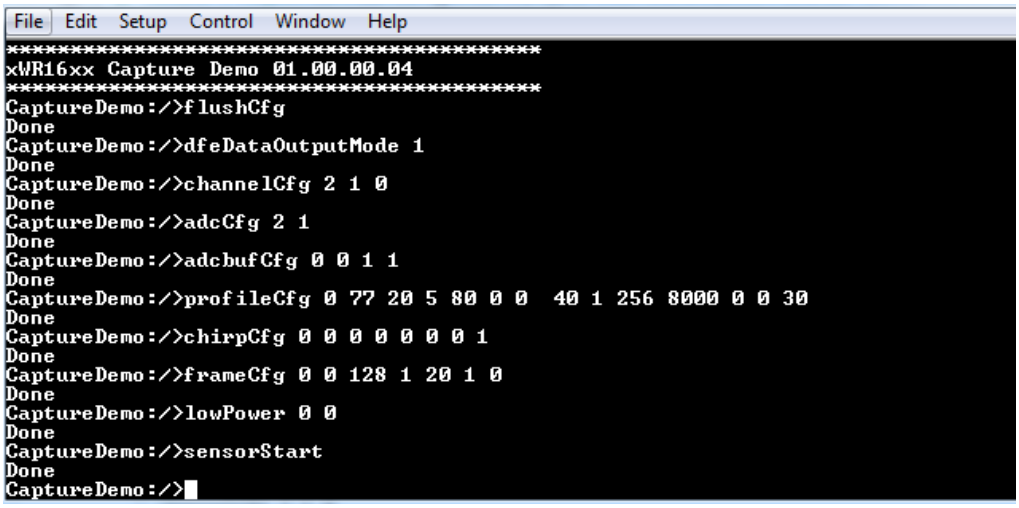
3. If mode 1 (MSS only) and 3 (MSS and DSS running in cooperative mode) are selected, the USER UART COM port shows demo configuration CLI as follows. For mode 2(DSS only mode), there is no configuration CLI. All configuration parameters are hard-coded in the code and are same as seen in the capture_demo_script.txt.

4. Next choose whether you want to run this demo in "**memory capture**" or "**streaming over LVDS**" mode and follow either step 5 or step 6 below. Note that only one of this mode can be selected per reboot. If streaming mode is selected, make sure a compatible device is connected to the other end of LVDS cable for capturing the stream of data.
5. **Memory capture procedure**
    a. Send the commands provided in file(capture_demo_script.txt) to the CLI window. Refer to Configuration table for details on parameters used in this demo.

Capture demo configuration

```
flushCfg
dfeDataOutputMode 1
channelCfg 2 1 0
adcCfg 2 1
adcbufCfg 0 0 1 1
profileCfg 0 77 20 5 80 0 0  40 1 256 8000 0 0 30
chirpCfg 0 0 0 0 0 0 0 1
frameCfg 0 0 128 1 20 1 0
lowPower 0 0
sensorStart
```

Teraterm window would like as follows:



b. Without powering off the board, connect the board to CCS. (See section How to connect xWR14xx/xWR16xx EVM to CCS using JTAG for details on how to connect to CCS). Load the symbols for the capture demo application using executable file provided at mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\capture. Use .xer4f for loading symbols onto Cortex_R4_0 and .xe674 for loading symbols onto DSP core C674X_0.

c. In CCS Expressions window, view the global variable gCaptureMCB and check DMA interrupt counter(dmaIntCounter) and chirp interrupt counter(chirpIntCounter) after sensorStart command. The chirpIntCounter should match the configuration. For the above configuration, the counter should be 0x80. The dmaIntCounter will be the actual number of DMA transfers triggered.

> The stats(counters) should be checked on the Core that runs "data path".
>
> Mode 1: check on R4F (Cortex_R4_0)
>
> Mode 2 and Mode 3: check on DSP (C674X_0)

gCaptureMCB	struct Capture_MCB	{...}
cfg	struct Capture_Cfg	{...}
ctrlHandle	void *	0x0800A2C0
eventHandle	struct ti_sysbios_knl_Event_Object *	0x08000C70
peerMailbox	void *	0x00000000
mboxSemHandle	struct ti_sysbios_knl_Semaphore_Object *	0x00000000
loggingUartHandle	struct UART_Config *	0x0800C134
commandUartHandle	struct UART_Config *	0x0800C128
socHandle	void *	0x08000038
dataPathObj	struct Capture_DataPathObj_t	{...}
state	enum Capture_STATE_e	Capture_STATE_STARTED
stats	struct Capture_STATS_t	{...}
configEvt	unsigned char	0x00
startEvt	unsigned char	0x00
stopEvt	unsigned char	0x00
cliSensorStartEvt	unsigned char	0x01
cliSensorStopEvt	unsigned char	0x00
chirpIntCounter	unsigned int	0x00000080
frameStartIntCounter	unsigned int	0x00000001
dmaIntCounter	unsigned int	0x00000080
chirpEvt	unsigned int	0x00000000
frameStartEvt	unsigned int	0x00000000
frameTrigEvt	unsigned int	0x00000001

d. Data generated can be saved from CCS and analyzed offline.
e. Data is saved from L3 memory, base address(gFrameAddress[0]) .
   Select Tools -> save memory in file ccs_data.dat with format 16-bit Hex TI style, select memory location and size according to test configuration.
   For the configuration from sample script, address is gFrameAddress[0], data size is 0x20000.
f. Run mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\capture\gui\capture_demo.m in Matlab. The script is just an example for offline analyzing.
   The users are encouraged to re-use or create their own processing algorithms for this data.

6. **Streaming over LVDS** :
   a. **Configurations**:
      i. For mode 1 (MSS only) and mode 3 (MSS and DSS running in cooperative mode): by default the Capture demo does not stream data. In order for the demo to stream out the data via the LVDS High speed interface enter the additional CLI "setHSI LVDS" command. A configuration file capture_demo_script_lvds.txt is provided for reference.

Enable LVDS streaming (capture_demo_script_lvds.txt)

```
flushCfg
dfeDataOutputMode 1
channelCfg 2 1 0
adcCfg 2 1
adcbufCfg 0 0 1 1
lowPower 0 0
profileCfg 0 77 100 5 80 0 0  40 1 256 8000 0 0 30
chirpCfg 0 0 0 0 0 0 0 1
frameCfg 0 0 128 1024 100 1 0
setHSI LVDS
sensorStart
```

The LVDS configuration used in the test is as follows:- [**NOTE:** This can be modified by changing the parameters in the Demo `CBUFF_open` invocation]

1. LVDS lanes is 2
2. 16bit Output format
3. DDR Clock Mode

Streaming more than 1 channel
By default the streaming assumes that there is only 1 active receive channel. If the configuration has been modified then the CBUFF configuration needs to be updated to reflect all the channels which are active.

Test Configurations

This demo has been tested against the configuration supplied with the release package. User may update the configuration (such as RX channels, TX ant, real/complex samples, chirp threshold, etc) but they may have to update the DMA configuration in the demo code as well to copy the right amount of samples to L3 or over LVDS/CSI-2 and/or update their post processing algorithm to match the format of samples collected.

The demo also supports continuous mode. The demo directory has the capture_demo_script_lvds_cont_mode.txt file which has a list of the CLI commands used for configuring the system in continuous mode.

ii. For mode 2(DSS only mode), as there is no configuration CLI: by default streaming is not enabled from the DSS; but users can modify the source file and set the global flag to enable streaming.

DSS only mode: Enable LVDS streaming

```
gCaptureMCB.cfg.streamCfg.enableHighSpeedInterface = 1U;
```

b. If a compatible device is connected to the other end of the LVDS, data will be streamed out over LVDS lanes and can be collected on that compatible device.
c. You can follow the CCS procedure from the "Memory capture" mode to look at internal state/variables.

3. 5. 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) and shown with the profile_2d.cfg configuration as an example below. Note that some of the commands (ex: guiMonitor) are available for mmWave Demo only.

profile cfg file (profile_2d.cfg for xWR16xx)

```
sensorStop
flushCfg
dfeDataOutputMode 1
channelCfg 15 3 0
adcCfg 2 1
adcbufCfg 0 0 1 1
profileCfg 0 77 7 7 58 0 0  68 1 256 5500 0 0 30
chirpCfg 0 0 0 0 0 0 0 1
chirpCfg 1 1 0 0 0 0 0 2
frameCfg 0 1 32 0 100 1 0
lowPower 0 0
guiMonitor 1  1 1 0 0 1
cfarCfg 0 2 8 4 4 0 5120
cfarCfg 1 0 8 4 4 0 5120
peakGrouping  1 0 0 1 224
multiObjBeamForming 1 0.5
calibDcRangeSig 0 -5 8 256
sensorStart
```

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.

Configuration	Parameters	Values	Comments	Values	Comments
		mmWave Demo: Example Profile for 2D detection (xWR16xx)		**Capture/Stream Demo: Example Profile**	
dfeDataOutputMode	1 - frame based chirps 2 - continuous chirping	1	frame based chirps	1	frame based chirps
channelCfg	(see mmwavelink doxgen for details)				
	Receive antenna mask e.g for 4 antennas, it is 0x1111b = 15	15	rx1, rx2, rx3, rx4	2	rx2 (See note "Streaming more than 1 channel" above)

	Transmit antenna mask For xWR1443 ▪ 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) For xWR1642 ▪ The 2 azimuth antennas can be enabled using bitmask 0x3 (i.e. tx1 and tx2)	3	tx1, tx2	1	tx1
	SoC cascading, not applicable, set to 0	0	N/A	0	N/A
adcCfg	(see mmwavelink doxgen for details)				
	Number of ADC bits (0 for 12-bits, 1 for 14-bits and 2 for 16-bits)	2	16-bit For mmW Demo, set this to 16-bits	2	16-bit
	Output format : 0 - real 1 - complex 1x (image band filtered output) 2 - complex 2x (image band visible))	1	complex 1x - image band filtered For mmW Demo, set this to complex	1	complex 1x - image band filtered
adcbufCfg					
	Input sample format: 0 - complex 1 - real	0	complex For mmW Demo, set this to complex	0	complex format.
	IQ swap selection: 0 - I in LSB, Q in MSB 1 - Q in LSB, I in MSB	0	For mmW Demo, set this to 0	0	
	Input sample format with respect to multiple antennas: 0 - interleaved (not supported on xWR16xx) 1 - non-interleaved	1	Must be 1 for xWR16xx. Use '0' for xWR14xx for mmW Demo.	1	Must be 1 for xWR16xx
	Chirp threshold, number of chirps for ping/pong buffer to trigger ping/pong buffer switch.	1	Use '1' for mmW Demo	1	
profileCfg	(see mmwavelink doxgen for details)				
	profile Identifier	0		0	
	start frequency in GHz	77	GHz	77	GHz
	idle time in u-sec	7	usec	20	usec
	ADC start time in u-sec	7	usec	5	usec
	Ramp end time in u-sec	58	usec	80	usec
	Tx output power back-off code for tx antennas	0		0	
	tx phase shifter for tx antennas	0		0	
	frequency slope constant	68 MHz/u-sec	total bandwidth = 68 * 58 = 3.944 GHz	40	total bandwidth = 40 * 80 = 3.200 GHz

	tx start time in u-sec	1		1	
	number of ADC samples	256	46.54 usec worth of samples (256/5500 ksps (next parameter)); sampled bandwidth = 68 * 46.54 = 3.16 GHz	256	46.54 usec worth of samples (256/5500 ksps (next parameter)); sampled bandwidth = 40 * 46.54 = 1.86 GHz
	ADC sampling frequency in ksps	5500	ksps	8000	
	HPF1 (High Pass Filter 1) corner frequency 0: 175 KHz 1: 235 KHz 2: 350 KHz 3: 700 KHz	0		0	
	HPF2 (High Pass Filter 2) corner frequency 0: 350 KHz 1: 700 KHz 2: 1.4 MHz 3: 2.8 MHz	0		0	
	rx gain in dB (valid values 24 to 48)	30	dB	30	dB
chirpCfg #0	(see mmwavelink doxgen for details)				
	chirp start index	0		0	
	chirp end index	0		0	
	profile identifier	0	use profile 0	0	use profile 0
	start frequency variation in Hz	0		0	
	frequency slope variation in Hz	0		0	
	idle time variation in u-sec	0		0	
	ADC start time variation in u-sec	0		0	
	tx antenna enable mask (Tx2,Tx1) e.g (10)b = Tx2 enabled, Tx1 disabled. See note under "Channel Cfg" command above.	1	enable Tx1 only	1	enable Tx1 only
chirpCfg #1	(see mmwavelink doxgen for details)			NA	
	chirp start index	1			
	chirp end index	1			
	profile identifier	0	use profile 0		
	start frequency variation	0			
	frequency slope variation	0			
	idle time variation	0			
	ADC start time variation	0			
	tx antenna enable mask	2	enable TX2 only (xWR16xx)		
lowPower	(see mmwavelink doxgen for details)				

