MMWAVE DFP User Guide

Product Release: DFP 2.2.2.1

Release Date: Aug 28, 2020

Contents

Abbreviations 4
1. System Overview 5
1.1 mmWave Firmware 5
1.2 mmWaveLink 5
1.3 mmWave RF Evaluation 6
2. AWR2243 - System Deployment 8
3. Getting started 9
4. PC Connection 10
4.1 Connection with AWR2243 ES1.1 BOOST 10
4.2 Connection with DCA1000-EVM 10
5. Demonstration Mode 13
6. Visual Studio Development Mode 15
7. Data capture using DCA1000 EVM 16
8. Introduction to DFP examples 17
8.1 mmWaveLink_SingleChip_Example (SPI / I2C mode of operation) 18
8.2 mmWaveLink_SingleChip_Monitoring 18
8.3 mmWaveLink_SingleChip_NonOS_Example 19
8.4 mmWaveLink_SFlash_FW_Example 19
8.5 mmWaveLink_Cascade_Example 19
8.6 mmWaveLink_Cascade_Monitoring 20
9. API Sequence 21
9.1 AWR2243 ES1.1 Single chip API sequence 22
9.2 AWR2243 ES1.1 Two/Four chip API sequence 23
10. Additional dependency 25
11. mmWave Sensor Advanced Features 26
11.1 Advanced frame configuration 26
11.2 Advanced chirp configuration 29
11.3 Dynamic chirp configuration 38
11.4 Dynamic profile configuration 40
11.5 Dynamic per chirp phase shifter configuration 40

11.6 Loopback configuration 41
11.7 Factory Calibration 43

Abbreviations

Abbreviation	Description
AE	Asynchronous Event
DFP	Device Firmware Package
RadarSS	Radar Sub system
MasterSS	Master Sub system
SPI	Serial Peripheral Interface

1. System Overview

The mmWave device firmware package (DFP) is split in three broad components: mmWave Firmware, mmWaveLink and mmWave RF evaluation.

1.1 mmWave Firmware

mmWave Firmware is responsible for configuring RF/analog, digital front end in TI mmWave radar devices and consists of below components:

- Master SS firmware (Only for AWR2243 ES1.1)
- Radar SS firmware (Only for AWR2243 ES1.1)

All the services of Master SS and Radar SS firmware are available to external processor using the APIs in mmWaveLink framework.

mmWave Firmware binaries are available at ***'mmwave_dfp_<ver>\firmware\'***

1.2 mmWaveLink

1.2.1 mmWaveLink framework

mmWaveLink framework acts as driver for Master sub-system and Radar sub-system. It exposes a suite of low level APIs which allow application to enable, configure and control Master SS and Radar SS. It provides a well-defined platform and OS abstraction for the application to plug in the communication driver and OS routine callbacks to communicate with the TI mmWave devices. External Processor can use this framework to connect and configure AWR2243 device over SPI.

mmWaveLink source and library are available at '***mmwave_dfp_<ver>\ti\control\mmwavelink***'

1.2.2 mmWaveLink examples

Visual studio based sample applications are included in DFP which emulates a host processor on windows PC and runs the mmWaveLink framework.

- In the case of single chip, SPI is emulated over USB using the FT4232H device.
- In the case of cascade chip, SPI is emulated over Ethernet using the TDA2XX device.

The platform library which implements the required SPI and OS callbacks is included in the mmWave DFP. Refer to section 7.2 for more details.

The examples are available under '***mwave_dfp_<ver>\ti\example\'*** path.

Single Chip Examples

- mmWaveLink_SingleChip_Example
- mmWaveLink_SingleChip_NonOS_Example
- mmWaveLink_SingleChip_Monitoring
- mmWaveLink_SFlash_FW_Example

Cascade Chip Examples

- mmWaveLink_Cascade_Example
- mmWaveLink_Cascade_Monitoring

Figure-1 depicts high level modules of external processor and AWR2243 device.

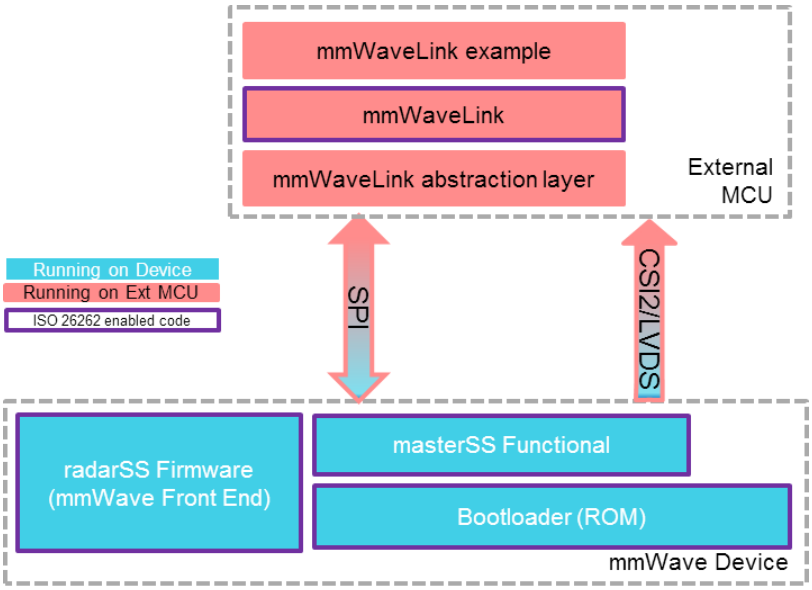


Figure 1. External Processor and AWR2243

1.3 mmWave RF Evaluation

For mmWave RF/System evaluation purpose, mmWave Studio/Radar studio Tool which is designed to communicate with all variants of TI mmWave devices for RF and system performance evaluation can be used; this tool is available at http://www.ti.com/tool/MMWAVE-STUDIO.

The RF evaluation firmware is available at ***'mmwave_dfp<ver>\rf_eval\ rf_eval_firmware'*** directory which should be used in SOP-2 (development) mode only. Refer to mmWave Studio user guide for more details.

Figure-2 depicts the modules of mmWave Studio and connection with AWR2243 and TSW1400 EVM Board (Single Chip)

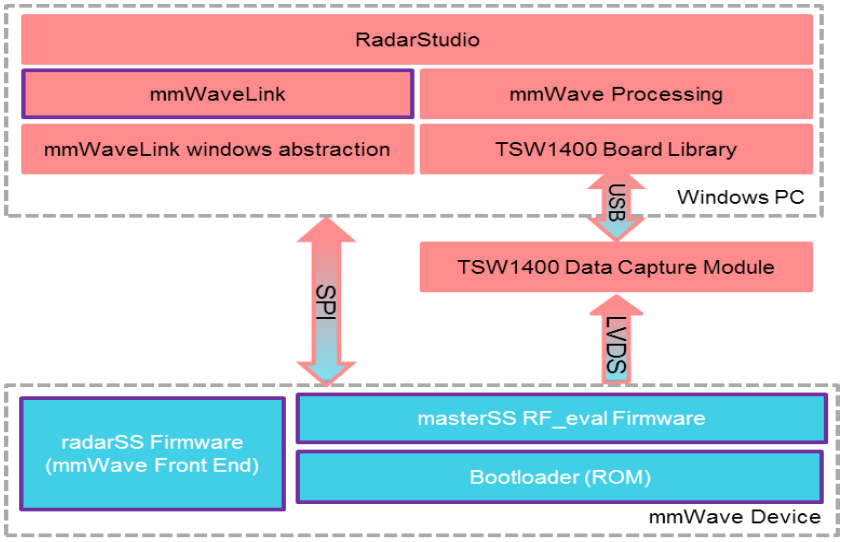


Figure 2. mmWave Studio and AWR2243 (Single Chip)

Figure-3 depicts the setup for Cascade chip.
mmWave Studio and connection with AWR2243 and TDA2XX Board (Cascade Chip)

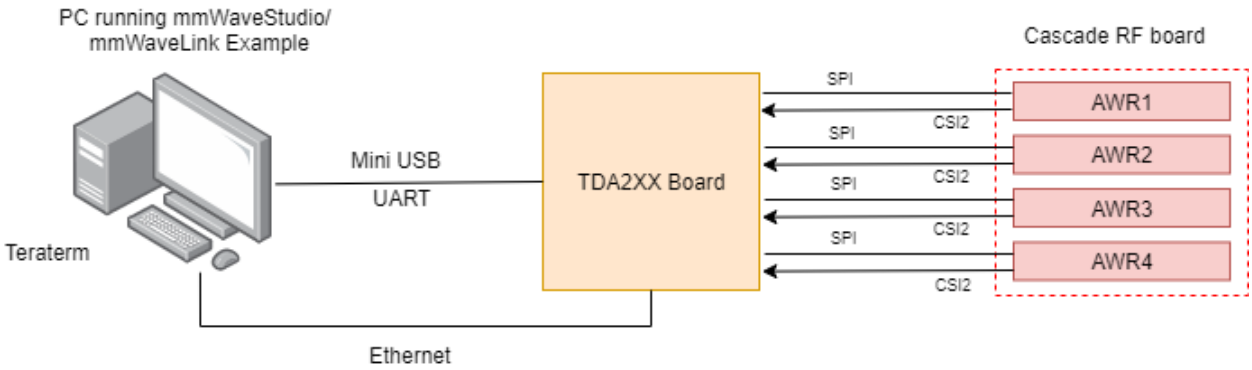


Figure 3. mmWave Studio and AWR2243 (Cascade Chip)

2. AWR2243 - System Deployment

User needs to connect an External Processor with AWR2243 device to configure mmWave front end (Radar SS) and process raw data captured over high speed interface. Figure-4 depicts the connection between external processor and AWR2243 device.

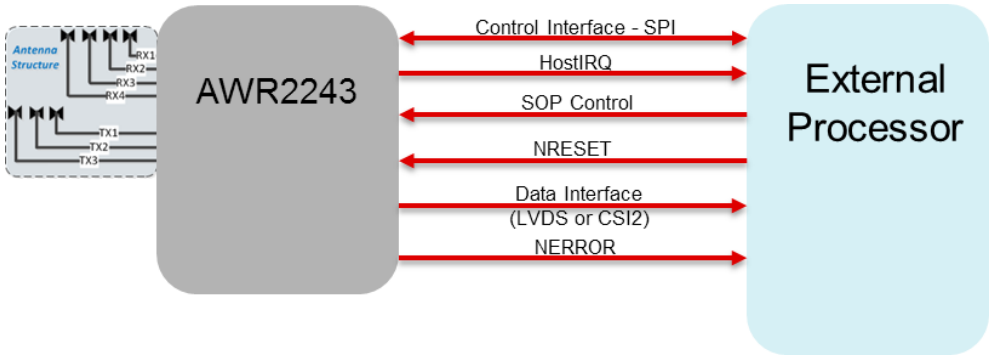


Figure 4. Connection between External Processor and AWR2243

Details of AWR2243 pins which needs to be connected to the external processor:

a. Control Interface- SPI

- SPI_CLK_1: Serial Clock by SPI master (external processor).
- SPI_CS_1: SPI chip select (active low), output from SPI master.
- MOSI_1: Master Output Slave Input (data output from master).
- MISO_1: Master Input Slave Output (data output from slave).

Note: External Processor should configure SPI with below configurations

- SPI mode 0 (Phase 0, Polarity 0)
- SPI word size = 16bit
- CS is toggled for each SPI word
- Minimum 2 SPI clock delay between CS active to rising edge of the first SPI clock
- SPI Clock is recommended to be less than 10MHz for SPI communication over FTDI chip of DCA1000 EVM from PC based application. AWR2243 device supports max 40MHz SPI clock.

b. Host IRQ: SPI_HOST1_INTR (SPI Host interrupt) line, output from AWR2243 to notify Host. It connects to one of the interrupt capable GPIO pin in external processor

c. **SOP Control:** For SOP mode 4 setting – SPI - (SOP0 pin = 1, SOP1 pin = 0, SOP2 pin = 0). For SOP mode 7 setting – I2C - (SOP0 pin = 1, SOP1 pin = 1, SOP2 pin = 1).

d. NRESET: Output from external processor to reset AWR2243 device. To power up AWR2243, NRESET = 1. To power Off, NRESET = 0

e. Data Interface: These lines are used for High speed interface to send raw data over CSI2/LVDS from AWR2243 to external processor.

CSI2_TXP[0]	Differential data Out – Lane 0
CSI2_TXM[0]	
CSI2_CLKP	Differential clock Out
CSI2_CLKM	
CSI2_TXP[1]	Differential data Out – Lane 1
CSI2_TXM[1]	
CSI2_TXP[2]	Differential data Out – Lane 2
CSI2_TXM[2]	
CSI2_TXP[3]	Differential data Out – Lane 3
CSI2_TXM[3]	

f. **NERROR:** NERROR_OUT line is used by AWR2243 to notify external processor when fatal error occurs in device.

3. Getting started

The best way to get started with the mmWave DFP is to start running mmWaveLink examples that are provided as part of the package. The example applications are available at '***mmwave_dfp_<ver>\ ti\example***' directory.

These applications demonstrate firmware download on AWR2243 and configuration of mmWave front end using mmWaveLink framework over SPI interface. It allows user to use difference chirp parameters and configure mmWave device using application executable.

For all the Cascade DFP examples, please refer the 'mmwave_studio_cascade_user_guide.pdf' present as a part of mmWaveStudio 3.0.0.14 package for getting started with the cascade system and running the DFP examples.

- To receive the raw data over LVDS, Connect the LVDS Interface of DCA1000 EVM to an external device which has a compatible LVDS. The detail of the external device is out-of-scope for this document.
- And to capture over CSI-2 AWR2243 can be directly connected to compatible external Host device with proper connector.
- Following sections describe the general procedure for connecting the single chip EVM and running the examples.

4. PC Connection

4.1 Connection with AWR2243 ES1.1 BOOST

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 as shown in Figure-5. See EVM User Guide for details on the COM port.

If below COM ports doesn't show up in the Device Manager install the emupack available here.