	Analog chain configuration 0 : Complex Chain 1 : Real Chain	0	Only complex chain is supported	0	Complex chain
	ADC Mode 0 : Regular ADC mode 1 : Low power ADC mode	0	Regular ADC mode	0	Regular ADC mode
frameCfg	(see mmwavelink doxgen for details)				
	chirp start index (0-511)	0	Alternating chirp #0 and chirp #1	0	chirp #0
	chirp end index (chirp start index-511)	1	Alternating chirp #0 and chirp #1	0	
	number of loops (1 to 255)	32	32 times: 64 chirps in total	128	128 times (128 chirps)
	number of frames (valid range is 0 to 65535, 0 means infinite)	0	infinite	1	one frame
	frame periodicity in ms	100	100 ms	20	20 ms
	trigger select 1: Software trigger 2: Hardware trigger.	1	Software trigger	1	Software trigger
	Frame trigger delay in ms	0	0 ms	0	0 ms
guiMonitor				Not applicable	
	All parameters below are flags (1 to enable and 0 to disable)				
	detected objects 1 - enable export of detected objects 0 - disable	1	Send detected objects		
	log magnitude range 1 - enable export of log magnitude range profile at zero Doppler 0 - disable	1	Send log-magnitude of range		
	noise profile 1 - enable export of log magnitude noise profile 0 - disable	1	Send noise profile		
	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)	0	Do not send range-azimuth heat map		
	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	0	Do not send range-doppler heat map		
	statistics (CPU load, margins, etc) 1 - enable export of stats data. 0 - disable	1	Send statistics information		
cfarCfg				Not applicable	

	Processing direction: 0 – CFAR detection in range direction 1 – CFAR detection in Doppler direction (applies to xWR16xx only)	<0|1>	2 separate commands need to be sent; one for Range and other for doppler (xWR16xx only)		
	CFAR averagining mode (<mode>): 0 - CFAR_CA (Cell Averaging) 1 - CFAR_CAGO (Cell Averaging Greatest Of) 2 - CFAR_CASO (Cell Averaging Smallest Of)	0	CFAR-CA		
	noise averaging window length (<noiseWin>) Length of the noise averaged cells in samples	8	units: samples		
	guard length in samples	4	units: samples		
	Cumulative noise sum divisor expressed as a shift <divShift>. Sum of noise samples is divided by 2^<divShift>. Based on platform, <mode> and <noiseWin> , this value should be set as: CFAR_CA: <divShift> = $\log_2$(2 x) CFAR_CAGO/_CASO: xWR14: <divShift> = $\log_2$(<noiseWin>) xWR16: <divShift> = $\log_2$(2 x <noiseWin>)	4	In this example, noise sum is divided by 2^4=16 to get the average of noise samples with window length of 8 samples in CFAR -CA mode. 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.		
	cyclic mode or Wrapped around mode. 0- Disabled 1- Enabled xWR16xx: This control is not supported on xWR16xx, where it is always enabled in CFAR detection in Doppler direction and always disabled in CFAR detection in range direction. xWR14xx: used for programming the CFAR engine inside HWA	0			
	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 for xWR14xx, or Q8 format for xWR16xx. The threshold value can be converted from the value expressed in dB as For AWR14xx: $T_{cli} = 512 \times T_{dB} / 6$ For AWR16xx: $T_{cli} = 256 \times$ numVirtualAntennas $\times T_{dB} / 6$. Note: log input is used for both xWR14xx and xWR16xx mmw demo	5000			
peakGrouping	With peak grouping scheme enabled, instead of reporting a cluster of detected neighboring points, only one point, the highest one, will be reported, this reducing the total number of detected points per frame. Two different schemes are implemented in AWR16xx, and one in AWR14xx.			Not applicable	

	scheme 1 – MMW_PEAK_GROUPING_DET_MATRIX_BASED Peak grouping is based on peaks of the neighboring bins read from detection matrix. CFAR detected peak is reported if it is greater than its neighbors, located in detection matrix. 2 – MMW_PEAK_GROUPING_CFAR_PEAK_BASED Peak grouping is based on peaks of neighboring bins that are CFAR detected. CFAR detected peak is reported if it is greater than its neighbors, located in the list of CFAR detected peaks. This scheme is implemented only in xWR16xx. For more detailed look at mmw demo's doxygen documentation.	1			
	peak grouping in Range direction: 0 - disabled 1 - enabled	1			
	peak grouping in Doppler direction: 0 - disabled 1 - enabled	1			
	Start Range Index Minimum range index of detected object to be sent out.	1	Skip 0th bin and start peak grouping from range bin#1		
	End Range Index Maximum range index of detected object to be sent out.	(Range FFT size -1)	Skip last bin and stop peak grouping at (Range FFT size -1)		
multiObjBeamForming	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.			Not applicable	
	Enabled 0 - disabled 1 - enabled	1			
	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.	0.5			
calibDcRangeSig	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 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.			Not applicable	
	Enable DC removal using first few chirps 0 - disabled 1 - enabled	0			

	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.	-5	Last 5 bins		
	positive Bin Index (to remove DC from closest range bins) Maximum positive range FFT index to be included for compensation	8	First 8 bins		
	number of chirps to average to collect DC signature (which will then be applied to all chirps beyond this). The value must be power of 2, and also in xWR14xx, it must be greater than the number of Doppler bins.	256	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.		
setHSI	High speed Interface name over which the data is streamed out	Not applicable		LVDS CSI	Stream out the data over the selected High Speed Interface i.e. LVDS (xWR16xx or xWR14xx) or CSI (IWR14xx only)
contModeCfg	(see mmwavelink doxgen for details)	Not applicable			
	startFreq (start frequency in GHz)			77	
	txOutPower (Tx output power back-off code for tx antennas)			0	
	txPhaseShifter (tx phase shifter for tx antennas)			0	
	digOutSampleRate (ADC sampling frequency in ksps)			8000	
	hpfCornerFreq1 HPF1 (High Pass Filter 1) corner frequency 0: 175 KHz 1: 235 KHz 2: 350 KHz 3: 700 KHz			0	
	hpfCornerFreq2 HPF2 (High Pass Filter 2) corner frequency 0: 350 KHz 1: 700 KHz 2: 1.4 MHz 3: 2.8 MHz			0	
	rxGain rx gain in dB (valid values 24 to 48)			30	
	txEnable tx antenna enable mask			1	
	numSamples number of ADC samples			1024	

sensorStart	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. Optionally, user can provide an argument 'doReconfig' 1 - Do full reconfiguration of the device 0 - Skip reconfiguration and just start the sensor using already provided configuration.				
sensorStop	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.				
flushCfg	This command **should** be issued after 'sensorStop' command to flush the old configuration and provide a new one.				
%	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. 6. 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 flash an image onto xWR14xx/xWR16xx EVM

You will need the mmWave Device TI EVM, USB cable and a Windows 7 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 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 place jumpers on pins marked as SOP2 and SOP0 .

SOP2 jumper	SOP1 jumper	SOP0 jumper	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, use the TI Uniflash tool available at http://www.ti.com/tool/UNIFLASH and follow the instructions described in the mmWave SDK document "**UniFlash User Guide for mmWave Devices**" located at http://processors.wiki.ti.com/index.php/Category:CCS_UniFlash.
   Select the following images in the Uniflash tool.
   a. **xWR16xx:**
      For the SDK packaged xWR16xx demos and ccsdebug utility, there is a bin file provided in their respective folder: xwr16xx_<demo|ccsdebug>.bin which is the metaImage to be used for flashing. The metaImage already has the MSS, BSS (RADARSS) and DSS application combined into one file. Users can use this for flashing their own metaImage as well.
      i. For demo mode, either mmwave_sdk_<ver>\ti\demo\xwr16xx\mmw\xwr16xx_mmw_demo.bin or mmwave_sdk_<ver>\ti\demo\xwr16xx\capture\xwr16xx_capture_demo.bin should be selected.
      ii. For CCS development mode, mmwave_sdk_<ver>\ti\ utils\ccsdebug\xwr16xx_ccsdebug.bin should be selected.
   b. **xWR14xx**:
      For correct operation of mmWave SDK demos, this utility needs 2 binary images:
      i. BSS **:** mmwave_sdk_<ver>\firmware\radarss\xwr12xx_xwr14xx_radarss.bin
      ii. MSS **:** For the SDK packaged xWR14xx demos and ccsdebug utility, there is a bin file provided in their respective folder
      For demo mode, choose from **one** of these binary files from the SDK:
         1. mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\mmw\xwr14xx_mmw_demo_mss.bin
         2. mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\capture\xwr14xx_capture_demo_mss.bin
      For CCS development mode, mmwave_sdk_<ver>\ti\ utils\ccsdebug\xwr14xx_ccsdebug_mss.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.

4. 2. How to erase flash on xWR14xx/xWR16xx EVM

1. Setup the Booster Pack EVM for flashing as mentioned in step 1 of the section: How to flash an image onto xWR14xx/xWR16xx 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 xWR14xx/xWR16xx EVM

4. 3. How to connect xWR14xx/xWR16xx 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. 3. 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. 3. 2. Device support package Update

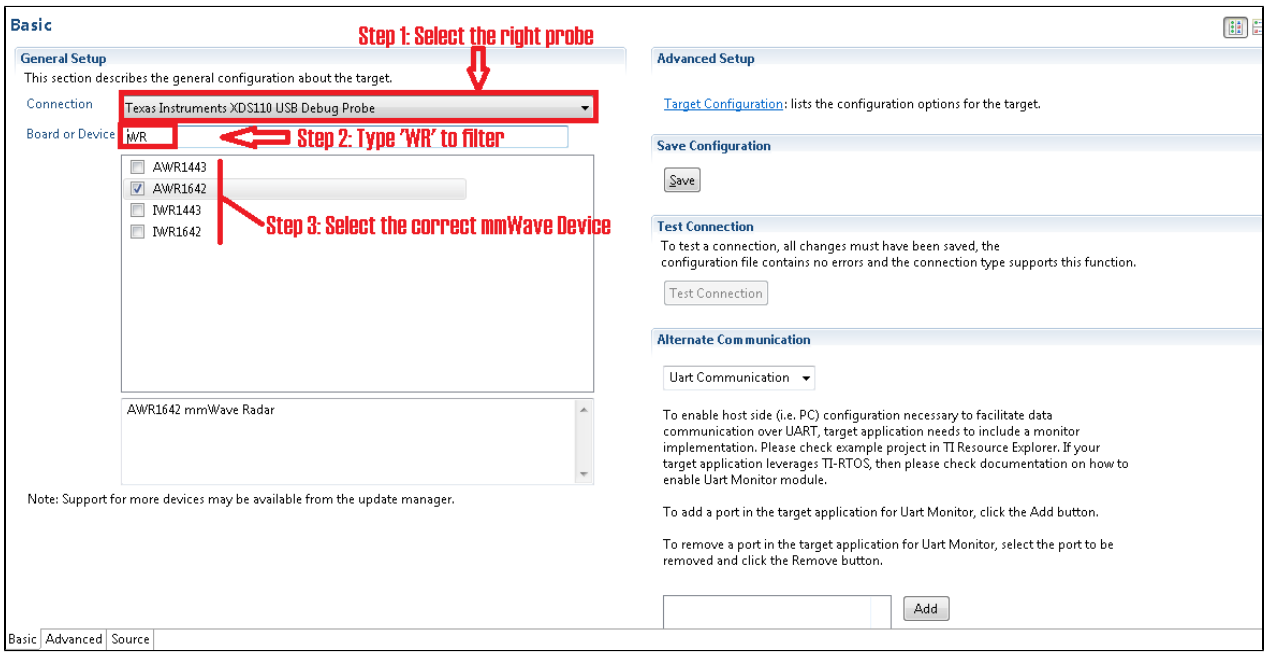
To create the ccxml file for connecting to the EVM, you will need to first update the device support package within CCS. Refer to the mmWave SDK release notes for the device support package version that would be needed within CCS to connect to the EVM. Check if that particular or its later version of "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. 3. 3. Target Configuration file for CCS (CCXML)

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

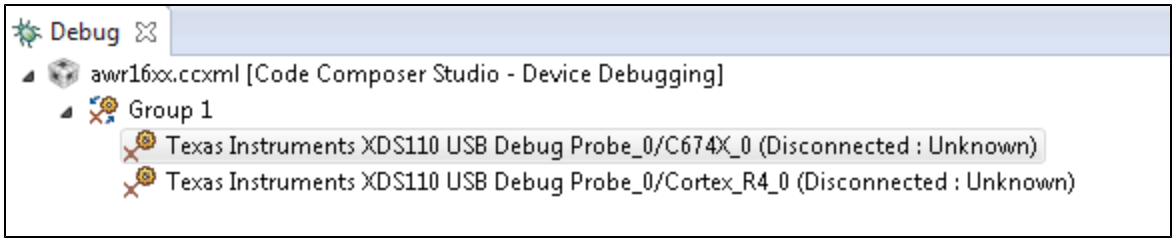
Assuming you have updated the device support package and Emulation pack as mentioned in the section above, follow the steps mentioned below to create a target configuration file in CCS.

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 AWR1642 or AWR1443 and check the box. Click Save and the file will be created.



4. 3. 3. 2. Connecting to xWR14xx/xWR16xx 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".


Debug
awr16xx.ccxml [Code Composer Studio - Device Debugging]
Group 1
Texas Instruments XDS110 USB Debug Probe_0/C674X_0 (Disconnected : Unknown)
Texas Instruments XDS110 USB Debug Probe_0/Cortex_R4_0 (Disconnected : Unknown)

4. 4. Developing using SDK

4. 4. 1. Build Instructions

Follow the mmwave_sdk_release_notes instructions to install the mmwave_sdk in your development environment (windows or linux). Install all the tools versions mentioned in the mmwave_sdk_release_notes for the particular release. Most tools have their separate installer (windows or linux) and installation procedure should be straightforward.

4. 4. 2. Setting up build environment

4. 4. 2. 1. Windows

1. Create command prompt at <mmwave_sdk_<ver> install path>\packages\scripts\windows folder. Set the paths shown below as per your tools installation location. Place these commands in a batch file setenv_tools.bat and run **setenv_tools.bat**.

setenv_tools.bat

```
@REM Select your device. Options (case sensitive) are: awr14xx,
iwr14xx, awr16xx, iwr16xx
set MMWAVE_SDK_DEVICE=awr16xx

@REM Common settings for awr14xx, awr16xx, iwr14xx and iwr16xx
@REM TI ARM compiler
set
R4F_CODEGEN_INSTALL_PATH=C:/ti/ccsv7/tools/compiler/ti-cgt-arm_16.9.1
.LTS
@REM Path to <mmwave_sdk installation path>/packages folder
set MMWAVE_SDK_INSTALL_PATH=c:/ti/mmwave_sdk_<ver>/packages
@REM TI XDC
set XDC_INSTALL_PATH=c:/ti/xdctools_3_50_00_10_core
@REM TI BIOS
set BIOS_INSTALL_PATH=C:/ti/bios_6_50_01_12/packages
@REM perl
set PERL_INSTALL_PATH=C:/Strawberry/perl/bin
@REM if using CCS to download, set below define to yes else no
set DOWNLOAD_FROM_CCS=yes
@REM install from web (free s/w). skip if doxygen output is not needed
@REM set DOXYGEN_INSTALL_PATH=C:/ti/doxygen

@REM Following only needed for awr16xx and iwr16xx
@REM TI DSP compiler
set
C674_CODEGEN_INSTALL_PATH=C:/ti/ccsv7/tools/compiler/ti-cgt-c6000_8.1
.3
@REM DSPlib
set C64Px_DSPLIB_INSTALL_PATH=C:/ti/dsplib_c64Px_3_4_0_0
@REM MATHlib
set C674x_MATHLIB_INSTALL_PATH=C:/ti/mathlib_c674x_3_1_2_1
@REM awr16xx/iwr16xx radarss firmware. Use the RPRC formatted binary
file.
set
XWR16XX_RADARSS_IMAGE_BIN=%MMWAVE_SDK_INSTALL_PATH%/../firmware/radar
ss/xwr16xx_radarss_rprc.bin
```

Even though the build is in Windows environment the paths have to use forward slashes "/" in the paths shown above

Path setting for TI tools is done by mmwave_sdk_setupenv.bat below

2. Obtain CRC.pm (http://cpansearch.perl.org/src/OLIMAUL/Digest-CRC-0.21/lib/Digest/CRC.pm) and copy it to your Perl installation's lib\Digest path. ex: C:\Strawberry\perl\lib\Digest
3. Run **mmwave_sdk_setupenv.bat.** This should **not give errors** and should give the following output. The build environment is now setup

```
------------------------------------------------
mmWave Build Environment Configured
------------------------------------------------
```

Please note that the versions shown above are examples. The actual versions of the mmwave sdk and tools to be used are given in the Release notes of a particular release. Paths have to be updated with the correct location of the tools installed on the user's machine

4. 4. 2. 2. Linux

1. Open a terminal and cd to <mmwave_sdk_<ver> install path>/packages/scripts/unix. Set the paths shown below as per your tools installation location. Place these commands in a shell script setenv_tools.sh, enable execute permission and source it.

setenv_tools.sh

```
# Select your device. Options (case sensitive) are: awr14xx, iwr14xx,
awr16xx, iwr16xx
export MMWAVE_SDK_DEVICE=awr16xx

# By default all tools are installed under ~/ti folder. Change this if
installing to a different location
export DEFAULT_TOOLS_PATH=~/ti

# Common settings for awr14xx, awr16xx, iwr14xx, iwr16xx
# TI ARM compiler
export
R4F_CODEGEN_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/ccsv7/tools/compiler/t
i-cgt-arm_16.9.1.LTS
# Path to <mmwave_sdk installation path>/packages folder
export
MMWAVE_SDK_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/mmwave_sdk_<ver>/packag
es
# TI XDC
export XDC_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/xdctools_3_50_00_10_core
# TI BIOS
export
BIOS_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/bios_6_50_01_12/packages
# perl
export PERL_INSTALL_PATH=/usr/bin
# if using CCS to download, set below define to yes else no
export DOWNLOAD_FROM_CCS=yes

# Following only needed for awr16xx and iwr16xx
# TI DSP compiler
export
C674_CODEGEN_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/ccsv7/tools/compiler/
ti-cgt-c6000_8.1.3
# DSPlib
export
C64Px_DSPLIB_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/dsplib_c64Px_3_4_0_0
# MATHlib
export
C674x_MATHLIB_INSTALL_PATH=${DEFAULT_TOOLS_PATH}/mathlib_c674x_3_1_2_1
# awr16xx/iwr16xx radarss firmware. Use the RPRC formatted binary
file.
export
XWR16XX_RADARSS_IMAGE_BIN=${MMWAVE_SDK_INSTALL_PATH}/../firmware/rada
rss/xwr16xx_radarss_rprc.bin
```

Run setenv_tools.sh

```
chmod +x setenv_tools.sh
source ./setenv_tools.sh
```

Path setting for TI tools is done by mmwave_sdk_setupenv.sh below

2. Install CRC for the perl installation:

Install other needed tools

```
sudo apt-get --assume-yes install libdigest-crc-perl

# Install Mono. This is needed to run windows executable to convert
.out to .bin (flashable application executable)
# The following command was tested for Ubuntu. For other Linux
distributions refer to
http://www.mono-project.com/download/#download-lin
sudo apt-get --assume-yes install mono-complete
```

3. 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
```

4. Run mmwave_sdk_setupenv.sh as shown below. This should not give errors and should print the message "Build Environment configured". The build environment is now setup.

Run mmwave_sdk_setupenv.sh

```
source ./mmwave_sdk_setupenv.sh
```

Please note that the versions shown above are examples. The actual versions of the mmwave sdk and tools to be used are given in the Release notes of a particular release. Paths have to be updated with the correct location of the tools installed on the user's machine

4. 4. 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 command. On successful execution of the command, 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 xWR14xx/xWR16xx EVM".

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

Building demo in windows

```
REM Use xwr14xx or xwr16xx for <device type> below. Use mmw or capture for
<demo> below
cd %MMWAVE_SDK_INSTALL_PATH%/ti/demo/<device type>/<demo>

REM Clean and build
gmake clean
gmake all

REM Incremental build
gmake all

REM For example to build the mmw demo for awr14xx or iwr14xx
cd %MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr14xx/mmw
gmake clean
gmake all
REM This will create xwr14xx_mmw_demo_mss.xer4f & xwr14xx_mmw_demo_mss.bin
binaries under %MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr14xx/mmw folder

REM For example to build the mmw demo for awr16xx or iwr16xx
cd %MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr16xx/mmw
gmake clean
gmake all
REM This will create xwr16xx_mmw_demo_mss.xer4f, xwr16xx_mmw_demo_dss.xe674
& xwr14xx_mmw_demo.bin binaries under
%MMWAVE_SDK_INSTALL_PATH%/ti/demo/xwr16xx/mmw folder
```

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

Building demo in linux

```
# Use xwr14xx or xwr16xx for <device type> below. Use mmw or capture for
<demo> below
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/<device type>/<demo>

# Clean and build
make clean
make all

# Incremental build
make all

# For example to build the mmw demo for awr14xx or iwr14xx
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr14xx/mmw
make clean
make all
# This will create xwr14xx_mmw_demo_mss.xer4f & xwr14xx_mmw_demo_mss.bin
binaries under ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr14xx/mmw folder

# For example to build the mmw demo for awr16xx or iwr16xx
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr16xx/mmw
make clean
make all
# This will create xwr16xx_mmw_demo_mss.xer4f, xwr16xx_mmw_demo_dss.xe674 &
xwr14xx_mmw_demo.bin binaries under
${MMWAVE_SDK_INSTALL_PATH}/ti/demo/xwr16xx/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. 4. 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. 4. 4. 1. Building drivers/control/alg components

To clean build driver, control 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/<path to the component>
gmake clean
gmake all

REM For example to build the adcbuf lib and unit test
cd %MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf
gmake clean
gmake all
REM If MMWAVE_SDK_DEVICE is set to awr14xx or iwr14xx, the commands will
create
REM     libadcbuf_xwr14xx.aer4f library under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/lib folder
REM     xwr14xx_adcbuf_mss.xer4f unit test binary under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/test/xwr14xx folder
REM If MMWAVE_SDK_DEVICE is set to awr16xx or iwr16xx, the commands will
create
REM     libadcbuf_xwr16xx.aer4f library for MSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/lib folder
REM     xwr16xx_adcbuf_mss.xer4f unit test binary for MSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/test/xwr16xx folder
REM     libadcbuf_xwr16xx.ae674 library for DSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/lib folder
REM     xwr16xx_adcbuf_dss.xe674 unit test binary for DSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/adcbuf/test/xwr16xx folder

REM For example to build the mmwavelink lib and unit test
cd %MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink
gmake clean
gmake all
REM If MMWAVE_SDK_DEVICE is set to awr14xx or iwr14xx, the commands will
create
REM     libmmwavelink_xwr14xx.aer4f library under
%MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink/lib folder
REM     xwr14xx_link_mss.xer4f unit test binary under
%MMWAVE_SDK_INSTALL_PATH%/ti/drivers/control/mmwavelink/test/xwr14xx folder
REM If MMWAVE_SDK_DEVICE is set to awr16xx or iwr16xx, the commands will
create
REM     libmmwavelink_xwr16xx.aer4f library for MSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink/lib folder
REM     xwr16xx_link_mss.xer4f unit test binary for MSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink/test/xwr16xx folder
REM     libmmwavelink_xwr16xx.ae674 library for DSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink/lib folder
REM     xwr16xx_link_dss.xe674 unit test binary for DSS under
%MMWAVE_SDK_INSTALL_PATH%/ti/control/mmwavelink/test/xwr16xx folder