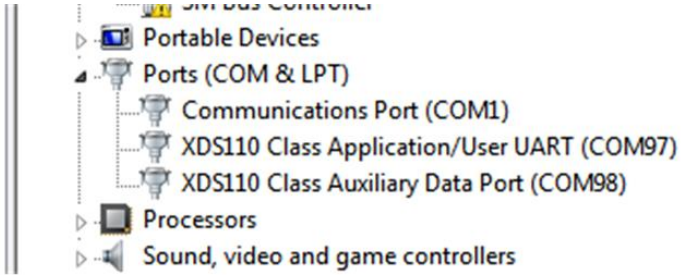


Figure 5. AWR2243-BOOST COM Ports

The USB connection provides below functionalities.

- JTAG for CCS connectivity – Only required for debug purpose. More detail on this is not in the scope of this document.
- UART for downloading firmware in serial data flash using UniFlash. Refer to UniFlash user guides for more details.
- UART for downloading Firmware in SOP-2 (development) mode using mmWave Studio GUI. Refer to mmWave Studio user guide for more details.

USB connection with AWR2243BOOST can be skipped if user wants to run only the mmWaveLink examples provided as a part of the package. It is primarily intended for mmWave Studio GUI and Debug purpose.

4.2 Connection with DCA1000-EVM

For running mmWaveLink_example and mmWave Studio GUI tool, User also needs to connect the DCA1000 along with the EVM. Stack the DCA1000 with the EVM as shown in Figure-6

For more details on DCA1000 connection, see the DCA1000 User Guide.

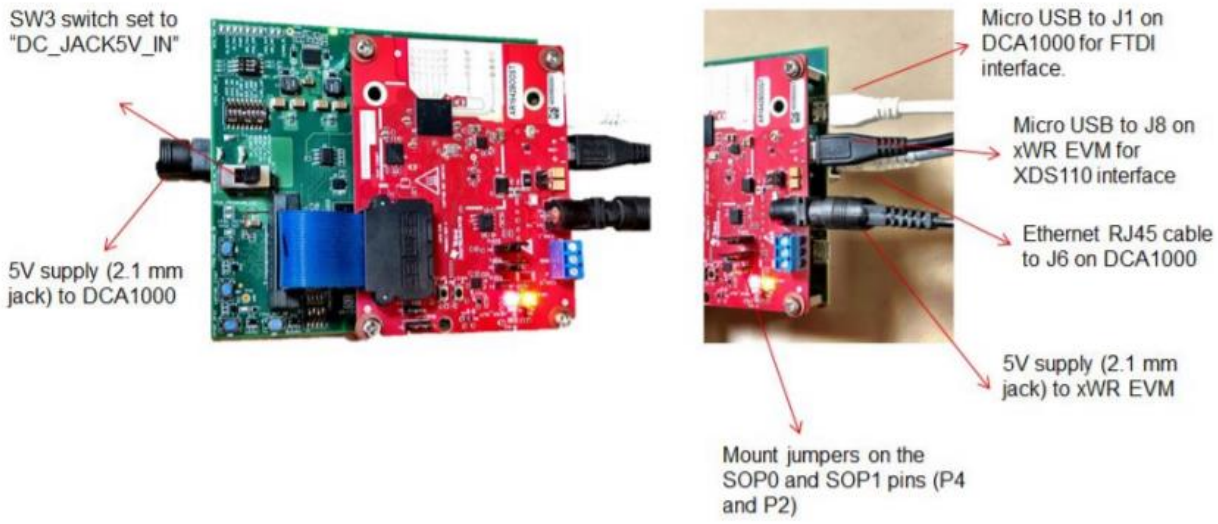


Figure 6. AWR2243 BOOST – DCA1000 Connection

Once stacking is done, connect DCA1000 to PC using the USB cable provided and connect the power cable. Once done, there should be 4 additional COM Ports as shown in Figure-9

When the DCA1000 is connected for the first time to the PC, Windows maybe not be able to recognize the device and would come up as 'Other devices' in device manager as shown in Figure-7.

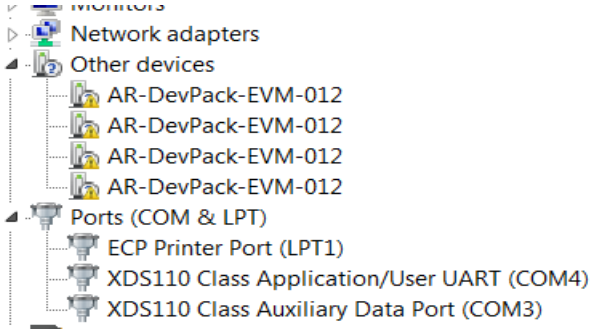


Figure 7. Device Manager (Other Devices)

In Windows device manager, right-click on these devices and update the drivers by pointing to the location of the FTDI driver (part of mmWaveStudio installer) as show in Figure-8

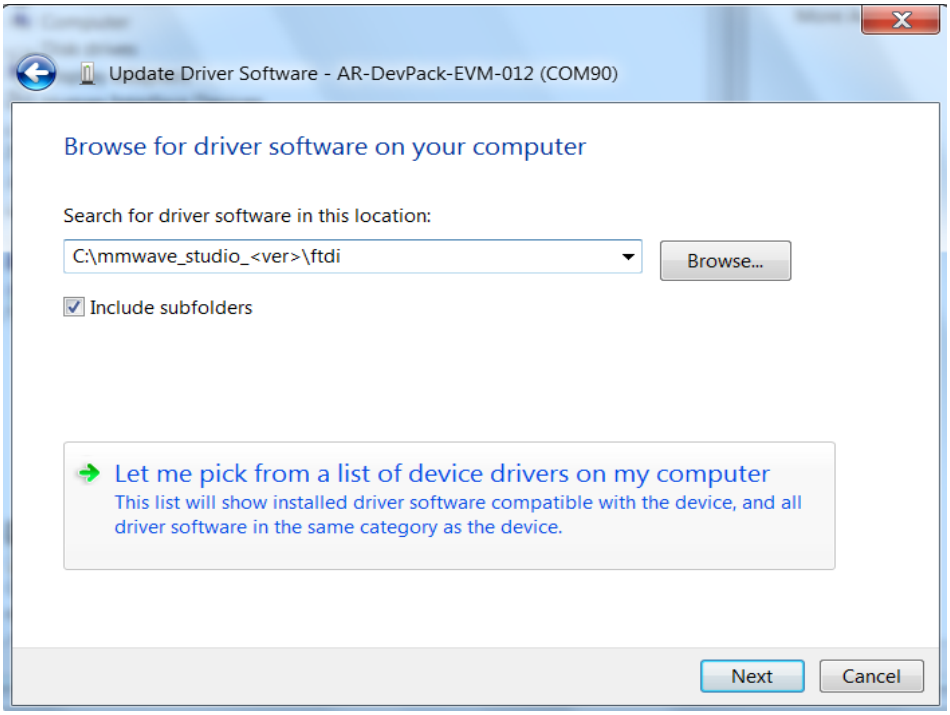


Figure 8. FTDI driver update

This must be done for all four COM ports. If after updating the FTDI driver, device manager still doesn't show 4 new COM Ports, as shown in Figure-9, you would need to update the FTDI driver once again.

Figure 9. Device Manager (USB Serial Port)

When all four COM ports are installed, the device manager recognizes these devices and indicates the COM port numbers, as shown in Figure-10.

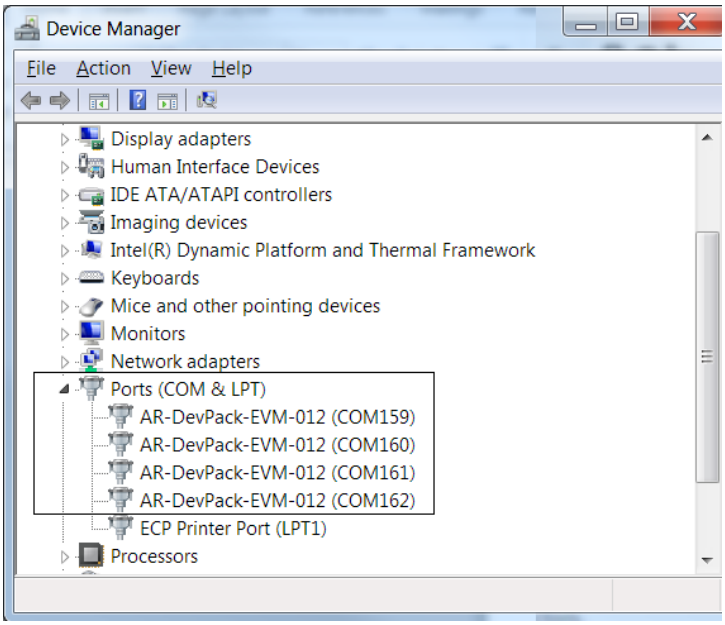


Figure 10. DCA1000 COM Ports

To run mmWaveLink example user must connect AWR2243 EVM and DCA1000 as shown in Figure 5

5. Demonstration Mode

This mode should be used with prebuild executable provided in DFP.

1. Connect AWR2243 EVM and DCA1000 setup to Windows PC as mentioned in section 4.2
2. Open windows command prompt and go to 'mmwave_dfp_<ver>\ti\example\' path.
3. To run mmWaveLink_SingleChip_Example application goto to '*mmWaveLink_SingleChip_Example*' directory and run *mmwavelink_example.exe*
4. To run mmWaveLink_SingleChip_Monitoring application goto to '*mmWaveLink_SingleChip_Monitoring*' directory and run *mmwavelink_monitoring.exe*

Once executed the application reads mmWave radar configuration from mmwaveconfig.txt file and sends it to AWR2243 device over SPI/I2C (based on the interface chosen in the mmwaveconfig.txt). If you prefer to change the configuration, open mmwaveconfig.txt in text editor and update the parameters. For more details on these parameters, refer to doxygen HTML files in '***mmwave_dfp_<ver>\ti\mmwavelink\docs'*** directory.

Note- This application resets the AWR2243 device via DCA1000EVM interface, so you can see NRST LED blink as application starts. In case you observe '-8' error in the application console at the beginning then you may need to press NRST manually then execute the application.

Figure 11 and figure 12 show output screen of this mmWaveLink_SingleChip_Example (default setting).

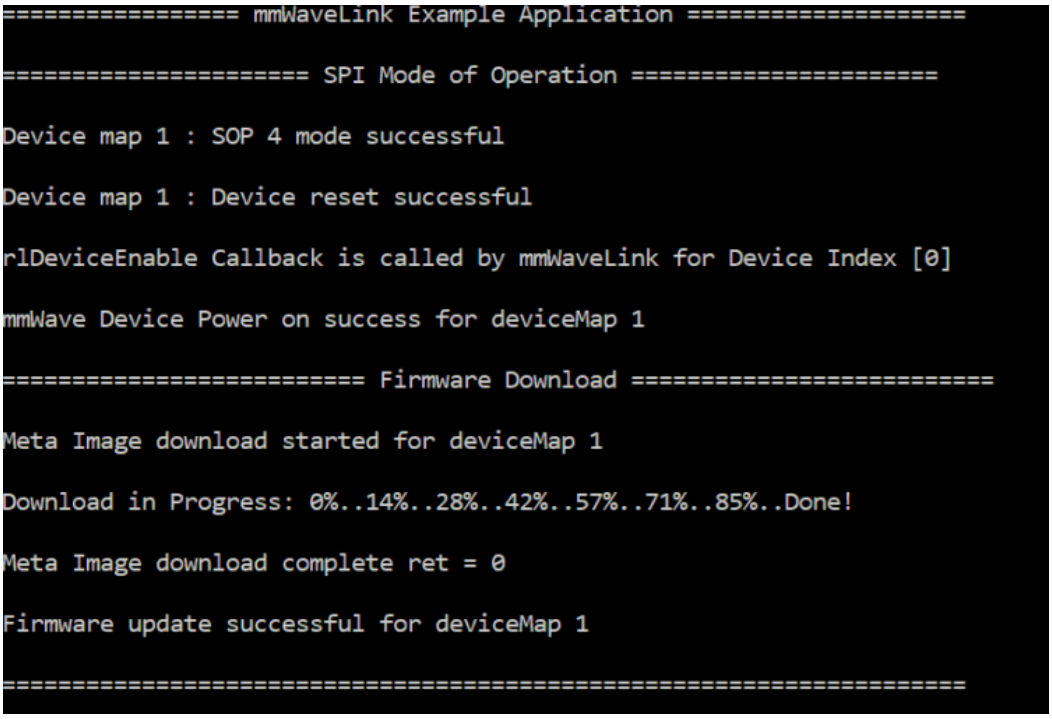


Figure 11. Output Window 1

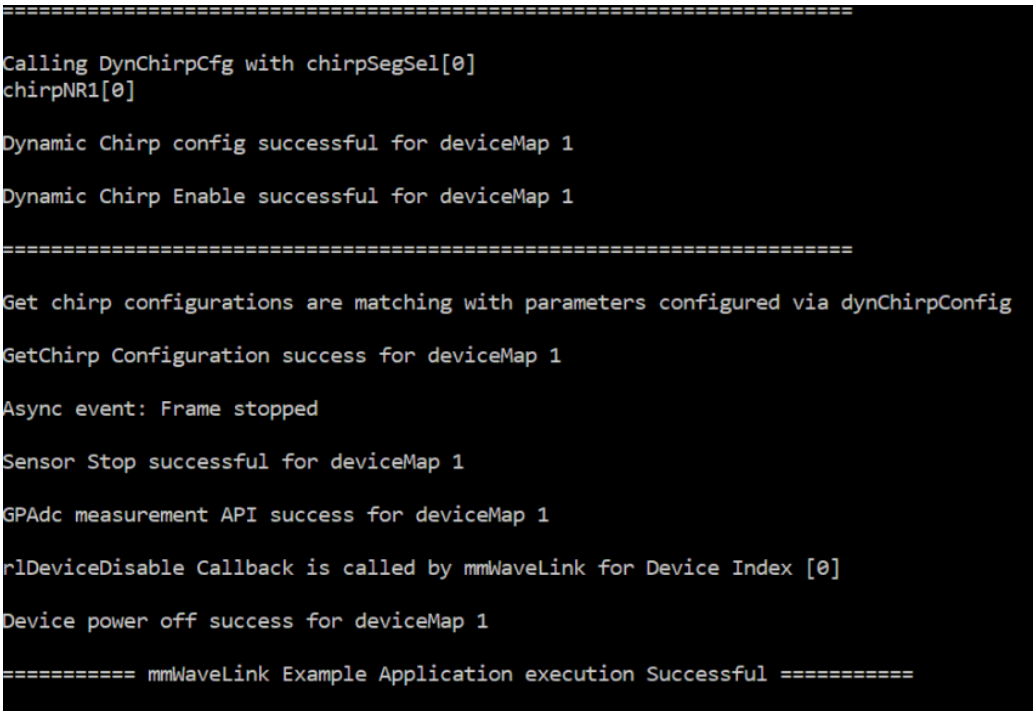


Figure 12. Output Window 2

6. Visual Studio Development Mode

This mode should be used when debugging with Visual Studio is involved and/or developing mmWaveLink application.