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/<path to the component>
make clean
make all

# For example to build the adcbuf lib and unit test
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf
make clean
make all
# If MMWAVE_SDK_DEVICE is set to awr14xx or iwr14xx, the commands will
create
#     libadcbuf_xwr14xx.aer4f library under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/lib folder
#     xwr14xx_adcbuf_mss.xer4f unit test binary under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/test/xwr14xx folder
# If MMWAVE_SDK_DEVICE is set to awr16xx or iwr16xx, the commands will
create
#     libadcbuf_xwr16xx.aer4f library for MSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/lib folder
#     xwr16xx_adcbuf_mss.xer4f unit test binary for MSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/test/xwr16xx folder
#     libadcbuf_xwr16xx.ae674 library for DSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/lib folder
#     xwr16xx_adcbuf_dss.xe674 unit test binary for DSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/adcbuf/test/xwr16xx folder


# For example to build the mmwavelink lib and unit test
cd ${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink
make clean
make all
# If MMWAVE_SDK_DEVICE is set to awr14xx or iwr14xx, the commands will
create
#     libmmwavelink_xwr14xx.aer4f library under
${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink/lib folder
#     xwr14xx_link_mss.xer4f unit test binary under
${MMWAVE_SDK_INSTALL_PATH}/ti/drivers/control/mmwavelink/test/xwr14xx
folder
# If MMWAVE_SDK_DEVICE is set to awr16xx or iwr16xx, the commands will
create
#     libmmwavelink_xwr16xx.aer4f library for MSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink/lib folder
#     xwr16xx_link_mss.xer4f unit test binary for MSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink/test/xwr16xx folder
#     libmmwavelink_xwr16xx.ae674 library for DSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink/lib folder
#     xwr16xx_link_dss.xe674 unit test binary for DSS under
${MMWAVE_SDK_INSTALL_PATH}/ti/control/mmwavelink/test/xwr16xx folder

# Additional build options for each component can be found by invoking make
help
make 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 drivers are supported for all devices. List of supported drivers for each device is listed in the Release Notes.

5. MMWAVE SDK deep dive

5. 1. Typical mmWave Radar Processing Chain

Following figure shows a typical mmWave Radar processing chain:

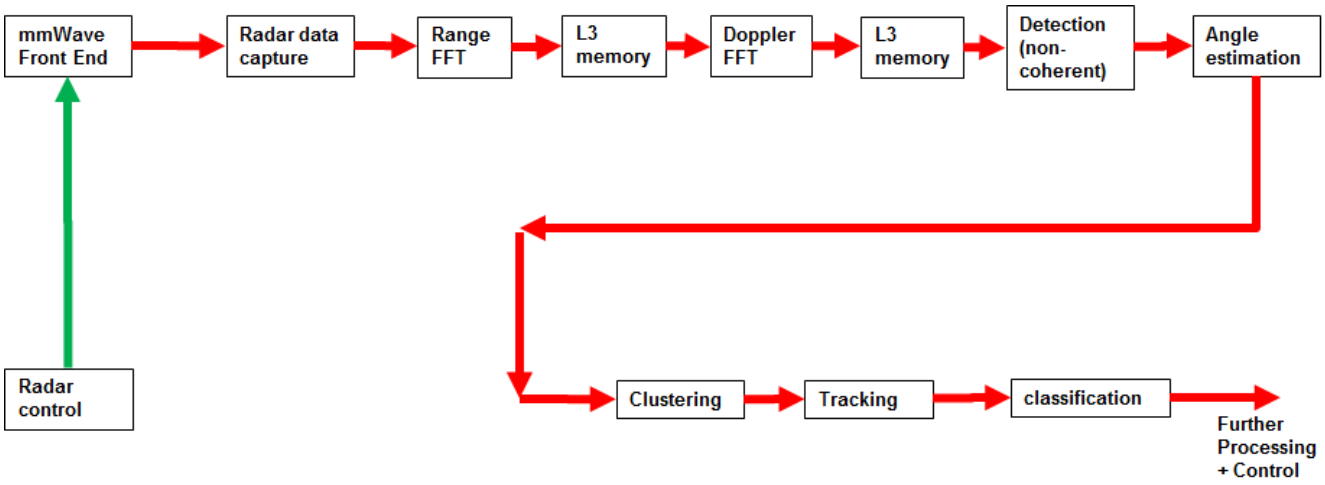


Figure 6: Typical mmWave radar processing chain

Using mmWave SDK the above chain could be realized as shown in the following figure for xWR14xx and xWR16xx. 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 AR device or would run entirely on the external processor.

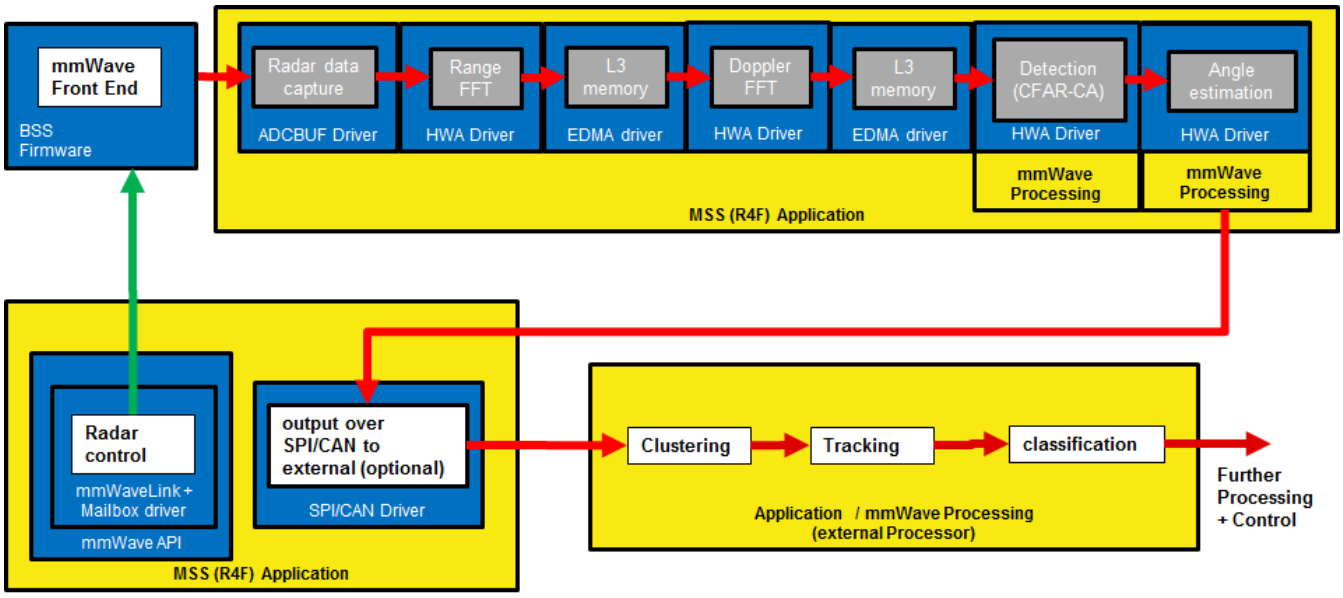


Figure 7: Typical mmWave radar processing chain using xWR14xx mmWave SDK

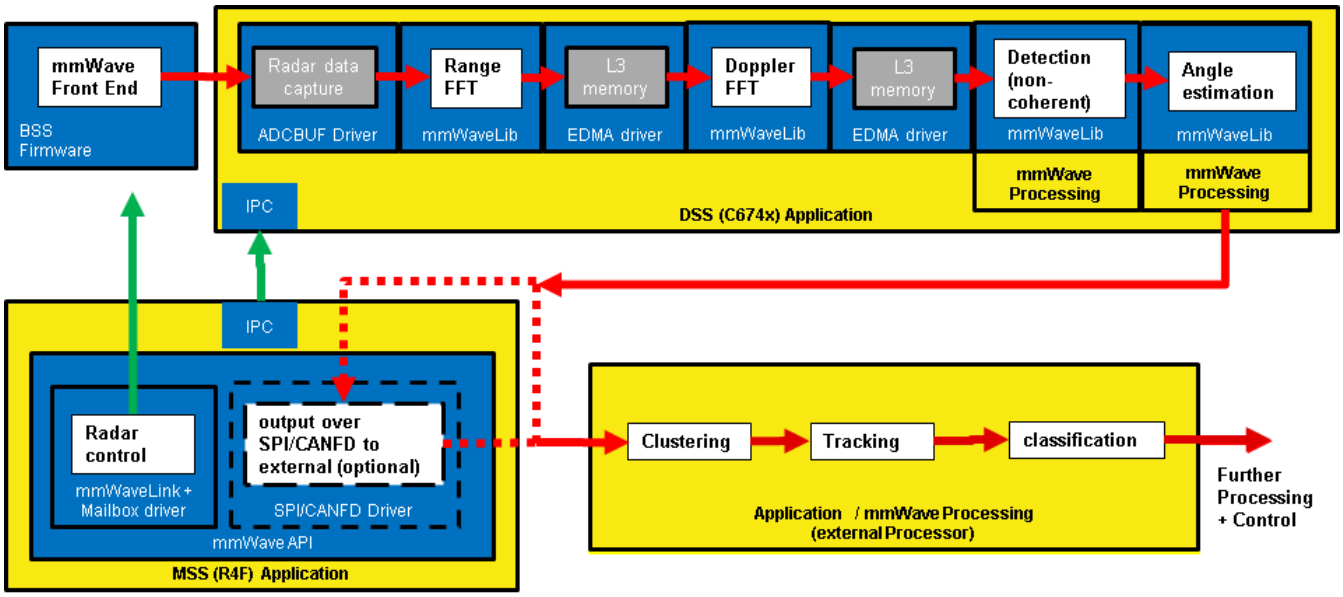


Figure 8: Typical mmWave radar processing chain using xWR16xx mmWave SDK

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. 2. Typical Programming Sequence

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

5. 2. 1. Control Path

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

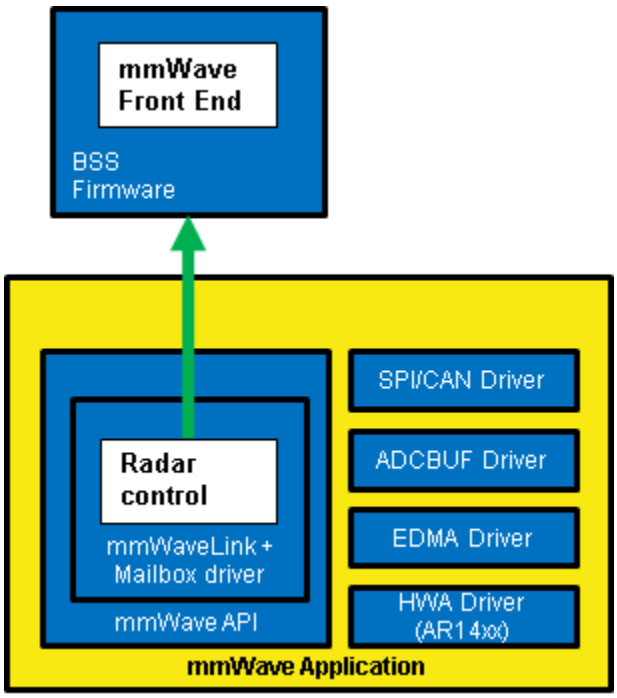


Figure 9: Typical mmWave radar control flow

Following set of figures shows how an application programming sequence would be for setting up the typical control path - init, config, start. This is a high level diagram simplified to highlight the main sofwtare 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. 2. 1. 1. xWR14xx (MSS<->RADARSS)

On xWR14xx, the control path runs on the Master subsystem (Cortex-R4F) and the application can simply call the mmwave APIs in the SDK to realize most of the functionality.