1. Install Microsoft Visual Studio 2017 express edition or newer version on windows computer.
2. Connect AWR2243 EVM and DCA1000 setup to Windows PC as mentioned in section 4.2
3. Open Visual Studio IDE and then go for menu option ‘File->Open->Project\Solution’.
4. Select ‘mmwavelink_example.sln’ file from ***mmwave_dfp_<ver>\ti\example\mmWaveLink_SingleChip_Example’*** directory.
5. It will open mmwavelink_example solution containing two projects mmwavelink and mmwavelink_example as given in below picture.

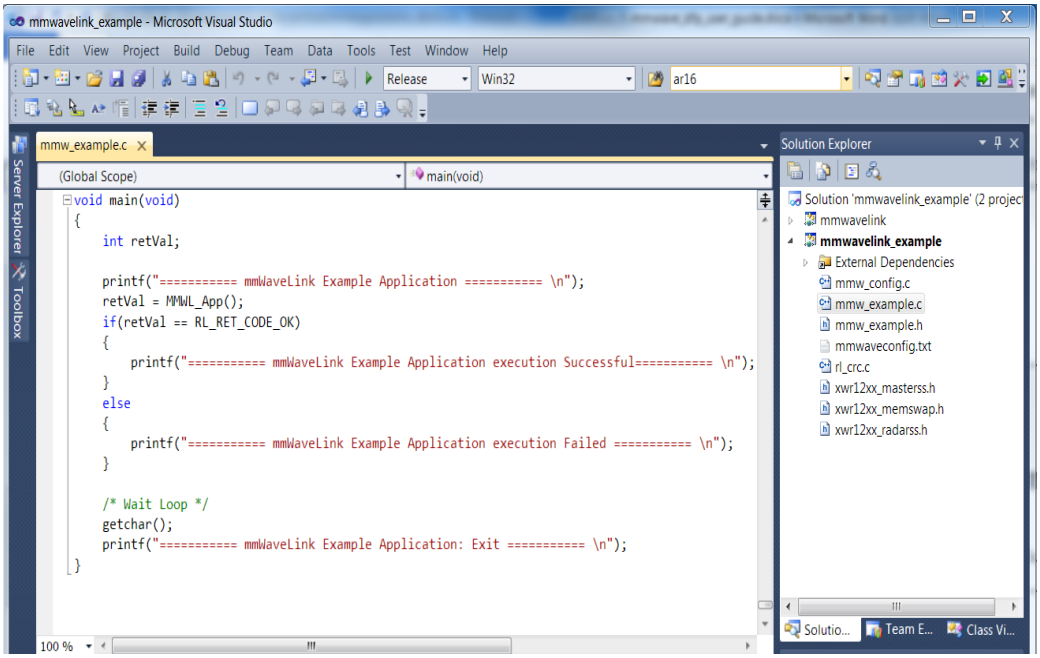


Figure 13. Visual Studio Development mode

6. Click menu option ‘Build->Build Solution’ which will build both projects.
7. To run the application in debugging mode press ‘F5’ key.
8. To debug, put breakpoints in the application source code as required.

Please follow the same steps as mentioned above to open and run the other mmwavelink example Visual Studio projects.

7. Data capture using DCA1000 EVM

As the DFP examples are being executed, we can capture the data from the AWR2243 EVM using the DCA1000 capture card via the DCA1000EVM CLI application.

The DCA1000EVM CLI application is primarily a command line tool for configuration of DCA1000 FPGA and recording based on the user inputs (from a json file). The DCA1000EVM CLI application connects to DCA1000EVM system through Ethernet for configuration and recording of data.

The CLI application (DCA1000EVM_CLI_Control.exe) comes as a part of the mmWaveStudio package (*C:\ti\mmwave_studio_03_00_00_14\mmWaveStudio\PostProc*). The CLI application makes use of the "cf.json" file present at the PostProc folder. This json file contains information regarding the configuration and capture of AWR2243 EVM. For more information on using the CLI extensively, please refer to the CLI user guide present in the mmWaveStudio package (*C:\ti\mmwave_studio_03_00_00_14\mmWaveStudio\ReferenceCode\DCA1000\Docs*) folder.

For getting started, the following commands are issued

1. **fpga** (for connection and configuration of DCA1000 with the PC)

```
C:\ti\mmwave_studio_03_00_00_07\mmWaveStudio\PostProc>DCA1000EVM_CLI_Control.exe fpga cf.json

FPGA Configuration command : Success
```

2. **record** (for providing information on packet delay)

```
C:\ti\mmwave_studio_03_00_00_07\mmWaveStudio\PostProc>DCA1000EVM_CLI_Control.exe record cf.json

Configure Record command : Success
```

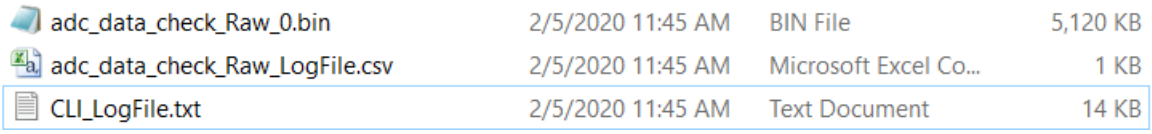
> The commands 1 and 2 can be given even before the execution of the DFP example.
> For starting the capture, the command 3 needs to be given during the execution of the DFP example.

3. **start_record** (for starting the data capture. It closes on its own after a timeout of 2 secs when there is no further data being pumped out from the AWR2243)

```
C:\ti\mmwave_studio_03_00_00_07\mmWaveStudio\PostProc>DCA1000EVM_CLI_Control.exe start_record cf.json

Start Record command : Success
```

   This command can be given roughly while the firmware is being downloaded in the DFP example application. From the time the command is issued, the DCA1000 will be actively looking for data over the LVDS lanes. Once the AWR2243 has stopped framing, and roughly after a timeout of 2 secs, the DCA1000 CLI application closes and the captured data will be available at the location mentioned in the cf.json file. The following picture describes the raw ADC data being captured, a capture log file indicating the statistics of the capture and a CLI log file showing the commands being executed.



8. Introduction to DFP examples

DFP package contains various examples to demonstrate miscellaneous mmWave sensor features.

- For all the DFP examples, there is a 'trace.txt' file created on running the application. It is created at the same directory where the corresponding executable is present. This file can be used for debugging as it logs all the SPI/I2C communication commands.
- Sole purpose of the examples provided in DFP package is to showcase the usage of different mmWaveLink APIs which will further help user to port it on any external processor.
- For all the Cascade DFP examples, please refer the '*mmwave_studio_cascade_user_guide.pdf*' present as a part of mmWaveStudio 3.x package for getting started with the cascade system and running the DFP examples.

```
------ mmwaveconfig.txt ------
#
#Global configuration
#Advanced frame test enable/disable; 1 - Advanced frame; 0 - Legacy frame
#Continuous mode test enable/disable; 1 - Enable; 0 - Disable
#Dynamic chirp test enable/disable; 1 - Enable; 0 - Disable; This should not be enabled
if Advanced chirp test is enabled
#Dynamic profile test enable/disable; 1 - Enable; 0 - Disable
#Advanced chirp test enable/disable; 1 - Enable; 0 - Disable; The legacy chirp API is not
required if this is enabled
#Firmware download enable/disable; 1 - Enable; 0 - Disable
#mmWaveLink logging enable/disable; 1 - Enable; 0 – Disable
#Calibration enable/disable; To perform calibration store/restore; 1 - Enable; 0 -
Disable
#Calibration Store/Restore; If CalibEnable = 1, then whether to store/restore; 1 - Store;
0 - Restore
#Transport mode; 1 - I2C; 0 - SPI
#
LinkAdvanceFrameTest=0;
LinkContModeTest=0;
LinkDynChirpTest=1;
LinkDynProfileTest=0;
LinkAdvChirpTest=0;
EnableFwDownload=1;
EnableMmwlLogging=0;
CalibEnable=0;
CalibStoreRestore=1;
TransferMode=0;
#END
```

8.1 mmWaveLink_SingleChip_Example (SPI / I2C mode of operation)

Along with basic mmwave features, this application showcases some of advanced features such as advanced frame, advanced chirp, continuous mode, dynamic chirp and dynamic profile configuration. User can enable or disable these advanced features in mmwaveconfig.txt.

Once the required settings are done, run mmwavelink_example.exe.

This example supports both SPI and I2C mode of operation. Refer to mmwaveconfig.txt on choosing the mode of operation.

1. Application sets the device in SOP4 mode (in the case of SPI) or SOP7 mode (in the case of I2C).
2. Downloads the meta image over SPI/I2C (based on the mode chosen)
3. API parameters for all the commands are read from mmwaveconfig.txt.

NOTE:

1. "*CalibrationData.txt*" file is created which stores the calibration data. When Calibration restore is issued, it makes use of the data present in this file.
2. "PhShiftCalibrationData.txt" file is created which stores the phase shifter calibration data. When phase shifter Calibration restore is issued, it makes use of the data present in this file.
3. "*AdvChirpLUTData.txt*" file is created which stores the locally programmed LUT data that is sent to RadarSS to populate the LUT at the device end.
4. This example contains code to support both of ES1.0 and ES1.1 for reference.

In order to exercise the I2C control path, 1. Change the 'MMWL_FW_FIRST_CHUNK_SIZE' macro to 220 and 'MMWL_FW_CHUNK_SIZE' to 228 in mmw_example.c file. Re-build the example. 2. Change 'TransferMode' to 1 in mmwaveconfig.txt

8.2 mmWaveLink_SingleChip_Monitoring

This application demonstrates various monitoring features of mmwave sensor device. User can enable or disable the features in mmwaveconfig.txt.

All monitoring reports are received as asynchronous event from the device and handled under MMWL_asyncEventHandler function.

1. Application sets the device in SOP4 mode.
2. Downloads the meta image over SPI.
3. API parameters for all the commands are read from mmwaveconfig.txt
4. Once the device starts framing, the async event reports from the device are logged into 4 different files under the "Reports" directory created at the same location as that of the executable.

a. *MonitoringReport.txt*: This report logs all other async events coming from the device specifically the analog and the digital monitors.
b. *CalibrationReport.txt*: This report logs all the async events related to the calibrations.
c. *MSSEvents.txt*: This report logs all the async events related to the MSS.
d. *BSSEvents.txt*: This report logs all the async events related to the BSS.

8.3 mmWaveLink_SingleChip_NonOS_Example

This example is exactly same as that of mmWaveLink_SingleChip_Example except for the usage of a Non-OS environment. Along with basic mmwave features, this application showcases some of advanced features such as advanced frame, advanced chirp, continuous mode, dynamic chirp and dynamic profile configuration.

User can enable or disable the features in mmwaveconfig.txt, build the application using the visual studio project provided in DFP package and run mmwavelink_example_nonos.exe

Refer to Section 8.1 for more details.

8.4 mmWaveLink_SFlash_FW_Example

Along with basic mmwave features, this application showcases the feature of SPI to sFlash image download on an AWR2243 mmWave device.

1. Application sets the device in SOP4 mode.
2. Downloads the meta image to sFlash over SPI.
3. API parameters for all the commands are read from mmwaveconfig.txt

To modify and re-run the application, use Visual Studio based project provided in the same directory. Refer to Section 8.1 for more details.

NOTE:

Once the flashing operation is successful, it is not necessary to download the meta image in the subsequent power cycles. In order to skip the firmware download for the subsequent iterations, change EnableFwDownload=0 in mmwaveconfig.txt

8.5 mmWaveLink_Cascade_Example

Along with basic mmwave features, this application showcases the configuration and ADC data capture of AWR2243 mmwave cascade chip with the TDA2XX board.

1. Application sets the Master in SOP4 mode.
2. Downloads the meta image over SPI for Master.
3. Application sets all the Slave devices in SOP4 mode.
4. Downloads the meta image over SPI for all the Slaves.

5. API parameters for all the commands are read from mmwaveconfig.txt
6. During framing, the data is captured in the TDA2XX SSD under the folder "MMWL_Capture" (under /mnt/ssd/ directory in TDA2XX file system)
7. Once the capture is complete, user can retrieve the data from the SSD using the WinSCP application.

To modify and re-run the application, use Visual Studio based project provided in the same directory.

NOTE:

1. Application by default has the "Parallel-SPI" feature enabled.
2. Application by default has all the devices enabled. User can control the devices to be configured via 'CascadeDeviceMap' field in mmwaveconfig.txt.
3. Application by default captures the data under the folder "MMWL_Capture" (under the /mnt/ssd/ directory of TDA2XX file system). It is recommended to change the directory name for every capture.

```
------ mmw_example.c -----
/* Capture Directory */
static char mmwl_TDA_CaptureDirectory[] = "/mnt/ssd/MMWL_Capture";
```

8.6 mmWaveLink_Cascade_Monitoring

This application showcases the monitoring of AWR2243 mmWave Cascade chip with the TDA2XX board.

1. Application sets the Master in SOP4 mode.
2. Downloads the meta image over SPI for Master.
3. Application sets all the Slave devices in SOP4 mode.
4. Downloads the meta image over SPI for all the Slaves.
5. API parameters for all the commands are read from mmwaveconfig.txt
6. Once the device starts framing, the async event reports from the device are logged into 4 different files under the "Reports" directory (created at the same location as that of the executable) separately for each device.
   a. *MonitoringReport.txt*: This report logs all other async events coming from the device specifically the analog and the digital monitors.
   b. *CalibrationReport.txt*: This report logs all the async events related to the calibrations.
   c. *MSSEvents.txt*: This report logs all the async events related to the MSS.
   d. *BSSEvents.txt*: This report logs all the async events related to the BSS.

To modify and re-run the application, use Visual Studio based project provided in the same directory.

NOTE:

1. Application by default has the “Parallel-SPI” feature enabled.
2. Application by default has all the devices enabled. User can control the devices to be configured via ‘CascadeDeviceMap’ field in mmwaveconfig.txt.

9. API Sequence

Below Figure shows the API sequence in mmwavelink_example (default setting). For more details on these APIs, refer to mmWaveLink framework doxygen HTML files in ‘***mmwave_dfp_<ver>\ti\mmwavelink\docs’*** directory.

9.1 AWR2243 ES1.1 Single chip API sequence

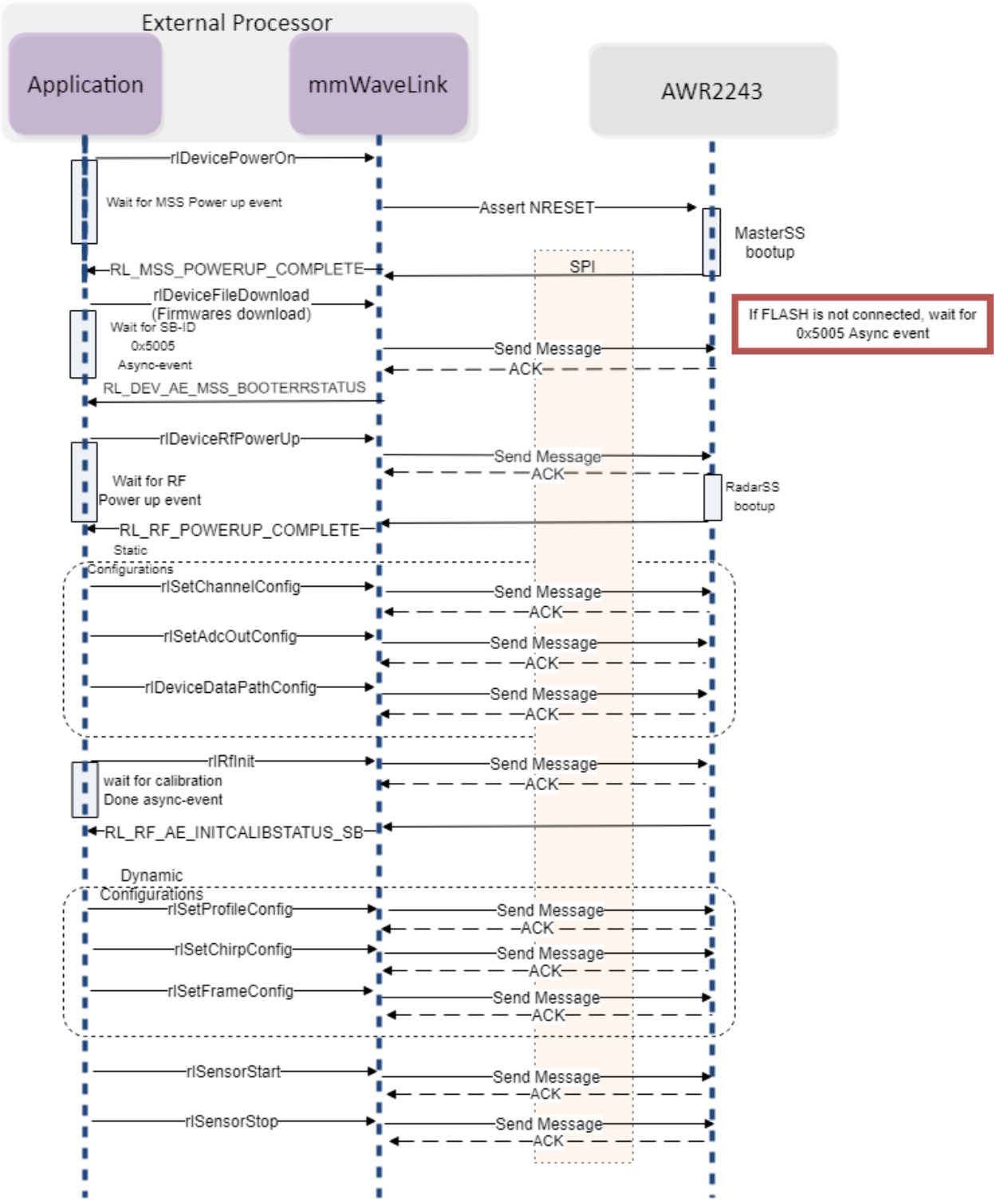


Figure 14. Single chip API Sequence

9.2 AWR2243 ES1.1 Two/Four chip API sequence

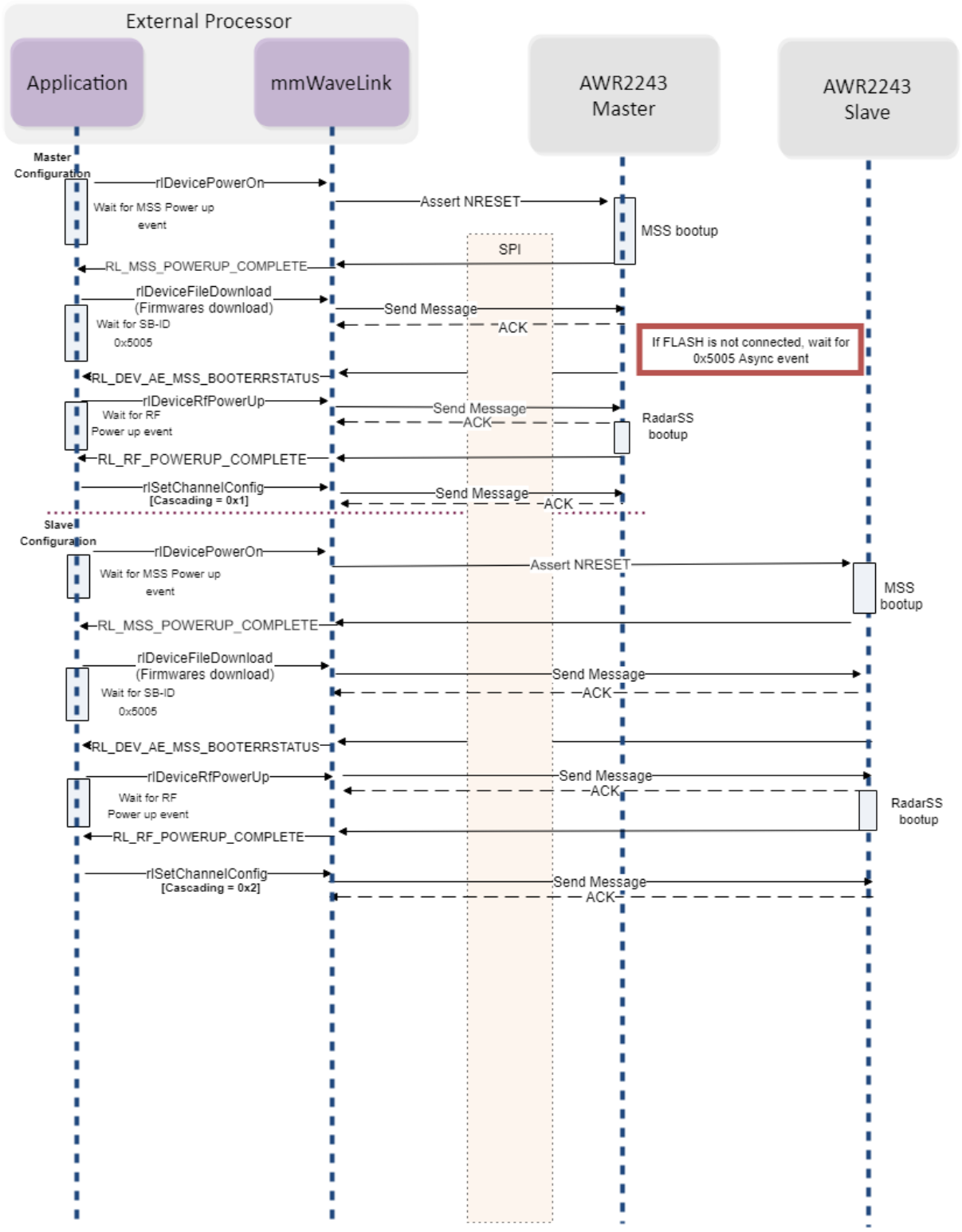


Figure 15. Cascade API Sequence-1

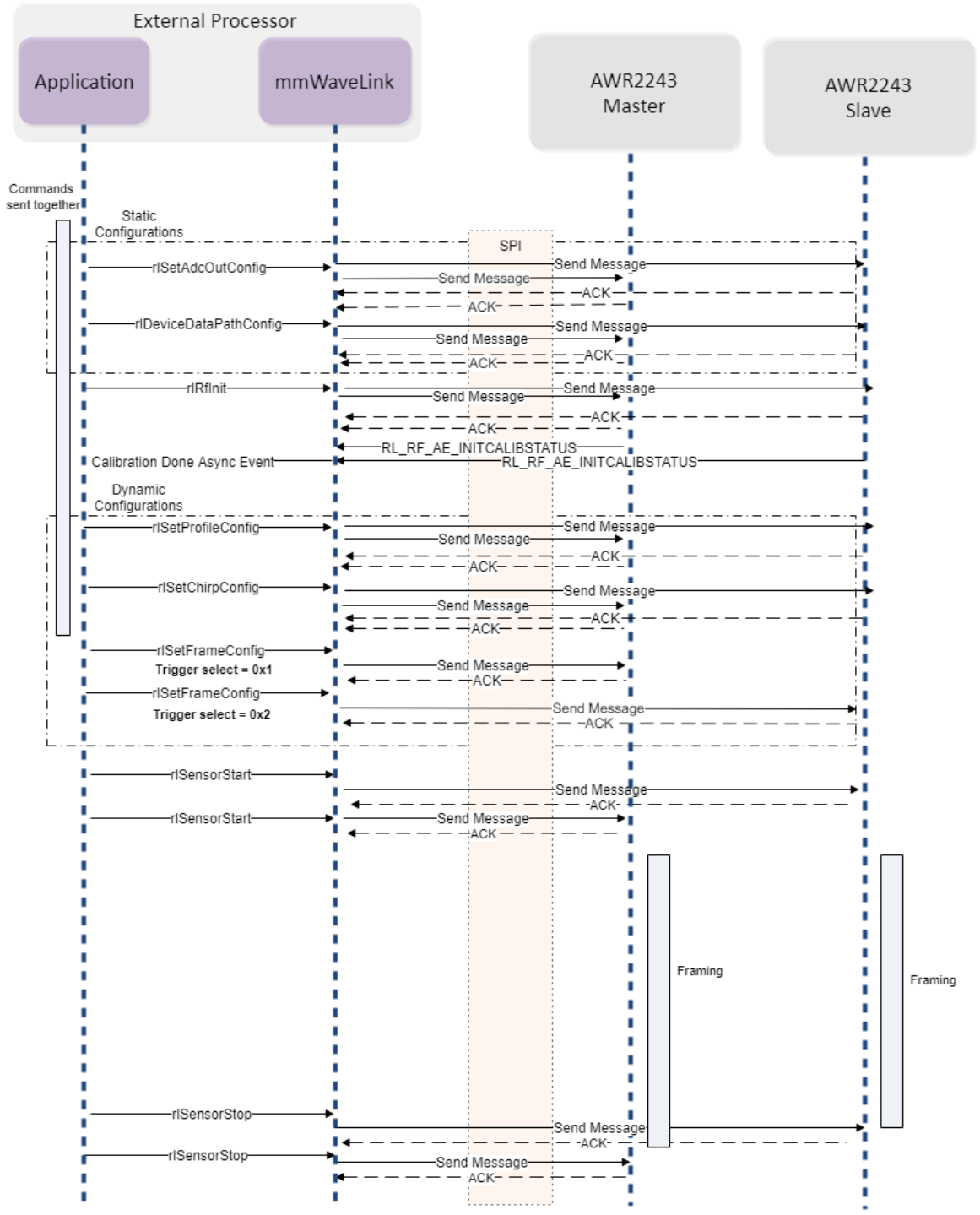


Figure 16. Cascade API Sequence-2

10. Additional dependency

This mmWaveLink example application on windows PC uses the following libraries to communicate with AWR2243.

1. **mmwl_port_ftdi.lib**: For single chip systems
2. **mmwl_port_ethernet.lib**: For cascade chip systems

The libraries provide implementation of all the platform dependent routines required by TI mmWaveLink framework. Broadly it implements below platform interfaces as mentioned in mmWaveLink porting guide.

- Communication Interface
- Device Control Interface
- OS Interface

The libraries are available at ***'mmwave_dfp_<ver>\ti\example\platform'*** directory.

To port mmWaveLink framework to a new processor, developer need to provide similar interfaces as implemented on a windows platform using these libraries.

For more information on porting steps, refer to mmWaveLink framework doxygen HTML files in '***mmwave_dfp_<ver>\ti\mmwavelink\docs'*** directory.

Note: The source code for these libraries is available in the mmWave Studio package for reference.
'mmwave_studio_03_00_00_<ver>\mmWaveStudio\ReferenceCode\FTDILib\' directory.
'mmwave_studio_03_00_00_<ver>\mmWaveStudio\ReferenceCode\EthernetLib\' directory.

11. mmWave Sensor Advanced Features

TI's mmwave sensor devices provide large flexibility in configuring chirp parameters. Many complex use-cases require dynamic reconfiguration of device within a frame or while device is framing. Below mentioned APIs can be used to cater those use-cases. The usage of advanced APIs is explained based on multiple possible use-cases we have come across but there are still other possibilities which may not covered here. These APIs are explained with the all required timing and existing limitation w.r.t. different device variants.