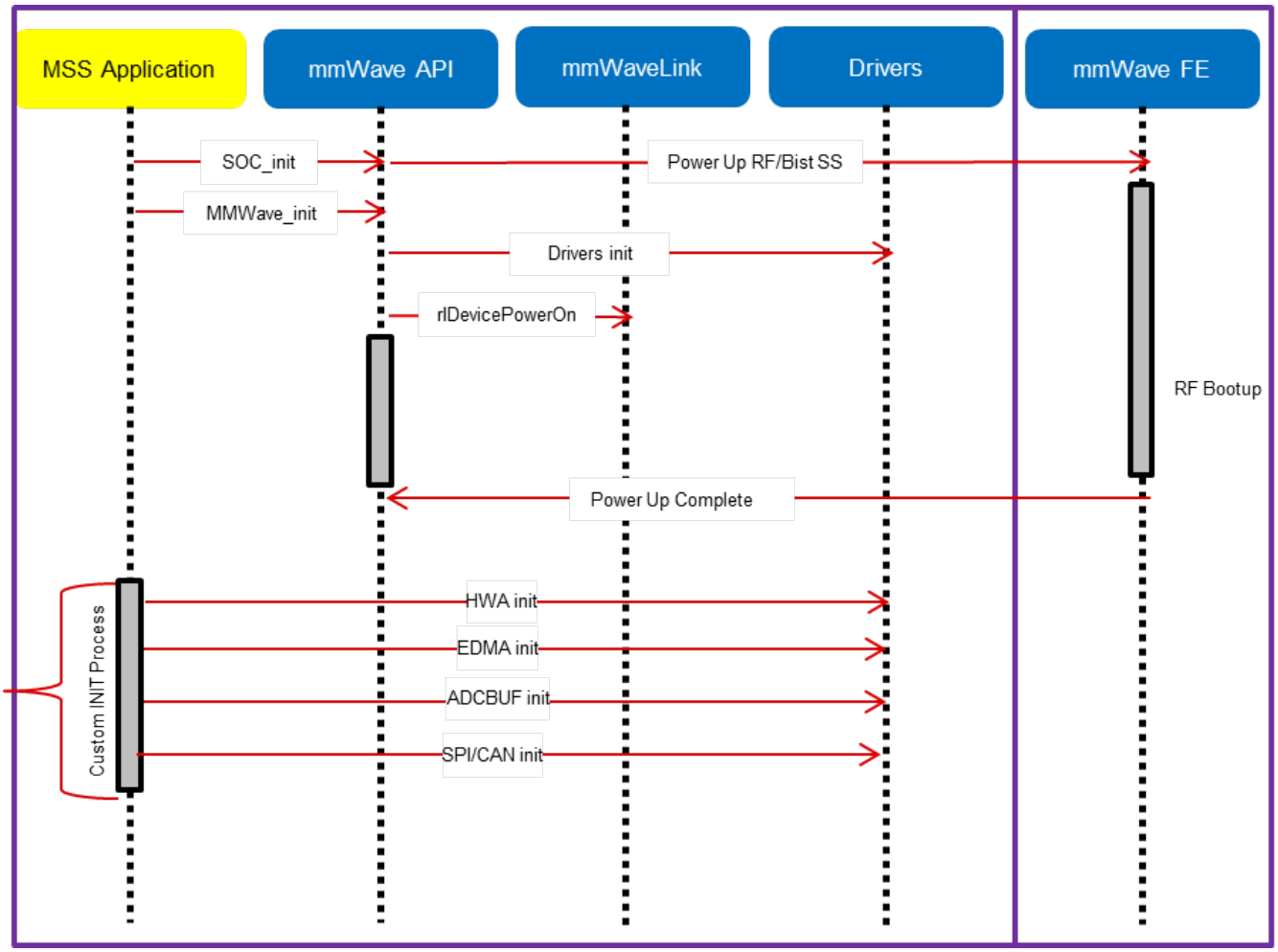


Figure 10: xWR14xx: Detailed Control Flow (Init sequence)

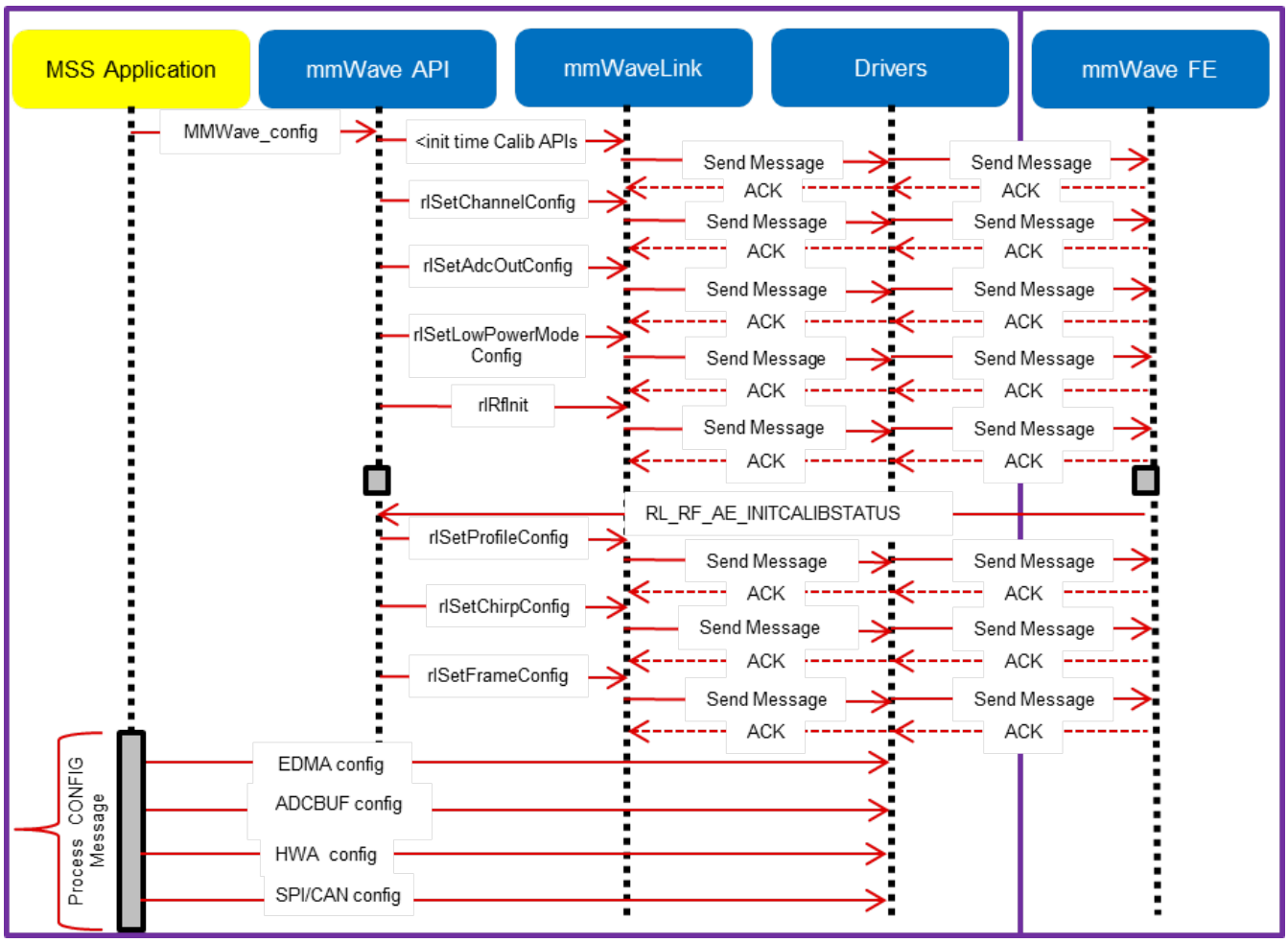


Figure 11: xWR14xx: Detailed Control Flow (Config sequence)

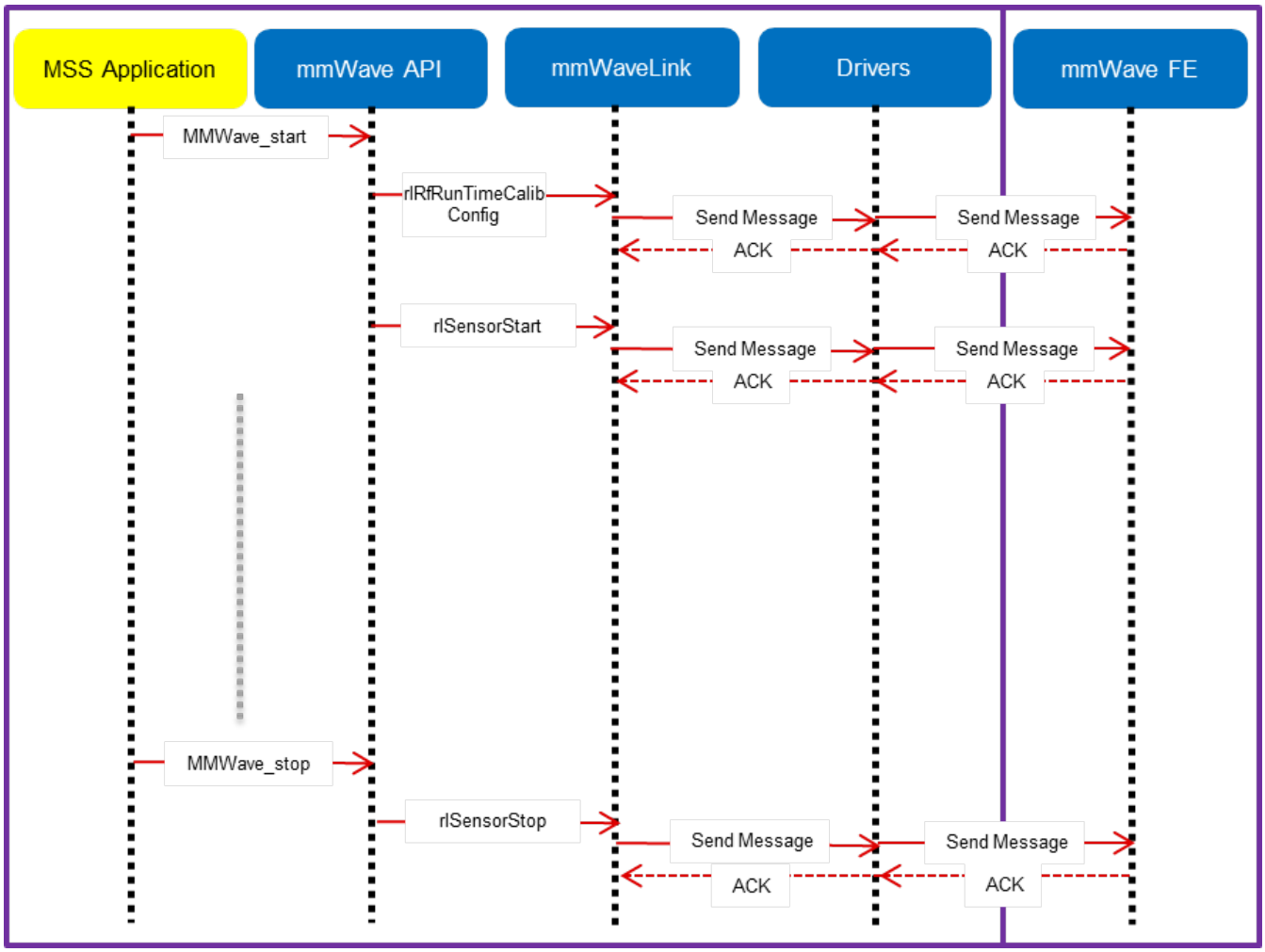


Figure 12: xWR14xx: Detailed Control Flow (start sequence)

5. 2. 1. 2. xWR16xx

On xWR16xx, the control path can run on MSS only, DSS only or in "co-operative" mode where the init and config are 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 co-operative mode can be seen in the MMW demo. The "capture" demo provides a sample implementation of all 3 modes.