This can referred on top of another app-note (https://www.ti.com/lit/an/swra553a/swra553a.pdf) which explains the fundamental programming chirp parameters in TI mmwave sensor device.

11.1 Advanced frame configuration

mmWave sensor provides an advanced option for frame configuration. In this configuration, user can configure multiple type of frames compare to legacy frame configuration where only one type of frame is permitted. Advanced frame configuration allows up to four sub-frames, with multiple bursts in each sub-frame.

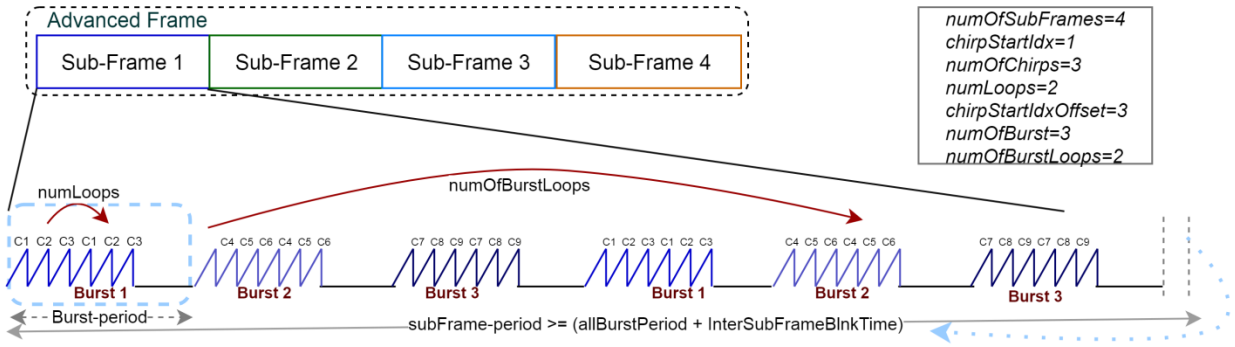


Figure 17. Example of Advanced Frame

As you have seen that based on the desired application the chirp configurations need to be set differently. But what if we need to support multiple modes, for example ultra-short range, short range and/or mid-range, simultaneously using a single radar device? The advanced frame configuration available in TI's mmwave sensor allows for large flexibility to have multiple chirp configurations in a single frame. The frame can be constituted using a sequence of "sub-frames" with each of these sub-frames representing a different radar mode. The *rlSetAdvFrameConfig* API helps enable this kind of configuration.

To provide maximum flexibility of the chirps within a frame, the advanced frame configuration API provides the ability to break a frame into different sub frames (up to 4). Each of the sub frames consists of multiple bursts of chirps (up to 512 bursts). Each burst can consist of 512 unique chirps which are associated to one of the 4 profiles, and the start chirp index can be programmed to have a fixed offset from the previous burst. A set of bursts in a sub-frame can

be further looped in software up to 64 times. The below figure is an example how a sub frame is formed.

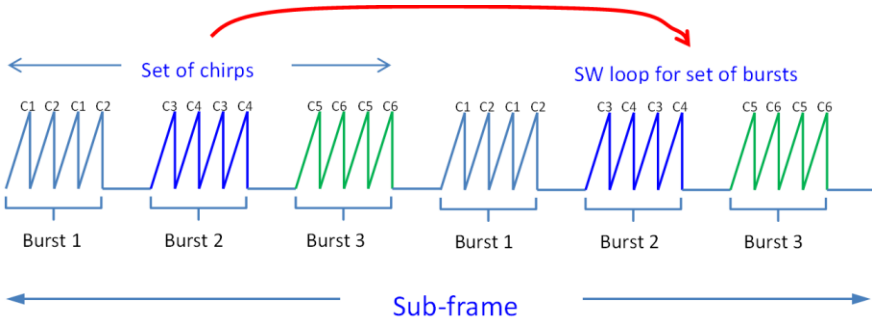


Figure 18. Sub Frame Structure Showing Three Bursts Looped Twice (start index having an offset of 2)

The frame is formed by a maximum of four such sub frames, and each of the sub frames can have a different set of chirps. The different chirps can also use different transmitters (which could possibly have different antenna configurations).

The timing requirements for the advanced frame are as follows: Inter-burst time should be ≥ 50 µsec, inter sub-frame time should be ≥ 100 µsec and inter frame time should be ≥ 200 µsec.

Note- all the variable/parameter used here to explain the concepts are based on mmWaveLink.

Creating a Burst

A burst can have max of 512 unique chirps in it but can be looped over these chirps to generate a larger sequence. In a burst, index of chirp starts with chirpStartIdx and index increments till numOfChirps, this chirp sequence can be repeat by numLoops in a single burst. In the next burst, variation in chirp index can be done by setting chirpStartIdxOffset. As depicted in last figure, chirp index for the 2nd burst will be 4 when 'chirpStartIdxOffset=3'.

[next_burst_chirp_start_idx = last_busrt_chirpStartIdx + chirpStartIdxOffset]

Burst Periodicity

Burst periodicity should be accounted with extra blank time along with total chirp time. User needs to set the burst periodicity based on calculation below to avoid any error where *InterBurstBlankTime* is required for sensor calibration/monitoring.

*[BurstPeriodicity >= (numOfChirp * numLoops) + InterBurstBlankTime]*

Sub-Frame Periodicity

While setting the sub-frame periodicity in advanced frame user needs to provide extra time [*InterSubFrameBlankTime*] which is required for sensor calibration/monitoring, transferring out any safety data, hardware reconfiguration and trigger the next sub frame. Sensor may throw an error while triggering the advanced frame, if sufficient blank time is not provided.

*[BurstPeriodicity >= (numOfChirp * numLoops) + InterBurstBlankTime]*

Scenario	InterSubFrameBlankTime (usec)
Default	>= 100
Loopback configuration enable	>= 350
Test source configuration enable	+150

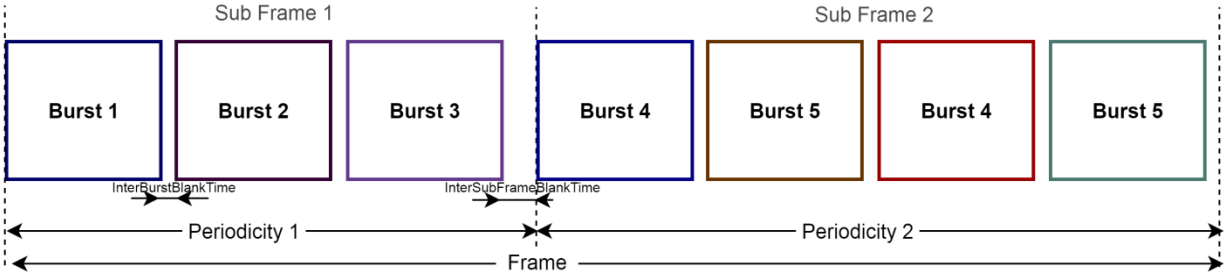


Figure 19. Example of Advanced Frame Configuration of Two Sub Frames

Override the existing Chirp Profile Index

It may possible that each chirp is connected to different profile index as set by rlSetChirpConfig API. So with this condition, chirps comprised in a sub-frame may be connected to different profile index. But advanced frame API provides an option to unify profile index in a sub-frame, i.e. all the chirps in a sub-frame can be connected to single profile index.

Set rlAdvFrameSeqCfg_t->forceProfile to '1' which enables the option to select specific profile index for the burst and set rlFrameCfg_t->forceProfileIdx to required profile index for that sub-frame. It will override the profile-ID selection for all the chirps in sub-frame/burst.

NOTE:

1. This setting won't overwrite the profile-index in chirp RAM, it will be effective only during this sub-frame.

Loopback configuration

Refer section for Loopback APIs.

Sub-Frame Trigger

By default, all the configured sub-frame will be triggered internally by the device. But if some use-case requires triggering the sub-frame manually, then RadarSS provides software and hardware triggered mode options in advanced frame to trigger each sub-frame. Host needs to invoke rlSensorStart API if SW triggered mode is set, else provide SYNC_IN pulse in HW triggered mode to set off each sub-frame. Host must need to provide these trigger events beyond last sub-frame boundary.

Following are the set of configurations for trigger option

Configuration	Action
triggerSelect=1, subFrameTrigger=0	rlSensorStart only at first frame.
triggerSelect=1, subFrameTrigger=1	rlSensorStart at every sub-frame boundary.
triggerSelect=2, subFrameTrigger=0	SYNC_IN pulse at every burst boundary.
triggerSelect=2, subFrameTrigger=1	SYNC_IN pulse at every sub-frame boundary.

11.2 Advanced chirp configuration

It provides the ability to program fixed delta increment for certain chirp parameters (chirp start frequency, idle time, phase shifter, etc.), on top of unique dithers selected from configurable look-up-table (LUT). The configurable look-up-table is an array of values loaded into a pre-configured “Generic SW Chirp Parameter LUT”. The size of the generic LUT is 12KB and user has the flexibility to program any number of unique dithers for each chirp parameters. Thus the user can achieve fixed increment, or LUT based dither, or a combination of both.

Four types of control can be achieved on each parameters of a chirp

1. Fixed value for all the chirps: To generate sequence of chirps which never changes, then only one value can be programmed in LUT (P =1 and K = 0).
2. Unique chirps: Index every K chirps in LUT to generate unique sequence of chirps.
3. Delta increment every N chirps: On top of sequence of unique chirps from LUT, the fixed delta increment can be done every N chirps.
4. The set of chirp parameters across bursts and sub-frames can be different by using offset to LUT.

There are total 10 chirp parameters that the advance chirp API can configure.

Chirp Parameter Index	Description
0	CHIRP PROFILE SELECT
1	CHIRP FREQ START VAR
2	CHIRP FREQ SLOPE VAR
3	CHIRP IDLE TIME VAR
4	CHIRP ADC START TIME VAR
5	CHIRP TX EN
6	CHIRP BPM VAL
7	TX0 PHASE SHIFTER
8	TX1 PHASE SHIFTER
9	TX2 PHASE SHIFTER

It is highly recommend referring ICD document first to get the basic knowledge of this API. Here we exemplify this API using two different use-cases which explain many of complex concepts of this API.

Advanced chirp feature cannot be used with legacy chirp API as this API includes legacy API configurations along with additional complex features. Host needs to enable this feature first using rlRfSetMiscConfig API (AWR_RF_MISC_CTL_SB).

Example Illustration – 1

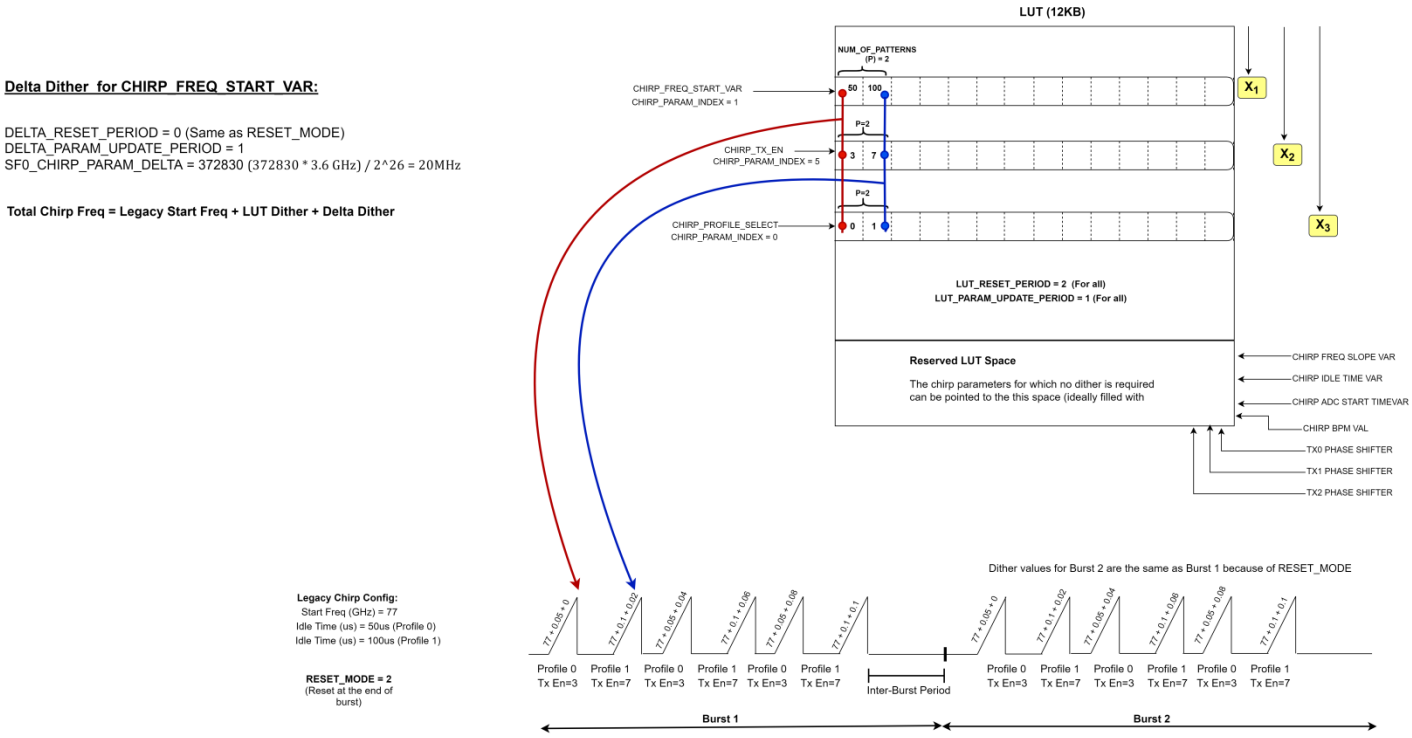


Figure 20. Advanced chirp – Example Illustration 1

In this figure, dither for 3 chirp parameters is considered:

a. *CHIRP_TX_EN*
b. *CHIRP_FREQ_START_VAR*
c. *CHIRP_PROFILE_SELECT (CHIRP_PARAM_INDEX = 0)* : Start address offset = X_3

Rest of the chirp parameters for which no dither is required can point to the “Reserved LUT space” as shown in the figure. This space is ideally filled with zeros.