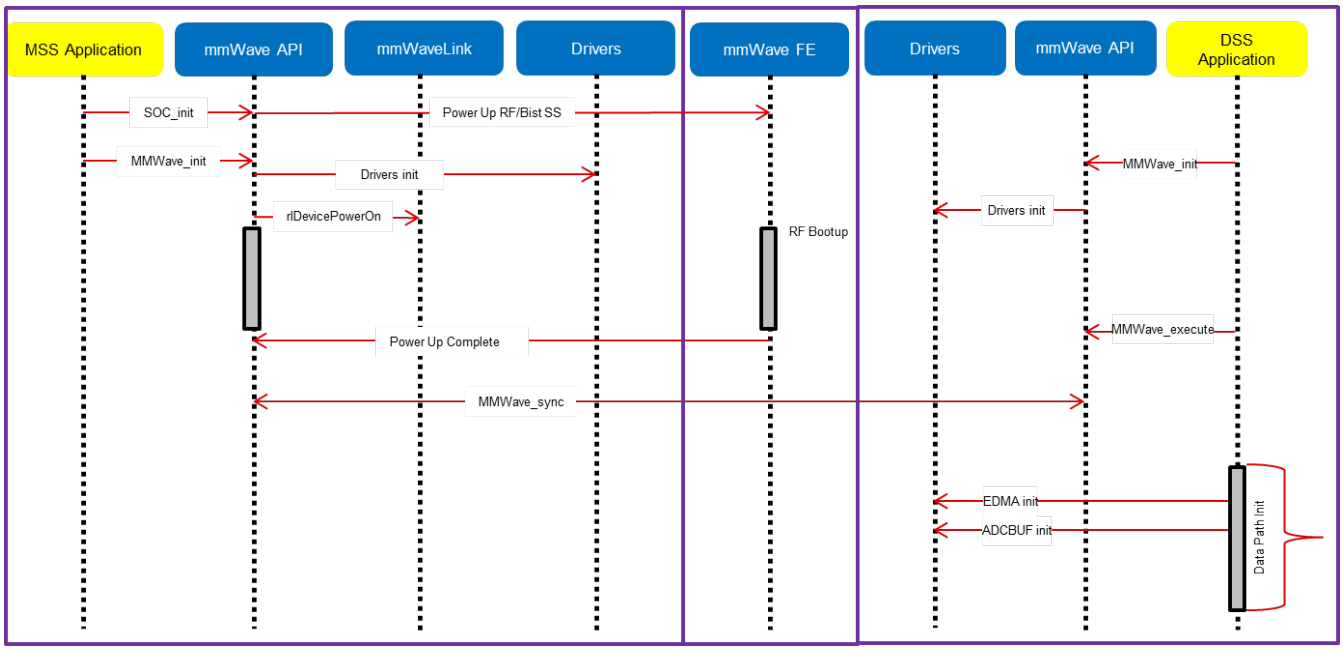


Figure 13: ***xWR16xx: Detailed Control Flow (Init sequence)***

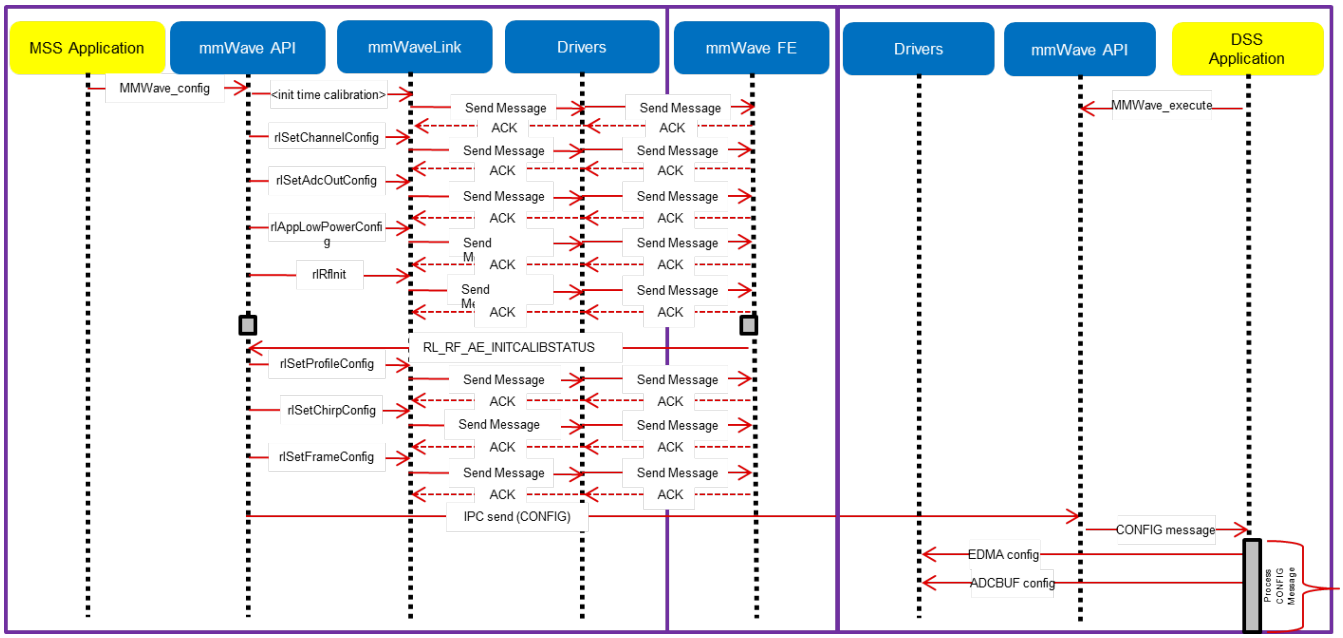


Figure 14: ***xWR16xx: Detailed Control Flow (Config sequence)***

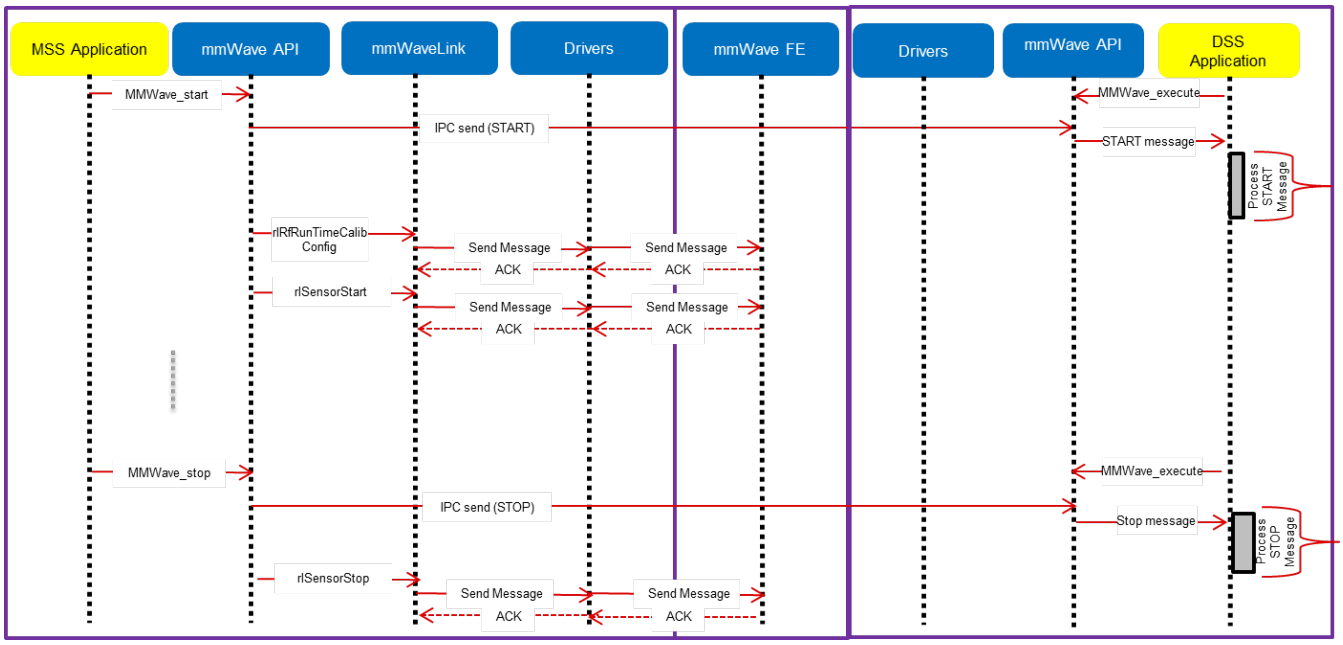


*Figure 15: **xWR16xx: Detailed Control Flow (Start sequence)***

5. 2. 2. Data Path

5. 2. 2. 1. xWR14xx

In xWR14xx, the data path in the above processing chain is depicted by the following blocks running on the MSS (Cortex-R4F).

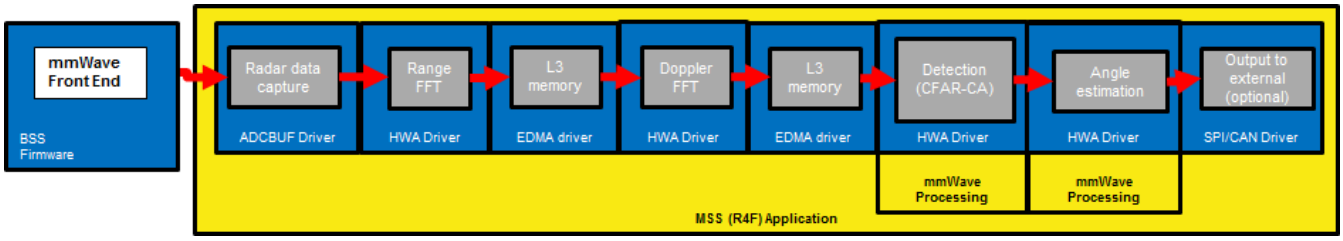


Figure 16: Typical mmWave radar data flow in xWR14xx

Please refer to the documentation provided here mmwave_sdk_<ver>\packages\ti\demo\xwr14xx\mmw\docs\doxygen\html\index.html for more details on each of the individual blocks of the data path.

5. 2. 2. 2. xWR16xx

In xWR16xx, the data path in the above processing chain is depicted by the following blocks running on primarlity on the DSS (C674x).

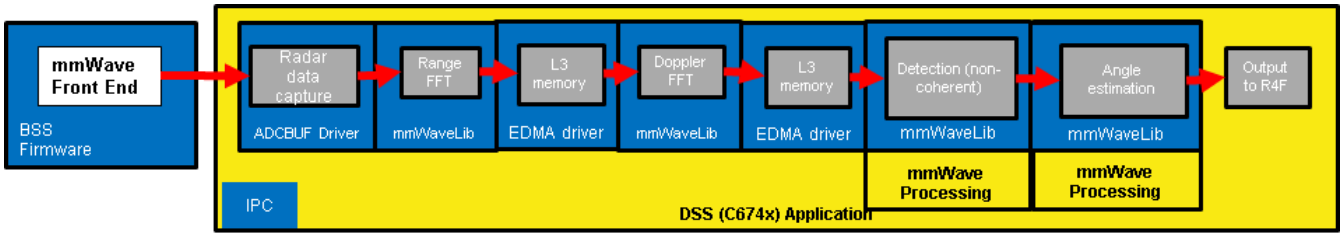


Figure 17: Typical mmWave radar data flow in xWR16xx

Please refer to the documentation provided here mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\mmw\docs\doxygen\html\index.html for more details on each of the individual blocks of the data path.

5. 3. mmWave SDK - TI components

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

5. 3. 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.

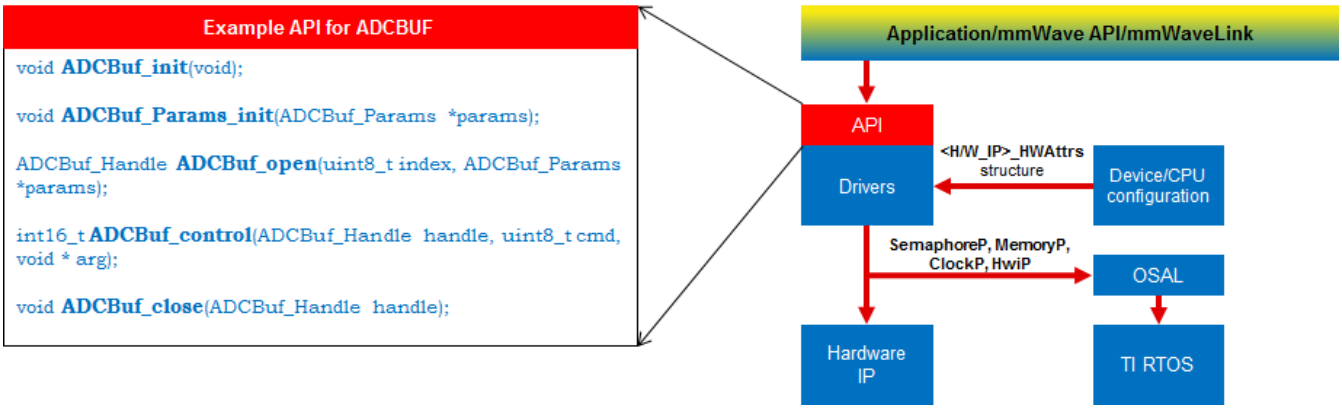


Figure 18: mmWave SDK Drivers - Internal software design

5. 3. 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. 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 on xWR16xx 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. 3. 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, xWR16xx only). 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. 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\ .

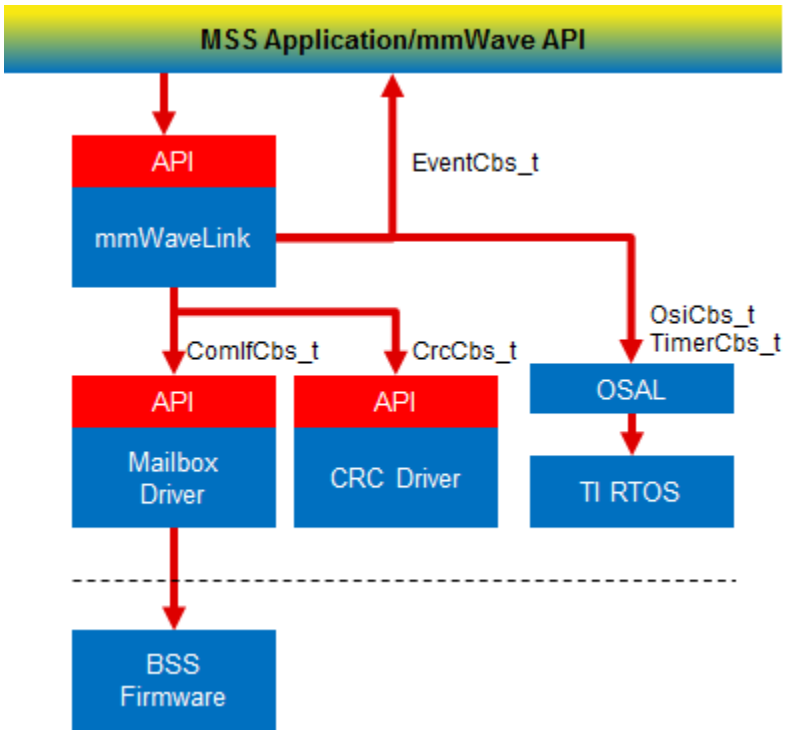


Figure 19: mmWaveLink - Internal software design

5. 3. 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 xWR16xx, 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\.

mmWave Front End Calibrations
mmWave API, by default, enables all init/boot time time calibrations for mmWave Front End. Moreover, 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.

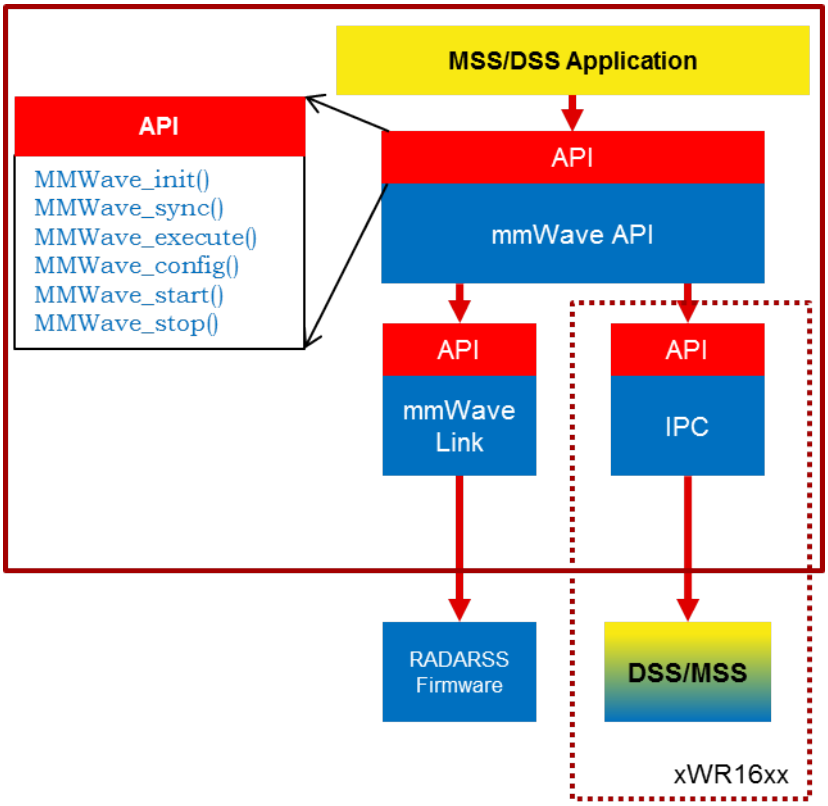


Figure 20: mmWave API - Internal software design

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

5. 3. 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 on the xr16xx
- Asynchronous Event Management
- Start & Stop services
- Configuration of the RADARSS for Chirp & 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. 3. 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 xr16xx
- 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 xWR16xx, the application is responsible for passing the configuration between the MSS and DSS if required.

See sample call flow below:

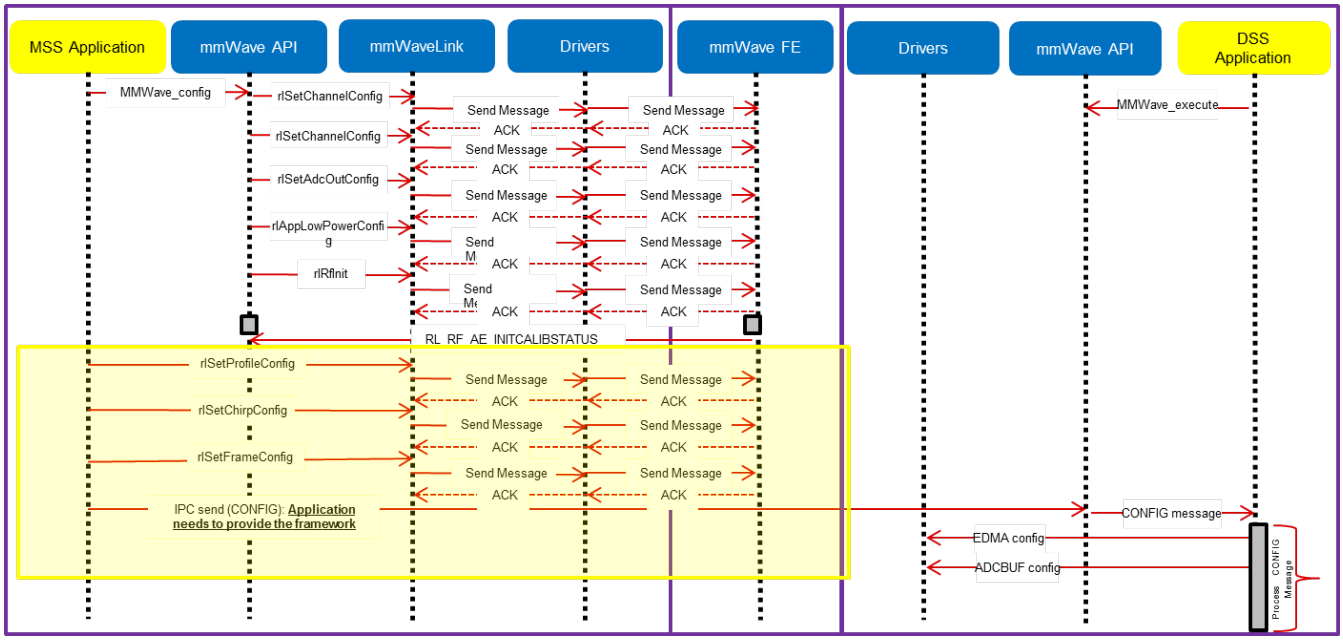


Figure 21: mmWave API - 'Minimal' Config - Sample flow (xWR16xx)

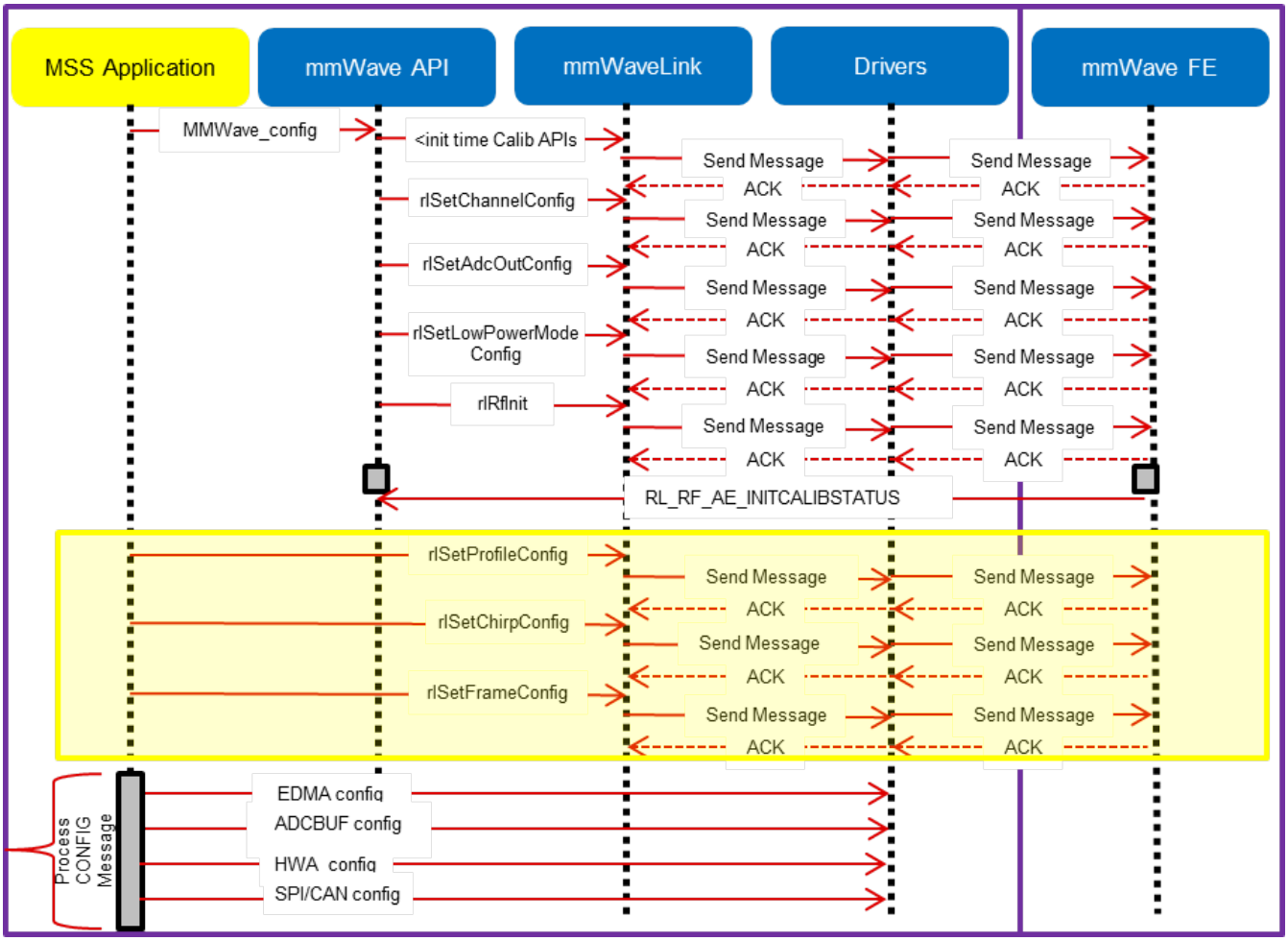


Figure 22: mmWave API - 'Minimal' Config - Sample flow (xWR14xx)

5. 3. 5. mmWaveLib

mmWaveLib is a collection of algorithms that provide basic functionality needed for FMCW radar-cube processing. This component is available for xWR16xx only and 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\ .

5. 3. 6. RADARSS Firmware

This is a binary (mmwave_sdk_<ver>\firmware\radarss) that runs on Radar subsystem of the xWR14xx/xWR16xx 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.