In the figure, *RESET_MODE = 2* means Reset at the end of every burst. This is the reason why burst 1 and burst 2 are identical in the figure.

Now let’s understand each of the three dithers one by one:

1. CHIRP_TX_EN (CHIRP_PARAM_INDEX = 5)
   This Chirp Parameter dither is possible only through the LUT (Refer to the ICD).

   a. Start address offset of this chirp parameter in the LUT or the *LUT_PATTERN_ADDRESS_OFFSET* = X_2

   b. As it can be seen from the LUT, only two values are programmed for this chirp parameter , i.e. “3” and “7”, *NUM_OF_PATTERNS (P) = 2*. The values in the LUT denote 2 TX and 3 TX enabled simultaneously.

   c. Chirp 1 of Burst 1 takes the value from the 0th index of the LUT (Red marked arrow), whereas Chirp 2 of burst 1 takes the values from the 1st index (Blue marked arrow). Hence *LUT_PARAM_UPDATE_PERIOD (K) = 1*. This field updates the chirp parameter with the new value from LUT after every K chirps.

   d. It can be seen from the figure, although the default *RESET_MODE = 2* (Reset at the end of every burst), reset of the LUT values is happening after every 2 chirps. The 3rd chirp shows the same value of *CHIRP_TX_EN* as the 1st chirp. This means, in addition to the default reset period, reset is happening after every 2 chirps from the LUT. Hence, *LUT_RESET_PERIOD (J) = 2*.

   e. *LUT_CHIRP_PARAM_SCALE* and *LUT_CHIRP_PARAM_SIZE* are not applicable for this Chirp Parameter (Refer to ICD).

2. CHIRP_PROFILE_SELECT (CHIRP_PARAM_INDEX = 0)
   Again, this Chirp Parameter dither is possible only through the LUT (Refer to the ICD).

   a. *LUT_PATTERN_ADDRESS_OFFSET* = X_3

   b. Again, *NUM_OF_PATTERNS (P) = 2* as only two values are programmed for this chirp parameter, i.e. “0” and “1”.

      Definition of Profile 0:
      Start Frequency (GHz) = 77
      Idle Time (us) = 50

      Definition of Profile 1:
      Start Frequency (GHz) = 77
      Idle Time (us) = 100

   c. Chirp 1 of Burst 1 takes the value from the 0th index of the LUT (Red marked arrow)and shows an idle time as 50us, whereas Chirp 2 of burst 1 takes the values from the 1st index (Blue marked arrow) and shows a higher idle time of 100us. Hence *LUT_PARAM_UPDATE_PERIOD (K) = 1*.

   d. It can be seen from the figure, in addition to the default *RESET_MODE*, reset is also happening after every 2 chirps from the LUT. Hence, *LUT_RESET_PERIOD (J) = 2*.

e. *LUT_CHIRP_PARAM_SCALE* and *LUT_CHIRP_PARAM_SIZE* are not applicable for this Chirp Parameter also (Refer to ICD).

3. CHIRP_FREQ_START_VAR (CHIRP_PARAM_INDEX = 1)
This chirp parameter has both the types of dither, i.e LUT and Delta.

LUT dither:

a. *LUT_PATTERN_ADDRESS_OFFSET* = X_1

b. *NUM_OF_PATTERNS (P) = 2* as only two values are programmed for this chirp parameter, corresponding to “50MHz” and “100MHz”.

c. The chirp parameter is updated with the new value from LUT after every 1 chirp, hence *LUT_PARAM_UPDATE_PERIOD (K) = 1.*

d. In addition to the default RESET_MODE, reset is also happening after every 2 chirps from the LUT. Hence, *LUT_RESET_PERIOD (J) = 2.*

e. *LUT_CHIRP_PARAM_SIZE = 0,* which corresponds to a size of 4 bytes in the LUT. Based on the frequency start value, this parameter needs to be set accordingly to accommodate within size (1/2/4 Bytes).

f. *LUT_CHIRP_PARAM_SCALE* is not used in this case, hence its value is 0.

Delta Dither:

a. *DELTA_RESET_PERIOD = 0*, i.e. same as RESET_MODE value (Reset at the end of every burst)

b. Let’s accumulate fixed delta dither every single chirp, hence *DELTA_PARAM_UPDATE_PERIOD = 1*

c. In the figure, 20MHz of start frequency delta dither is accumulated every chirp, hence *SF0_CHIRP_PARAM_DELTA = 20MHz* (372830 * 3.6 GHz) / 2^26)

To summarize,
Total Chirp Start Frequency = Legacy start frequency + LUT Dither + Delta Dither

	Chirp	Legacy Start Freq (GHz)	LUT dither (GHz)	Fixed delta dither (GHz)
Burst 1	0	77	0.05	0
	1	77	0.10	0.02
	2	77	0.05	0.02*2
	3	77	0.10	0.02*3
	4	77	0.05	0.02*4
	5	77	0.10	0.02*5
Burst 2	Same as Burst 1 (RESET_MODE = 2)			

Example Illustration - 2

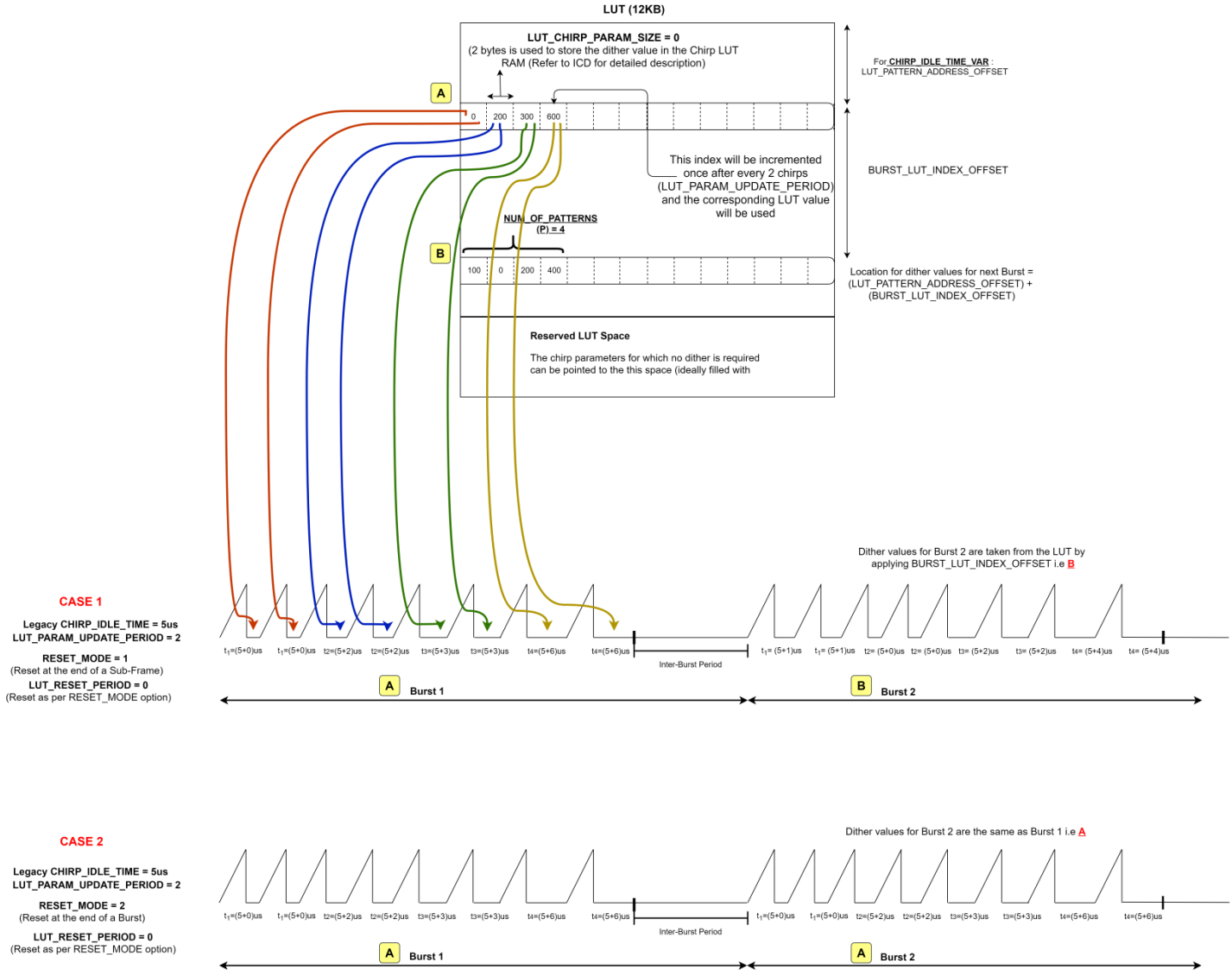


Figure 21. Advanced chirp – Example Illustration 2

This figure considers the variation of just one Chirp Parameter i.e. *CHIRP_IDLE_TIME_VAR*. As only Idle time Variation is considered in this example, all the rest of the chirp parameters point to the reserved LUT Space (filled with zeros).

For *CHIRP_IDLE_TIME_VAR*, only LUT Dither is considered in this example.

Let's Consider Pattern A first. As it is evident from the above figure, and working again like previous figure, we can infer:

a. *LUT_PATTERN_ADDRESS_OFFSET = X*

b. *NUM_OF_PATTERNS (P) = 4* as four values are programmed for this chirp parameter, corresponding to “0”, “200”, “300” and “600”. (1 LSB =10ns, Hence, the dithers correspond to 0us, 2us, 3us and 6us)

c. It can be seen from the figure that each of the four colors (red, blue, green and yellow) point to two chirps simultaneously. The chirp parameter is updated with the new value from LUT after every 2 chirp, hence *LUT_PARAM_UPDATE_PERIOD (K) = 2* in this case.

d. *LUT_CHIRP_PARAM_SIZE = 0,* which corresponds to a size of 2 bytes in the LUT (Refer to ICD)

e. *LUT_CHIRP_PARAM_SCALE* is not used in this case, hence its value is 0.

CASE 1:

Here *RESET_MODE = 2*, i.e. reset at the end of every burst and the *LUT_RESET_PERIOD = 0*, which points to the default reset mode only (assuming a burst is considered which has only 8 configured chirps). As it can be seen from the figure, the dither values for burst 2 are the same as burst 1 i.e. pattern A.

CASE 2:

Now as it is noticed, there are two idle time dither rows in the LUT. The lower row in the LUT corresponds to pattern B.

Here *RESET_MODE = 1*, i.e. reset at the end of every sub-frame and the *LUT_RESET_PERIOD = 0*, which points to the default reset mode only (assuming a burst is considered which has only 8 configured chirps). Let’s assume there are 2 bursts in this sub-frame, because reset happens only at the end of a sub-frame, both these bursts would be different.

Pattern B corresponds to the idle time dither values for Burst 2:

a. *Location of dither values for the next burst = LUT_PATTERN_ADDRESS_OFFSET + BURST_LUT_INDEX_OFFSET = X+Y***.** Parameter *BURST_LUT_INDEX_OFFSET* of this API needs to be set to ‘Y’ so at every next burst of sub-frame, dither offset is added by ‘Y’.

b. *NUM_OF_PATTERNS (P) = 4* as four values are programmed for this chirp parameter, corresponding to “100”, “0”, “200” and “400”. (1 LSB =10ns, Hence, the dithers correspond to 1us, 0us, 2us and 4us)

c. Again, the chirp parameter is updated with the new value from LUT after every 2 chirp, hence *LUT_PARAM_UPDATE_PERIOD (K) = 2.*

d. *LUT_CHIRP_PARAM_SIZE = 0,* which corresponds to a size of 2 bytes in the LUT (Refer to ICD)

e. *LUT_CHIRP_PARAM_SCALE* is not used in this case, hence its value is 0.

Now looking at the chirp diagram for Case 2, we can infer that burst 1 uses Pattern A, whereas burst 2 of the sub-frame uses pattern B.

This way, by configuring different *RESET MODES*, we can have different dithered bursts or sub-frames.

Limitation

- The first chirp in a burst (AFC) or in a legacy frame shall be discarded due to HW limitation in below cases:
    - If start frequency dither is negative for 2nd chirp of a burst/frame (either due to delta increment or due to LUT value).
      Workaround: program a small negative dither for 1st chirp in LUT.
    - If start frequency dither is >= +/-450MHz for 2nd chirp of a burst/frame (either due to delta increment or due to LUT value).
      Workaround: Discard 1st chirp data.
    - If slope dither is negative for 2nd chirp of a burst/frame (either due to delta increment or due to LUT value).
      Workaround: program a small negative dither for 1st chirp in LUT.
    - If slope dither is >= +/-3MHz/us for 2nd chirp of a burst/frame (either due to delta increment or due to LUT value).
      Workaround: Discard 1st chirp data.
- The minimum chirp duration or cycle time shall be 25us if advance chirp feature is used (vs 13us in case of legacy chirp configuration API is used).

Following figures depict the API flow to configure different parameters of advanced chirp. Please refer the DFP examples to find out more.

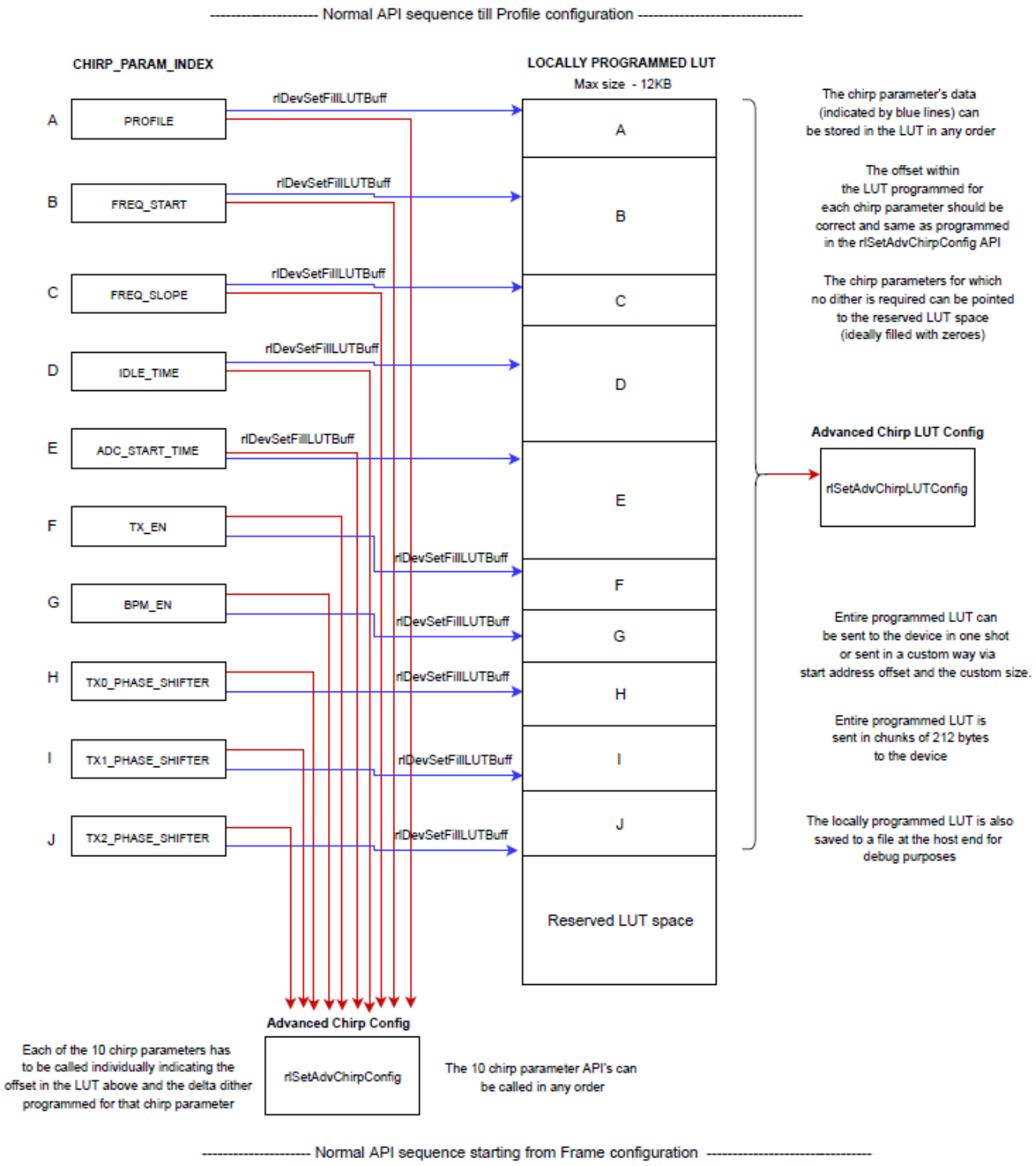


Figure 22. Advanced chirp – Big picture

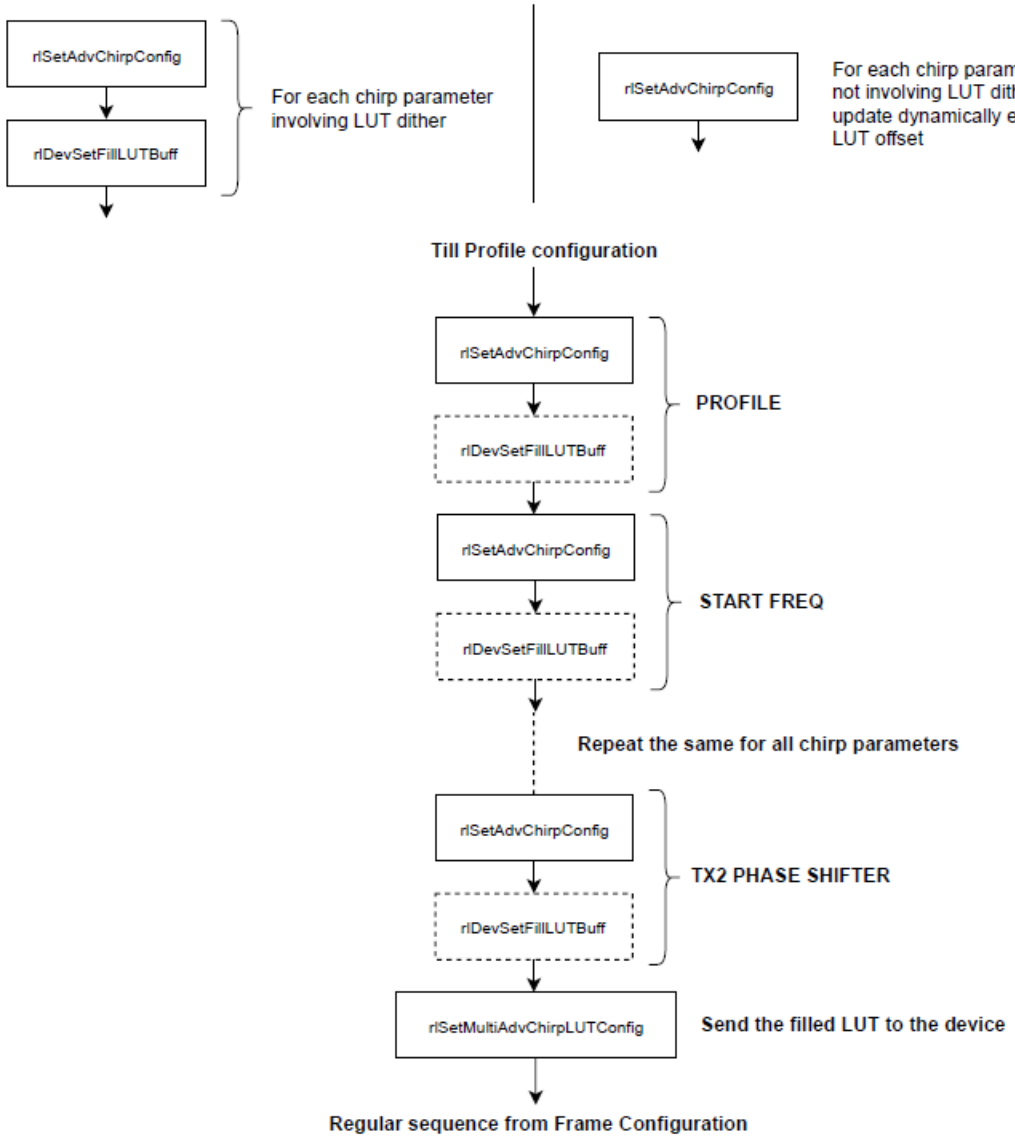


Figure 23. Advanced chirp – Flow diagram

11.2.1 Dynamic Dither (delta/LUT) Update

If at every next frame few of chirp parameters need to be updated dynamically then application needs to set LUT with new data at different offset (rlSetAdvChirpLUTConfig) and invoke rlSetAdvChirpConfig to connect this parameter to new offset (lutPatternAddressOffset). And on top of that it needs to call rlSetDynChirpEn API which actually applies new changes for next frame. All these three APIs must need to be called within current active-frame boundary only then new LUT dither will be applied to next frame.

11.3 Dynamic chirp configuration

There are scenarios where different chirp parameters need to update on the go. Updating the Chirp RAM is possible using rlSetChirpConfig API but this API is slow compare to frame periodicity to apply the changes which may break the device flow.

rlSetDynChirpCfg API can be used to dynamically change the chirp configuration while frames are on-going without much delay. The configuration will be stored in software using rlSetDynChirpCfg API and at rlDynChirpEnCfg API invocation, RadarSS copies these chirp configurations from SW RAM to HW RAM at the end of current on-going frame.

This API provides much flexibility to accommodate different type of chirp parameters of dissimilar chirp index/segment within single API which effectively speed up the configuration time for whole chirp RAM (512 unique chirps). This API gives direct access to device software Chirp RAM which is divided into different segments contains set of chirps. Each chirp in the RAM is divided into three rows.

Chirp row 1: Profile index, frequency slope, Tx enable, BPM constant.

Chirp row 2: frequency start var.

Chirp row 3: idle time, ADC start time.

If user wishes to reconfigure the entire chirp parameters i.e. fill all the chirp rows, then single rlSetDynChirpCfg API can configure 16 unique continuous chirps on the device RAM. This way it takes only 32 API calls to configure whole chirp RAM.

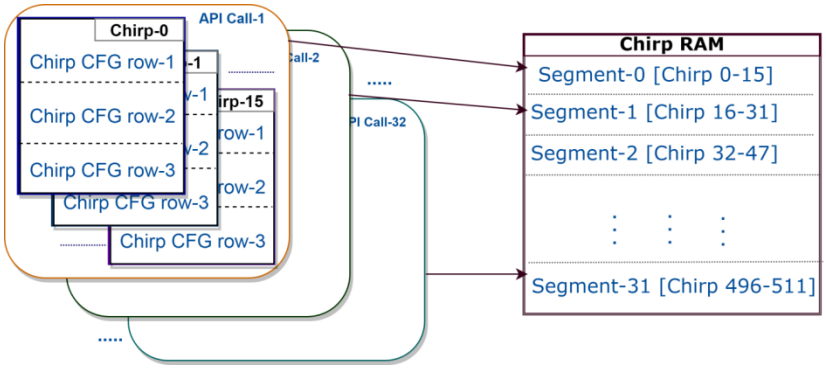


Figure 24. Chirp row select = 0

If user does not wish to reconfigure all 3 chirp rows, then the following mode can be used to configure only one row per chirp which enables the user to configure 48 chirps in one API, effectively saving on the reconfiguration time.

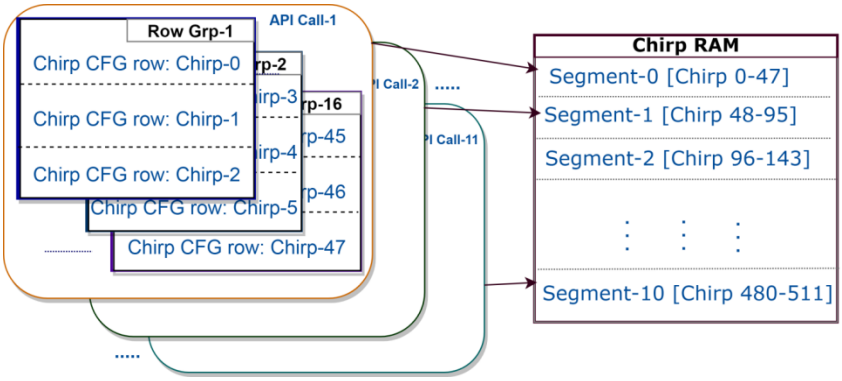


Figure 25. Chirp row select = 1/2/3. (Not using legacy chirp at the beginning)

All these chirp configurations are stored to software chirp RAM, to apply to HW chirp RAM device provides two programming modes.

In program mode 0, a separate API rlSetDynChirpEn needs to be called where device will re-configure the HW only at the end of current active frame boundary. In this case, application can invoke rlSetDynChirpCfg at any time during the frame and those chirp configurations will be stored at software memory but rlSetDynChirpEn API should be called within active frame time.

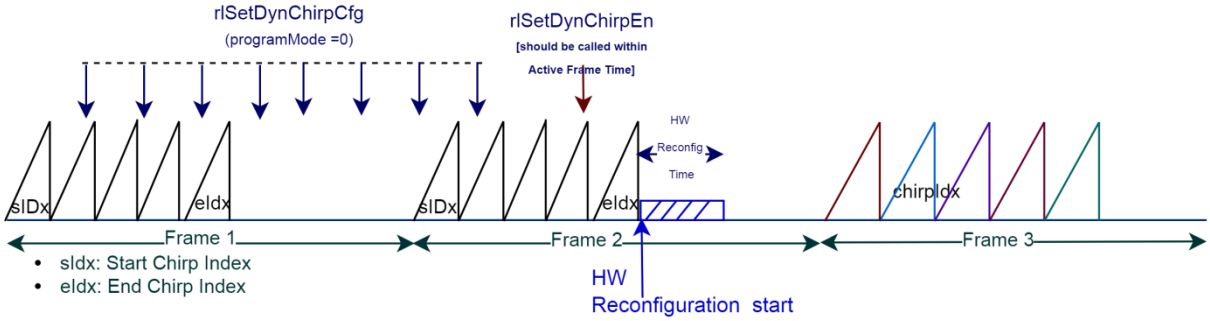


Figure 26. Program mode = 0

In program mode 1, new chirp configurations are applied to HW at the same time of rlSetDynChirpCfg API call.

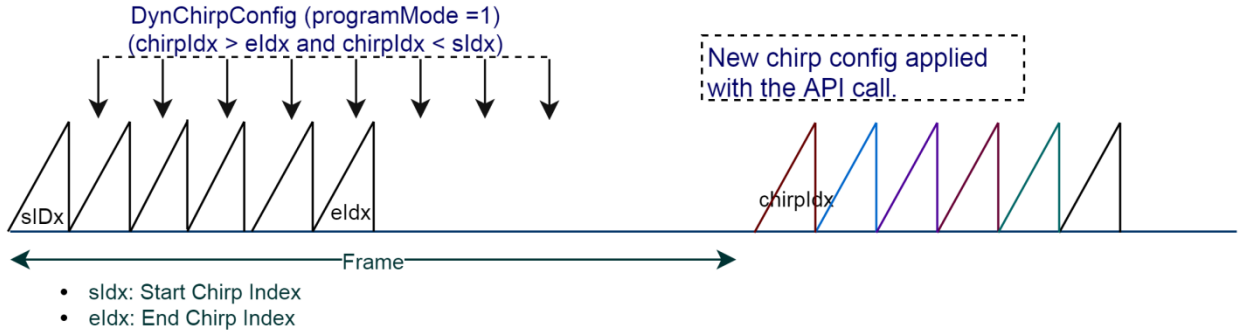


Figure 27. Program mode = 1. (Program the new configuration immediately)

NOTE:

1. User has to ensure that the chirps which are being reconfigured are not the ones which are being used by current running frame.
2. If used the legacy chirp configuration at the beginning, then dynamic chirp must have at least those chirp indices which were configured by legacy chirpConfig API
3. HW reconfiguration time (as shown in the figure below) is around 500 µs. User has to ensure that rlSetDynChirpEn API is issued at least 500 µs before the end of the ongoing frame active window (start of the frame idle time) to apply configurations for next frame onwards.

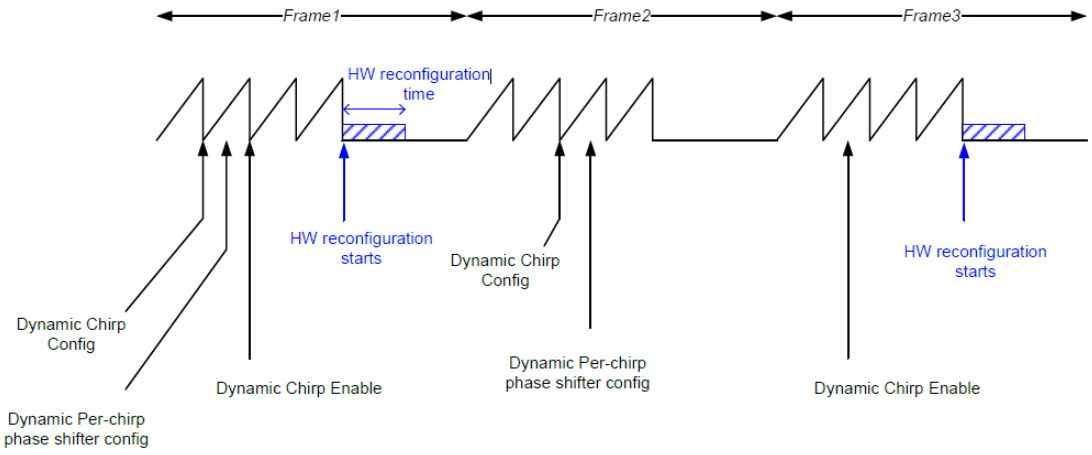


Figure 28. Dynamic Chirp use case timing diagram

11.4 Dynamic profile configuration

Profile configuration can be updated in the sensor during run time using the rlSetProfileConfig API. When this API is being called during framing, new parameters will be stored in the RAM which will be applied to hardware around frame end boundary. New profile parameters will be visible to the next frame.

TX calibration LUT needs to be updated if TX output power back off value is changed in the dynamic profile configuration. RadarSS updates the LUT for RX/TX at the beginning of next frame if requested in dynamic profile configuration. If application doesn't require to update LUT, then set [*pfCalLutUpdate=0x3*] in profile configuration API. It is recommended not to request for update LUT in profile configuration while updating other parameters in this API dynamically.

11.5 Dynamic per chirp phase shifter configuration

Phase shift to each TX can be configured in the mmWave sensor using either rlSetProfileConfig or rlRfSetPhaseShiftConfig API. Profile configuration provides an option to add phase shift for

specific profile index, so the chirps mapped to this profile index will have phase shift added to configured TX.

But if application requires adding phase shift to TX for each of chirp or group of chirps (independent of profile index), then use rlRfSetPhaseShiftConfig API. With this API, group of chirp can be selected to set phase shift value for TX [0-2] at the unit of 5.625°.

This API can be used to dynamically change the per-chirp phase shifter configuration (applicable only in certain devices) while frames are on-going. The configuration will be stored in software and the new configuration will be applied after receiving the rlSetDynChirpEn API.

If phase shifter is used by the per-chirp-phase-shifter, then RadarSS uses phase shifter value only from this API and not from Profile-phase-shifter. On the other hand, if the profile phase-shifter is set & not in the per-chirp, then RadarSS uses phase shifter value only from the profile configuration API.

11.6 Loopback configuration

mmWave Sensor provides internal RF loopback feature to monitor RX and TX analog chain. Currently there are three loopback that the device supports - PA, PS and IF.

Device provides debug loopback feature where all the functional chirps for all enabled profiles will be in loopback mode. In this case, only one type of loopback can be enabled at a time where application needs to reconfigure the device for another loopback type. If monitoring is enabled with the loopback APIs, then loopback will not work after monitoring is completed.

To use loopback with monitoring and along with functional chirp, application needs to use advanced frame with loopback burst configuration. In this case user can define specific sub-frame of advanced frame as loopback chirps.

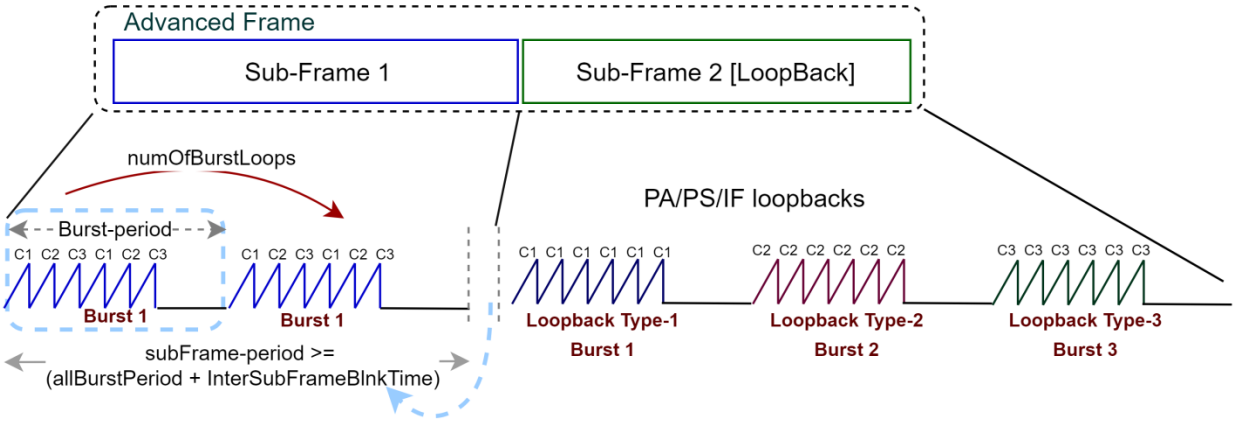


Figure 29. Loopback configuration

As per above figure, loopback feature can be combined with functional frame/burst of the device. In advanced frame configuration, one of sub-frame can be defined for loopback feature where different type of loopback needs to connect with individual burst of this sub-frame.

Features of Loopback sub-frame

- Max 16 different loopback configurations in 16 different burst of a given sub-frame.
- Advance frame configuration defines which sub-frame is being used for loopback.
- Only profile based phase shifter is supported in loopback configuration, not per-chirp phase shifter.
- No. of chirp in loopback burst must be set to '1' whereas same type of chirp can be looped within the burst.
- Device uses start Frequency and slope loopback-burst API overriding these parameters from profile-configuration.
- PS loopback is available for Tx0 and Tx1 only in the device.

rlSetAdvFrameConfig and rlSetLoopBckBurstCfg APIs are required to configure loopback chirp along with functional chirp in a set of sub-frames.

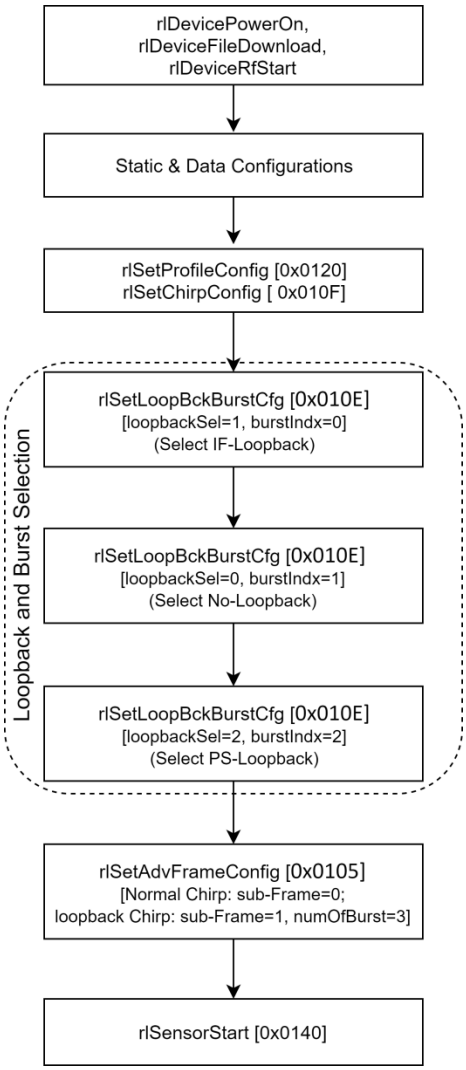


Figure 30. Loopback configuration – flow diagram

As another option, loopback can be used for debug purpose where all the functional chirps are in loopback mode. Application needs to invoke this API to enable one of the loopback at a time.

API	SB-ID	Feature
rlRfSetPALoopbackConfig	0x02CD	Enables/disables PA loopback for all enabled profiles. This is used to debug both the TX and RX chains are working correctly.
rlRfSetPSLoopbackConfig	0x02CE	Enables/disables PS (phase shifter) loopback for all enabled profiles. This is used to debug the TX (before the PA) and RX chains.
rlRfSetIFLoopbackConfig	0x02CF	This sub block enables/disables IF loopback for all enabled profiles. This is used to debug the RX IF chain.

	LNA	PA	PS	Mixer
No loop back	ON	ON	ON	ON
IF loop back	OFF	OFF	OFF	OFF
PS loop back	ON	OFF	ON	ON
PA loop back	ON	ON	ON	ON

NOTE:

1. Above APIs are recommended to use only for debug purpose. Functionality of device after switching from debug loopback mode is not guaranteed.

11.7 Factory Calibration

mmWave Sensor radar system on chip includes self-calibrations to mitigate process and temperature effects on analog performance. The calibrations include RF INIT (i.e., boot time) calibrations to mitigate manufacturing process variation effects, and Run Time calibrations to mitigate temperature effects. These calibrations mostly involve optimizing the RF register settings for TX, RX and LO based on the temperature read by the built-in temperature sensors.

This allows the devices to self-calibrate to account for manufacturing process variation. The ambient temperature (called factory calibration temperature) may be 25°C or the sensors typically expected in-field temperature. The factory calibrations should be done at the RF frequency range of interest for the sensor.

The following are recommended to be done in-field to ensure that all devices are restored to the same state as during Tier 1 factory calibration. This can be done using the following procedure. It has slight deviations from the normal RF INIT call and start up sequence.

Note that since selected calibrations are disabled during calibration restore, the calibration report API messages (AWR_AE_RF_INITCALIBSTATUS_SB) will also report those calibration statuses as 0. Note also that the mandatory calibrations (e.g. to keep the PLLs in lock) are anyway done without host control.

Please refer single chip DFP example for implementation of this feature.

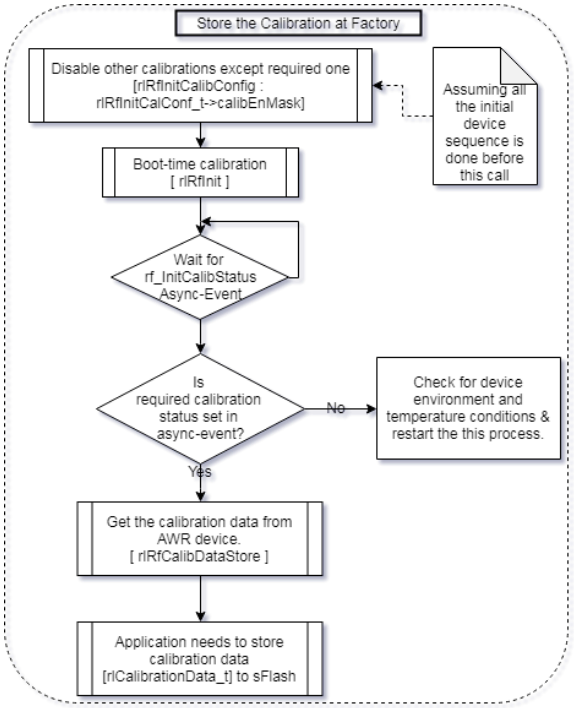


Figure 31. Calibration Store – flow diagram

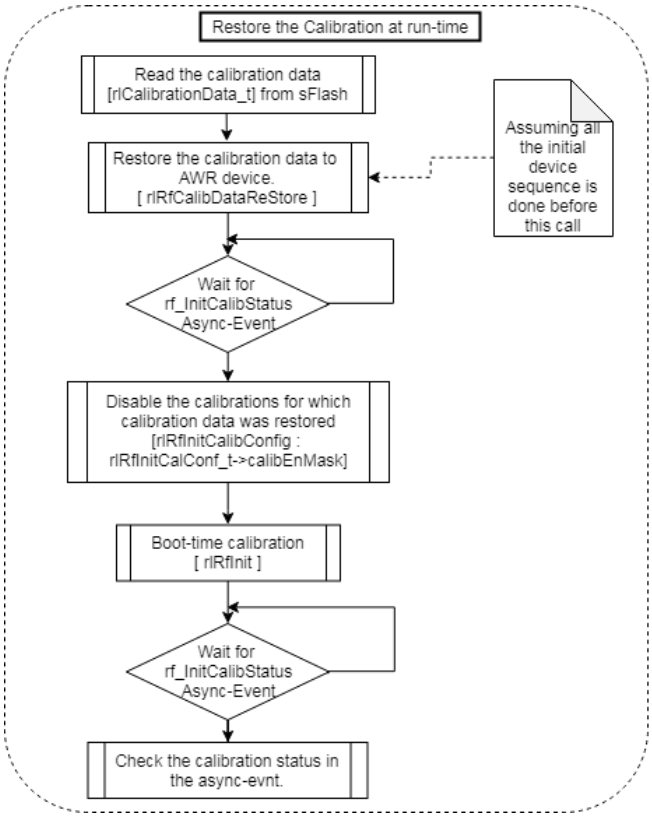


Figure 32. Calibration Restore – flow diagram

NOTE:

1. While capturing the calibration data, it should be done in the controlled environment.
2. No other Radar device or signal to cause interference to the DUT (device under test).
3. Device environment should be at room temperature.
4. Use rlRfPhShiftCalibDataStore and rlRfPhShiftCalibDataRestore APIs for Phase calibration store/restore.