5. 3. 7. 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 CCSdevelopmentmode for more details. For xWR14xx, this binary is for R4F (MSS) and for xWR16xx, 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.

5. 3. 8. 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. 3. 8. 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. 3. 8. 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 xWR16xx only. 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 xWR16xx MSS provides this functionality under its hood.

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 xWR14xx and xWR16xx 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.

MPU has been implemented in the SOC module as a private function SOC_mpu_config() that is called by public API SOC_init(). 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.

> A build time option called DOWNLOAD_FROM_CCS has been added which when set to yes prevents program space from being protected. 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 (xWR16xx C674)

The cacheability property of the various regions as seen by the DSP (C674x in xWR16xx) 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. 3. 8. 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. 3. 9. 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 millimeter wave demo. The demo runs in real-time and has all the associated framework for RADARSS control etc with it.

The “HWA_EDMA” for xwr14xx and "DSP_EDMA" for xwr16xx tests (located at mmwave_sdk_<ver>\packages\ti\drivers\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 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 mmw demo documentation.

The “HWA_EDMA” and "DSP_EDMA" 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 “HWA_EDMA” and "DSP_EDMA" 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\drivers\test\<test_dir>\docs\doxygen\html\index.html.

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.

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

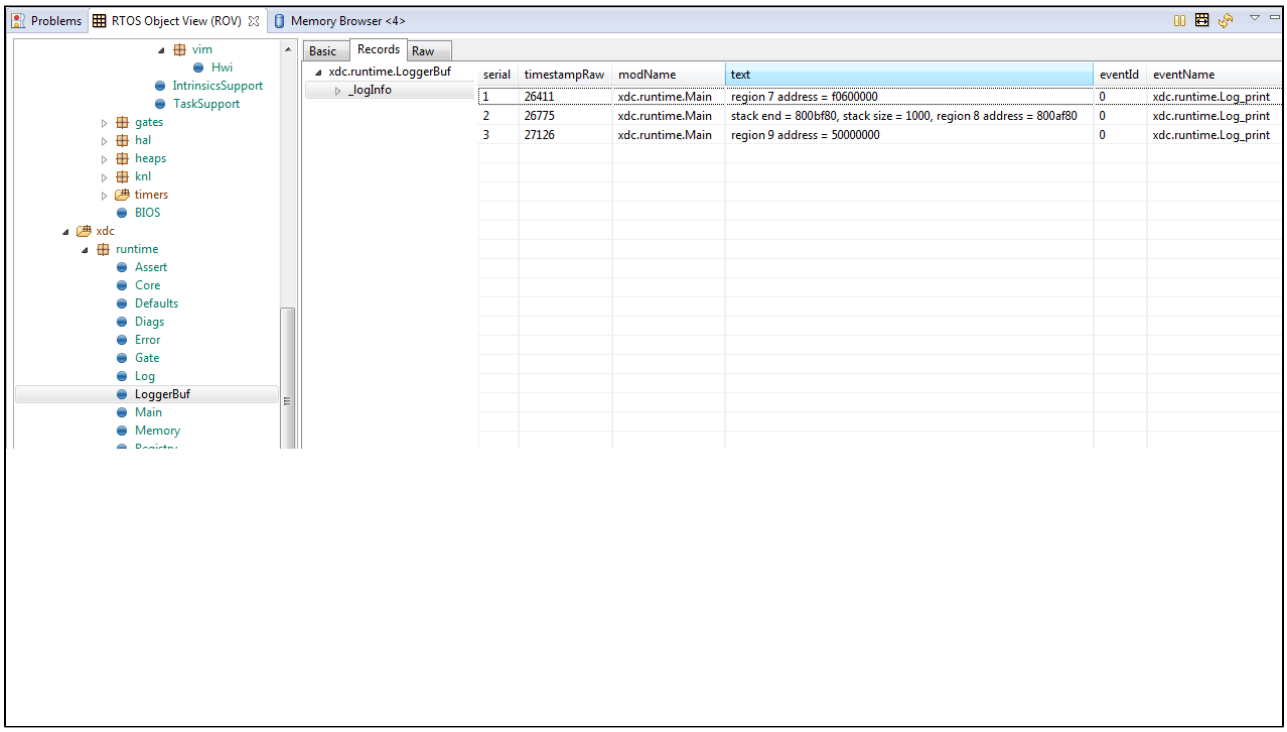
```
var Log       = xdc.useModule('xdc.runtime.Log');
var Main = xdc.useModule('xdc.runtime.Main');
var Diags = xdc.useModule('xdc.runtime.Diags');
var LoggerBuf                    =
xdc.useModule('xdc.runtime.LoggerBuf');
LoggerBuf.TimestampProxy         =
xdc.useModule('xdc.runtime.Timestamp');

/* 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 :



6. 4. Shared memory usage by SDK demos (xWR1642)

Existing SDK demos (capture, mmw) for xWR1642 assigns 5 banks of L3 memory to DSS (i.e. 640KB) and 1 bank of L3 memory to RADARSS(128KB). 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 xWR1642 device determines the shared memory allocation. So in CCS development mode, its the allocation defined in ccsdebug image that applies and not the application that you download via CCS.

In SDK code, you can relate to these settings in the following places:

- Linker command files in mmwave_sdk_<ver>\packages\ti\platform\xWR16xx show the 640KB allocation (0xA0000) for the L3 memory section
- Makefiles for the following components use the value 0x01000005 for SHMEM_ALLOC parameter when invoking the generateMetaImage script. (See xWR1xxx Image Creator section)
  - mmwave_sdk_<ver>\packages\ti\utils\ccsdebug
  - mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\mmw
  - mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\capture

6. 5. xWR1xxx 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 xWR1xxx 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, follow the steps below to create the flash images based on the mmWave device that it is intended for.

6. 5. 1. xWR14xx

Use the generateBin script present under mmwave_sdk_<ver>\packages\ scripts\windows or mmwave_sdk_<ver>\packages\ scripts\linux for the conversion of out file to bin for the MSS R4F. Set **MMWAVE_SDK_DEVICE=awr14xx** or **MMWAVE_SDK_DEVICE=iwr14xx** and **MMWAVE_SDK_INSTALL_PATH=mmwave_sdk_<ver>\packages** in your environment before calling this script.This script needs 3 parameters:

- .out file : this is the input file and the parameter represents the file generated after the link step for the application (application executable). (this can be with file path)
- .bin file : this is the output file generated by the script and this parameter represents the filename for the flash binary (this can be with file path)
- offset : this is the offset for the MSS image section needed by the Bootloader and should be 0x200000

The .bin file generated by this script should be used for the MSS_BUILD during flashing step (How to flash an image onto xWR14xx/xWR16xx EVM)

Unaligned sections
MSS Bootloader for xWR14xx requires that the loadable sections be aligned to 16 bytes using ALIGN(16) in the linker command file. If this is not done, then out2rprc.exe (called from generateBin script) will throw following error and the bin file generated by generateBin will be incomplete!

Unaligned sections: File conversion failure

```
Parsing the input object file, xwr14xx_mmw_demo_mss.xer4f.
Appending zeros 0
Appending zeros 256
Appending zeros 125192
Unaligned section 125192
File conversion failure!
```

6. 5. 2. xWR16xx

Application Image generation is two-step process for xWR16xx. Refer to "xWR1xxx Image Creator User Guide" in the SDK docs directory for details on the internal layout and format of the files generated in these steps.

1. RPRC format conversion:
   Firstly, application executable has to be converted from ELF/COFF format to custom TI RPRC image format.
   Use the generateBin script present under mmwave_sdk_<ver>\packages\ scripts\windows or mmwave_sdk_<ver>\packages\ scripts\ linux for the conversion of out file to bin for the MSS R4F and for the DSS C674 (need to run the script twice). Set **MMWAVE_SDK_ DEVICE=awr16xx** or **MMWAVE_SDK_DEVICE=iwr16xx** and **MMWAVE_SDK_INSTALL_PATH=mmwave_sdk_<ver>\packages** i n your environment before calling this script. This script needs 2 parameters:
   - executable (.xer4f or .xe674) file : this is the input file and the parameter represents the file generated after the link step for the application (application executable).
   - binary (.bin) file : this is the output file generated by the script and this parameter represents the filename for the flash binary
2. Multicore Image file generation:
   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, DSS and RADARSS binaries into one metaImage and appending correct CRC. Set **MMWAVE_S DK_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 "xWR1xxx Image Creator User Guide" in the SDK.
   - MSS_IMAGE: [input] MSS input image in RPRC (,bin) format as generated by generateBin script in the step above, use keyword NULL if not needed
   - BSS_IMAGE: [input] RADARSS (BSS) input image in RPRC (,bin) format, use keyword NULL if not needed. Use mmwa ve_sdk_<ver>\firmware\radarss\xwr16xx_radarss_rprc.bin here.
   - DSS_IMAGE: [input] DSP input image in RPRC (,bin) format as generated by generateBin script in the step above, 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 xWR14xx/xWR16xx EVM)

6. 6. xWR16xx 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. 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 portions of code that is not compute (MIPS) critical with ARM thumb option (depending on the compiler use). If using the TI ARM compiler, the option to do thumb is `code_state=16`. The pre-built drivers in the SDK are not built with the thumb option because at the driver level, decisions cannot be made as to what APIs will be in compute critical path and what will not be and it will depend on the customer use cases. The demo code is also not built with thumb option to keep build artifacts and code organization simpler. 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.
2. On C674X, compile portions of code that are not in compute critical path with appropriate -msX option. The -ms options are presently not used in the SDK drivers or demos. For more details, 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 at `ti/demo/xwr16xx/mmw`.
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.
5. If there is no requirement to be able to restart an application without reloading, then following suggestions may be used:
    a. one time/first time only program code can be overlaid with data memory buffers used after such code is executed. This is illustrated in the mmw demo on C674X side where such code is overlaid with (load time uninitialized) radar cube data in L3 RAM, refer to the file `mmwave_sdk_<ver>\packages\ti\demo\xwr16xx\mmw\dss\dss_mmw_linker.cmd`. (Note: Ability to place code at function granularity requires to use the aforementioned -mo option).
    b. 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.

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. 8. DSPlib integration in xWR16xx C674x 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. 8. 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 mmw demo code as seen in the following code snippets: (Note: the mmw demo only uses one (C64P) dsplib so it could have been integrated in the straight-forward way but it is deliberately done this way to illustrate the method in question here and allows for future integration with C674x dsplib).

Sample DSPLib integration using individual functions

In file dss_mmw.mak:

dss_mmw.mak

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

-i$(C64Px_DSPLIB_INSTALL_PATH)/packages/ti/dsplib/src/DSP_fft16x
16/c64P \ <-- include path for DSP_fft16x16

-i$(C64Px_DSPLIB_INSTALL_PATH)/packages/ti/dsplib/src/DSP_fft32x
32/c64 \ <-- include path for DSP_fft32x32
                        -i$(C674x_MATHLIB_INSTALL_PATH)/packages
\
```

In dss_data_path.c:

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 (not illustrated in mmw demo) 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. 8. 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 file dss_mmw.mak:

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 dss_data_path.c:

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. 9. 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 xWR14xx, when 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. 10. CCS Debugging of real time application

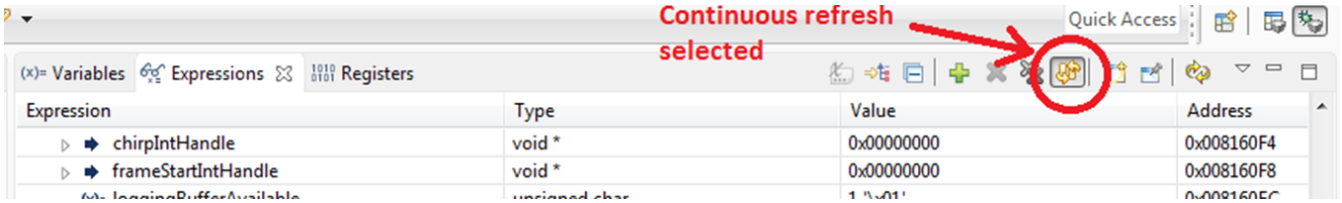
6. 10. 1. Using printfs in real time

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).

6. 10. 2. 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: