Processor SDK - Radar

(v03.xx)

Development Guide

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards ought to be provided by the customer so as to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is neither responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

TABLE OF CONTENTS

1 Introduction

Processor SDK Radar is a multi-processor, multi-channel software development platform for TI family of ADAS SoCs. The software framework allows users to create different Radar application data-flows involving integration of FMCW radar transceiver, radar signal processing, and visualization on a display device.

This document explains details for the following Radar Data processing specific aspects:

1. Programming the AWR12XX device in order to capture ADC data.
2. Using File input output as a mechanism to input pre-recorded ADC data and save the outputs for analysis.
3. Developing Algorithm functions which can wrap around radar data processing algorithms without having to worry about Algorithm plugin buffer management.

For details regarding Low Latency Inter-processor communication mechanism based on Work Queues kindly refer VisionSDK_UserGuide_WorkQ.pdf in the Feature Specific user guides document section.

Processor SDK Radar is based upon links architecture used in Vision SDK and assumes that reader is familiar with basics of it.

The reader is strongly encouraged to go through VisionSDK_DevelopmentGuide.pdf for information regarding the following:

1. Usecase development (Section 2 of VisionSDK_DevelopmentGuide.pdf)
2. Link Development (Section 3 of VisionSDK_DevelopmentGuide.pdf)
3. Algorithm Link Development (Section 4 of VisionSDK_DevelopmentGuide.pdf). ***Note***: for Radar data processing algorithm links kindly refer Section 4 in this document.

2 Radar Development Overview

This section gives an overview to get started with Radar processing with Processor SDK Radar. In the following sections we look at some details regarding the Sensor configuration and Algorithm Function development.

2.1 Usecases to get started

The Processor SDK Radar provides the following usecases out of box to enable developers get started with the TDAxx + ARXX setup and also enable Algorithm developers start developing their algorithms.

1. **UC_radar_capture_only**: The first stage for anyone with the TDA3xx + AWR12XX setup is to ensure that the capture of the ADC data is working correctly. This usecase configures the AWR sensor and ISS to achieve this. Refer Section 3 for more details on the AWR configuration.
2. **UC_radar_read_fft_write**: For those without access to TDA3xx + AWR12XX setup but involved in Radar Processing Algorithm Development using either TDA2x or TDA3x EVM only setup, this usecase enables reading pre-recorded ADC data from SD Card or via Ethernet, perform FFT Processing and Save the output data back to SD Card or via Ethernet. Refer Section 4 for more details on Algorithm function creation for radar data processing. FFT processing on EVE is demonstrated with low latency IPC using work queue.
3. **UC_radar_capture_objectdetect_display**: This usecase forms an out of box demo for TDA3xx + AR12xx Booster Pack setup which allows displaying the output of the radar based object detection (FFT + Peak detection + Beam Forming) on an HDMI display. Radar processing on EVE is demonstrated with low latency IPC using work queue.
4. **UC_radar_capture_objectdetect_null**: This usecase forms an out of box demo for the ALPS board which does not have a display device connection. The output of the radar based object detection (FFT + Peak detection + Beam Forming) is sent out over Ethernet. Radar processing on EVE is demonstrated with low latency IPC using work queue.
5. **UC_multi_radar_capture_fft_display:** This usecase forms an out of box demo for the RVP + FPDlink based Radar setup which allows displaying the output of the FFT Heat map on HDMI Display for multiple Radar channels over FPDlink. Radar processing on EVE is demonstrated with low latency IPC using work queue.
6. **UC_radar_capture_process**: This usecase is a demonstration of how algorithm developers can develop their algorithm functions using the simple frame copy algorithm of this usecase.

2.2 Radar Specific Folders

The folders of specific interest during radar processing development are:

1. **Usecases:** vision_sdk\apps\src\rtos\radar\src\usecases
2. **Algorithm Functions:** vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns
3. **Algorithm Plugins:** vision_sdk\apps\src\rtos\radar\src\alg_plugins

4. **Common Infrastructure files for Radar:**
   vision_sdk\apps\src\rtos\radar\src\common
5. **Radar Specific Header files:** vision_sdk\apps\src\rtos\radar\include

3 AWR12XX Configuration and Radar Data Capture

Radar sensors like camera sensors are capable of capturing "frames" of data. In the case of the Radar sensor the "width" of the frame is defined by the number of ADC samples captured per Chirp times the number of receive antennas and the "Height" of the frame is defined by the number of chirps times the number of transmit antennas in a given frame.

3.1 STEP 0: Board specific Pin and Pad Configurations

The AWR12XX requires the following interfaces for control communication with the TDA device:

GPIO Input for Interrupt from AWR12XX to TDA

- BSP_AWR12XX_GPIO_INPUT_HOST_INTR

GPIO Output from TDA to AWR12XX for NRESET of the AWR12XX

- BSP_AWR12XX_GPIO_OUTPUT_NRESET

GPIO Output from TDA to AWR12XX for WARM RESET of the AWR12XX

- BSP_AWR12XX_GPIO_OUTPUT_WARM_RESET

GPIO Output from TDA to AWR12XX for Sense on Power Setting:

- BSP_AWR12XX_GPIO_OUTPUT_SOP_MODE_SEL_TDO
- BSP_AWR12XX_GPIO_OUTPUT_SOP_MODE_SEL_SYNCOUT
- BSP_AWR12XX_GPIO_OUTPUT_SOP_MODE_SEL_PMICOUT

Specific configurations with respect to the GPIO instance number and pin number can be configured using the Bsp_Ar12xxGpioInputParams and Bsp_Ar12xxGpioOutputParams.

Refer
ti_components\drivers\pdk_XX_XX_XX_XX\packages\ti\drv\vps\include\devices\bsp_ar12xx.h

Along with this the corresponding pad configurations also need to be updated. The PAD configuration for the default configuration happens in Bsp_boardTda3xxAr12xxRadarInit in the file ti_components\drivers\pdk_XX_XX_XX_XX\packages\ti\drv\vps\src\boards\src\bsp_boardTda3xx.c

This function needs to be updated if your board configuration differs from the default.

The SDK supports the following boards for AWR12XX communication out of box:

1. TDA3 + AWR12XX ALPS board. (tda3xx_alps_bios_radar)

2. TDA3 EVM + VAB + DIB + AWR Booster Pack (tda3xx_evm_bios_radar)
3. TDA3 RVP + AWR12XX (tda3xx_rvp_bios_radar)

The application developer can switch the configuration by using the corresponding make configuration.

3.2 STEP 1: Radar AWR1XX Configuration Parameters

AWR12XX devices support different configurations in order to be able to send out Radar data frames to the TDA device via CSI interface. The full list of configuration parameters can be found in <Install Directory>/ti_components/radar/ mmwave_dfp_XX.XX.XX.XX/docs/AWR1xx_Radar_Interface_Control.pdf

The BSP device implementation for AWR12XX abstracts the required programming sequences for configuring the AWR12XX device. (Code reference: <Install Directory>/ ti_components\drivers\pdk_XX_XX_XX_XX\packages\ti\drv\vps\src\devices\radar_ar12xx and <Install Directory>/ ti_components\drivers\pdk_XX_XX_XX_XX\packages\ti\drv\vps\include\devices\bsp_ar12xx.h

The first step to configuring the AWR12XX device is to create a configuration structure which defines what are the values corresponding to the AWR12XX configurations. This configuration contains the following information that needs to be populated:

- Static Channel Configuration Arguments
- ADC output data format
- AWR Low power mode
- Chirp Profile Configuration
- Frame (Normal/Advanced) Configuration
- Data format
- HSI Clocking and Lane Configuration

A sample configuration structure is defined in <Install Directory>/PROCESSOR_SDK_RADAR\vision_sdk\apps\src\rtos\radar\src\common\chains_common_ar12xx.c.

Using this as a reference the developer can define their own specific radar configuration using a variable with permanent storage as shown below:

```
Bsp_Ar12xxConfigObj   gAr1234CustomRadarConfig = {
. rfChanCfgArgs = { … }
. adcOutCfgArgs = { … }
…
}
```

Once this new configuration structure is populated, the developer can register this structure with the BSP AWR12XX driver using the API:

```
Int32 Bsp_ar12xxRegisterConfig(
    char            name[BSP_AWR12XX_MAX_SENSOR_NAME_LENGTH],
    Bsp_Ar12xxConfigObj *configObj);
```

During registration the user must provide a name to the new configuration which will be used as an identifier of the new custom configuration:

```
char        customName[BSP_AWR12XX_MAX_SENSOR_NAME_LENGTH]    =
"MY_CUSTOM_RADAR_CONFIG";
retVal = Bsp_ar12xxRegisterConfig(customName, & gAr1234CustomRadarConfig);
```

Note:

- You should register the configuration before calling Bsp_ar12xxInit and pass the same name of the configuration to the initialization parameters of Bsp_ar12xxInit.
- In case you have more than one user defined configuration, you should register the 1st configuration before Bsp_ar12xxInit. The other configurations can be registered after Bsp_ar12xxInit.
- Registration for a particular configuration needs to be done only once.
- In order to switch between different configurations one can use the following sequence:
    - Bsp_ar12xxRegisterConfig – 1st Configuration
    - Bsp_ar12xxInitParams_init (Refer Section 3.3 for more details)
    - Set the configuration name as the 1st configuration
    - Bsp_ar12xxInit (Refer Section 3.3 for more details)
    - Bsp_ar12xxConfigParams – Based on 1st Configuration
    - Start capture data and process with 1st configuration
    - Stop capture
    - Bsp_ar12xxRegisterConfig – 2nd Configuration
    - Bsp_ar12xxSwitchConfig – Switch the configuration to 2nd Configuration
    - Bsp_ar12xxConfigParams – Based on 2nd Configuration
    - Start capture data and process with 2nd configuration
    - Stop capture
    - Bsp_ar12xxUnRegisterConfig for 1st and 2nd configuration and close the usecase.
- The BSP maintains the pointer to the configuration parameters. If the same global structure variable is being modified with the 2nd Configuration then the registration and switch configuration can be avoided.

An example usage for registration of user defined configuration is given in PROCESSOR_SDK_RADAR\vision_sdk\apps\src\rtos\radar\src\common\chains_common_ar12xx.c

3.3 STEP 2: Radar Sensor Boot and Driver Configuration

Once the configuration structure is chosen and defined as per requirement, the next step is to initialize the AWR12XX sensor with these configurations. This is a one-time setup per usecase chain done at the beginning of the usecase configuration.

The below API call will setup the parameters corresponding to the radar configuration (excluding the actual sensor configuration).

```
/* Get the initial default parameters for the radar link configuration */
   Bsp_ar12xxInitParams_init (&pObj->radarInitParams);
```

Once this API is called the next step is to populate usecase specific details into this structure. The following parameters are of specific interest:

- radarConfigName : This name should correspond to the sensor configuration name registered earlier during STEP 1.
- edmaHandle : A handle for the Radar driver. The following statement can be used:

  pObj->radarInitParams.edmaHandle = Utils_dmaGetEdma3Hndl(UTILS_DMA_SYSTEM_EDMA_INST_ID);
- GPIO Instance and pin number being used. (used for reset and SOP control)
- McSPI Instance being used. (used for SPI communication)
- UART Instance being used. (used for RS232 based AWR firmware flashing)
- numRadarDevicesInCascade: Number of radar devices in cascade or FPDLink configuration.
- masterDevId: Identify which AWR12xx device is a master when multiple AWR12xx are connected. 0 can be populated in this field when the number of devices is 1.

For the full list of parameters of the radar sensor driver refer: <Install Directory>/ ti_components\drivers\pdk_XX_XX_XX_XX\packages\ti\drv\vps\include\devices\bsp _ar12xx.h

Since the control communication interface between TDA and AWR1XX is McSPI, ensure the following API from <Install Directory>\vision_sdk\links_fw\src\rtos\utils_common\src\utils_mcspi.c is called.

```
Utils_mcspiInit(CHAINS_SINGLE_RAD_SENSOR_MCSPI_INST0);
```

When the AWR1XX firmware is to be flashed via UART kindly ensure the following API from <Install Directory>\ vision_sdk\links_fw\src\rtos\utils_common\src\utils_uart.c is called.

```
Utils_uartCreateDevice(gAr12xx_initParams.uartDevInst, &uartAr12FlashDevObj);
```

Following this, the AWR12XX device can be initialized and booted. Note the below API will load set up the required initialization parameters.

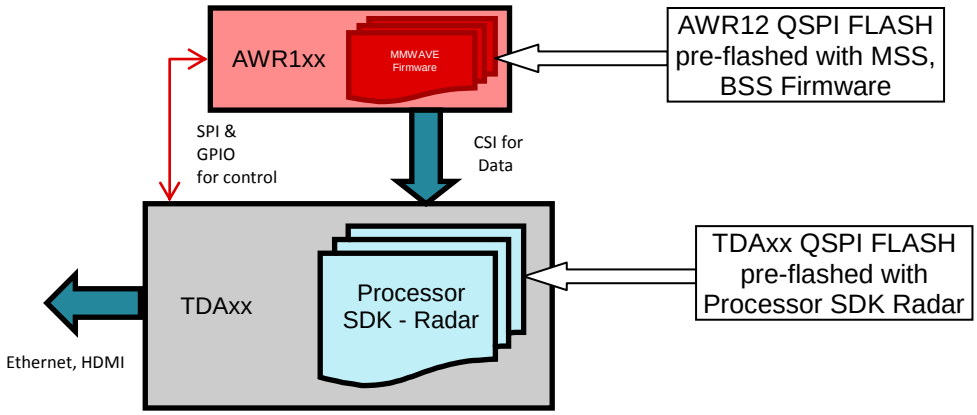
```
retVal = Bsp_ar12xxInit(&pObj->radarInitParams);
```

Once initialized the application should call

Int32 Bsp_ar12xxBoot(UInt32 downloadFirmware)

If downloadFirmware = 1, the Bsp_ar12xxBoot will download the MSS and BSS firmware to the AWR12XX via SPI and then lift the reset to wait for the AWR12XX to be booted successfully.

If downloadFirmware = 0, the API skips the MSS and BSS firmware download via SPI. It assumes the application has already flashed the AWR12XX firmware in the AWR12XX flash (highlighted in the figure below)



3.3.1 Flashing AWR1XX Firmware

The routines to flash the AWR1XX firmware are given in ChainsCommon_ar12xxFirmwareFlash. This utility is accessible through the usecase menu (ALPS board only) and can be run to either erase the flash completely or erase the flash and then program the MSS and BSS firmware binaries which are part of the mmwavedfp package.

The ChainsCommon_ar12xxFirmwareFlash calls the Bsp_ar12xxEraseFirmware or the Bsp_ar12xxFlashFirmware API to erase and program the flash based on the user input.

Note:

- For the ALPS and RVP board we always assume the AWR12XX flash memory is pre-flashed and the usecase does not try to load the MSS and BSS firmware via SPI.
- For the TDA3xx EVM + AWR12XX Booster Pack board the flash memory is kept erased and the AWR12XX firmware is always loaded via SPI.
- If there is a firmware image flashed in the AWR12XX flash, loading the MSS and BSS firmware again via SPI causes contention and can potentially lead to the firmware load via SPI to fail.
- The Flashing time can be between 3-4 minutes.
- For the RVP board the flashing is done via a PC tool and not via the TDA3x. For details regarding the steps for flashing please refer to the D3 Engineering user guide available with the TDA3x RVP board Radar Setup.

3.4 STEP 3: Program Sensor Configuration

Once the AWR12XX is booted and basic initialization is complete, the next step is to configure the parameters corresponding to STEP 1 to the AWR12XX.

This is achieved by calling the below API:

retVal = Bsp_ar12xxConfigParams(BSP_AWR12XX_CONFIG_ALL, 0U);

This API has flexibility to control what the developer wants to configure. For instance if the developer wanted to configure only the CHIRP and FRAME parameters, the parameters to the API call can be changed as below:

retVal = Bsp_ar12xxConfigParams(BSP_AWR12XX_CONFIG_CHIRP_PARAM | BSP_AWR12XX_CONFIG_FRAME_PARAM, 0U);

As described in STEP 1, the application developer can choose to have multiple configurations registered and can switch between the different configurations using Bsp_ar12xxSwitchConfig API and then calling the Bsp_ar12xxConfigParams.

3.5 STEP 4: Setting ISS Parameters

Once the configuration of the AWR12XX is complete on the TDA3 side the ISS parameters need to be set to receive the data correctly.

This configuration involves setting width and height of the frame, data format specification etc. This configuration can be referred to in the function

ChainsCommon_ar12xxSetIssCaptureParams in the file vision_sdk\apps\src\rtos\radar/src/common/chains_common_ar12xx.c

Typical configuration mismatches which lead to the capture to fail and the application developer should be careful about are:

1. Lane position configurations does not match the AWR lane configuration.
2. inCsi2DataFormat does not match dataFmtCfgArgs.adcBits. For the 16 bit ADC format, 2 RAW8 bit MIPI CSI format is used. For 12 bit and 14 bit, RAW12 and RAW14 is MIPI CSI format is used.

Note: The AR1xx sends the ADC samples out via CSI. When using a different radar sensor which does not use CSI but parallel interface, the application developer should use the VIP capture link and program the parameters based on the frame size and data format.

3.6 STEP 5 (Optional): Setting up the AWR1XX Test Source

The AWR1XX sensor supports generating the ADC data corresponding to up to two synthetic objects. This can be configured using the API:

Int32 Bsp_ar12xxEnableDummySource(UInt32 deviceId,

Bsp_Ar12xxTestSource *testSource);

This can enable developers to test their setup with a deterministic input and ensure data is received properly at the TDA.

The example implementation for enabling test source is given in vision_sdk\apps\src\rtos\radar/src/common/chains_common_ar12xx.c in the function ChainsCommon_ar12xxEnableTestSource.

Note: The Test source velocity values are not correct when the number of Transmit Antenna > 1.

3.7 STEP 6: Starting the Radar Sensor

Once the rest of the usecase chain is configured correctly the next step is to start the sensor to trigger AWR12XX to start sending ADC data using

retVal = Bsp_ar12xxStartRadar();

3.8 STEP 7: Checking if radar data capture is happening

In order to check if the data is received correctly, in the usecase menu type "p". The statistics that are shown on the UART console should indicate the FPS and ISS Capture Call back rates. An example is shown below.

Refer the example

usecase vision_sdk\apps\src\rtos\radar/src/usecases/radar_capture_only/

The following UART console output is shown for the Radar Sensor configured for 15 FPS operation.

```
[IPU1-0]    ### CPU [IPU1-0], LinkID [ 77],
[IPU1-0]
[IPU1-0]    [ ISSCAPTURE ] Link Statistics,
[IPU1-0]    ******************************
[IPU1-0]
[IPU1-0]    Elapsed time       = 15319 msec
[IPU1-0]
[IPU1-0]    Get Full Buf Cb    =  15.20 fps
[IPU1-0]    Put Empty Buf Cb   =  15.20 fps
[IPU1-0]    Driver/Notify Cb   =  15.14 fps
[IPU1-0]
[IPU1-0]    Input Statistics,
[IPU1-0]
[IPU1-0]    CH | In Recv | In Drop | In User Drop | In Process
[IPU1-0]       | FPS     | FPS     | FPS          | FPS
[IPU1-0]    ------------------------------------------------
[IPU1-0]     0 |  15.20      0. 0      0. 0          15.20
[IPU1-0]
[IPU1-0]    Output Statistics,
[IPU1-0]
[IPU1-0]    CH | Out | Out     | Out Drop | Out User Drop
[IPU1-0]       | ID  | FPS     | FPS      | FPS
[IPU1-0]    ------------------------------------------
[IPU1-0]     0 |  0     15.20     0. 0      0. 0
```

4 Algorithm Function Creation

This section describes the steps required for a developer to hook up their radar processing algorithm to the usecase data flow.

We use a simple frame copy algorithm to demonstrate this.

The nomenclature we use in this section is as below:

1. **Alg_plugin:** Algorithm Link which is used to build up the usecase chain
2. **Alg_function:** Wrapper functions over the actual algorithm implementation. Algorithm functions are called by the Algorithm Links whenever there is new data to be processed.
3. **Algorithm:** Actual data processing implementation. The Algorithm specific functions should be called by the Alg_functions.

***Note*: Algorithm functions can be used for all Radar processing links containing a single input queue and a single output queue.**

For custom Alg_plugin creation with more than one input or output queues refer Section 4 of the VisionSDK_DevelopmentGuide.pdf

4.1 Radar Process Alg_Plugin

Note: The developer needs to only write the Alg_function and Algorithm implementation. This section is mainly to enable the developer to get the overall picture of the code flow.

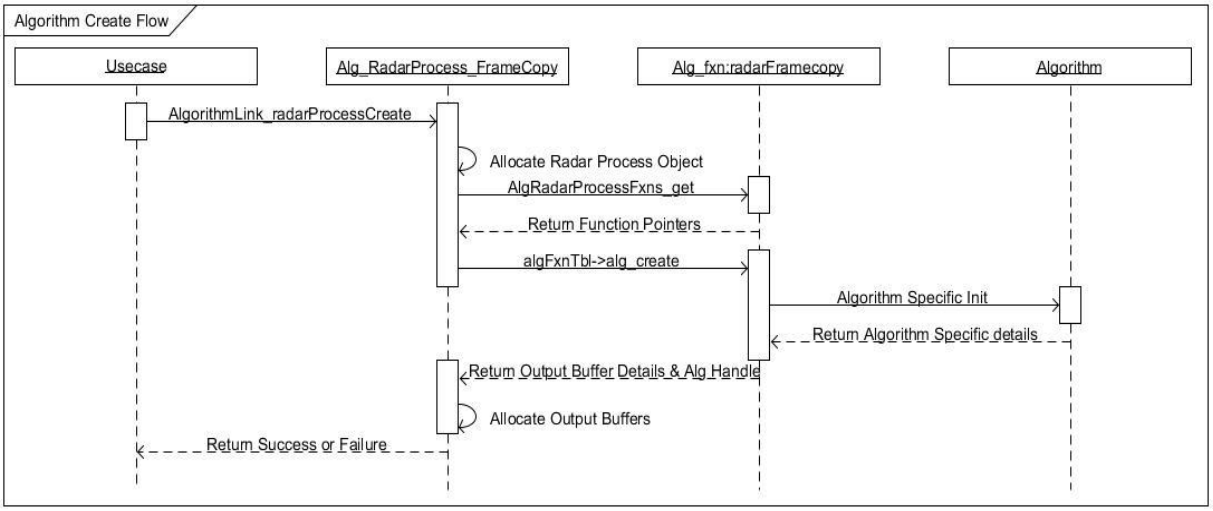
A common Radar Processing Algorithm Plugin has been created which would call the appropriate Alg_function function pointers which would invoke the processing specific functions.

We will look at the key steps/stages and code flow for the different components below:

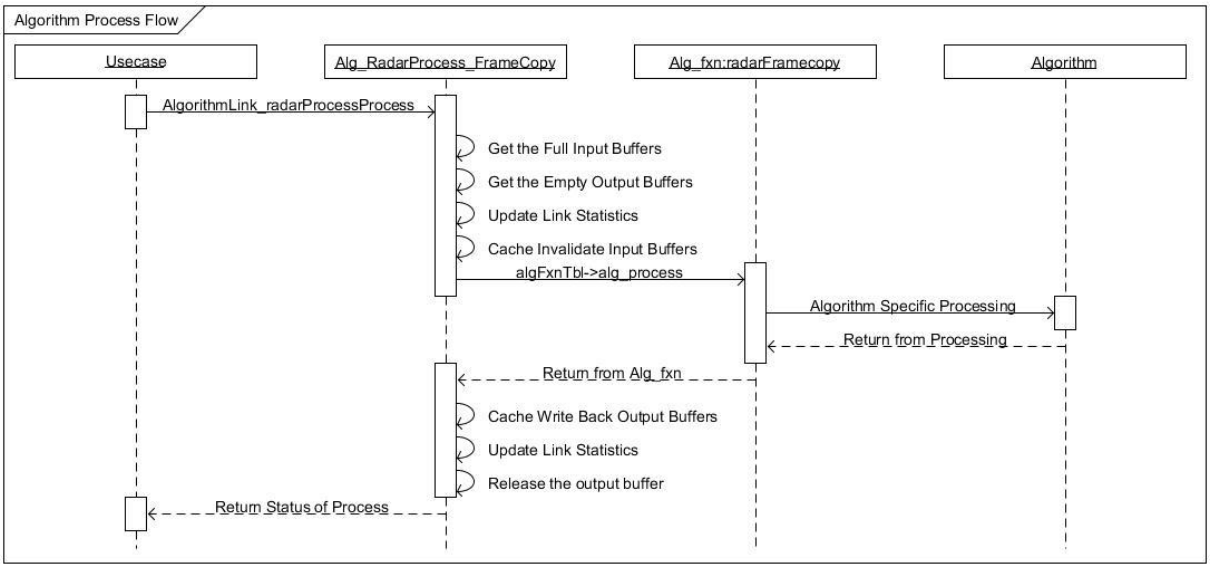
The code location for the Alg_plugin is

vision_sdk\apps\src\rtos\radar\src\alg_plugins\radarprocess\radarProcessLink_algPlugin.c

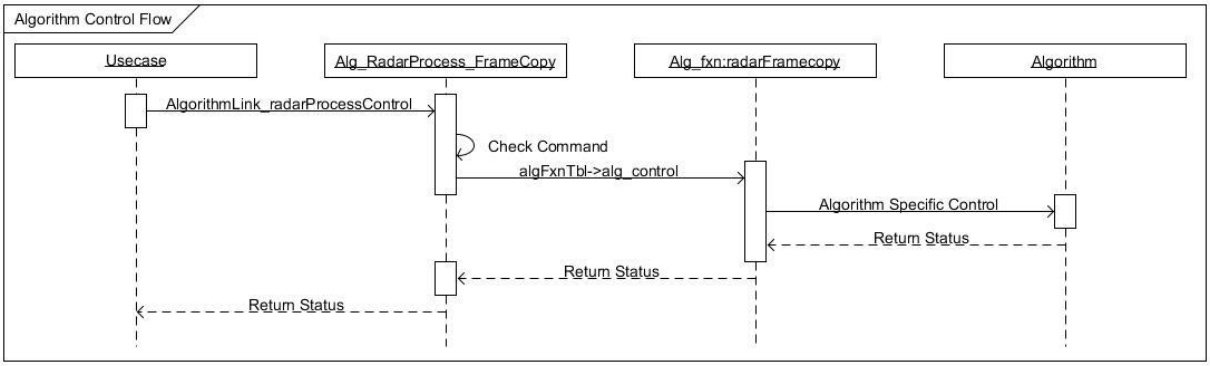
Algorithm Create: The code flow for the Algorithm Create is as below:



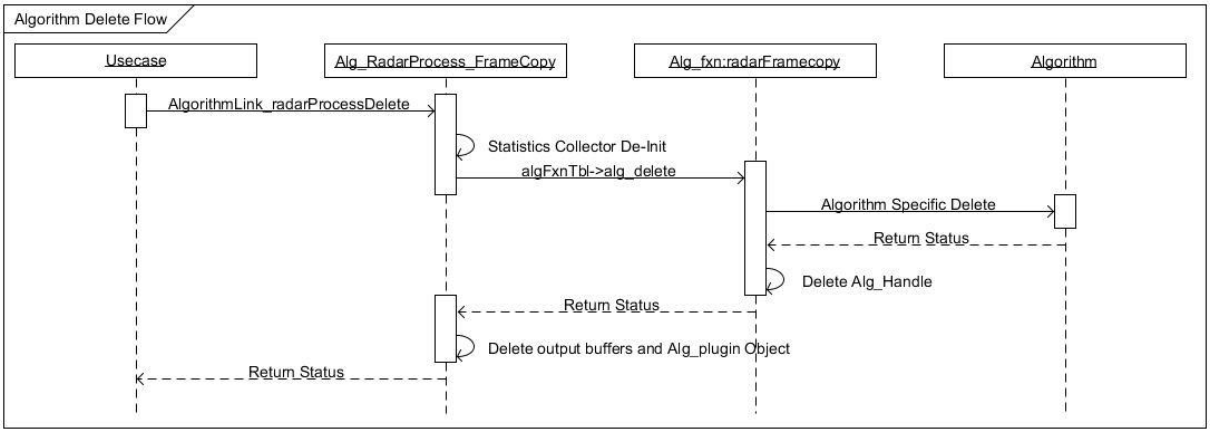
Algorithm Process: The code flow for the Algorithm process is as below:



Algorithm Control: The code flow for the Algorithm Control is as below:



Algorithm Delete: The code flow for the Algorithm Delete is as below:



4.2 Alg_Function Interface

The Alg_function interface is defined in vision_sdk\apps\src\rtos\radar\include\common\alg_functions.h

Each Algorithm stage is expected to have its own Alg_function interface. Although the Alg_plugin interface remains common, every instance of Alg_plugin is associated with one Alg_function and every instance of Alg_function is associated with one Algorithm.

When defining the Algorithm Function layer one must define the following functions:

4.2.1 Create Function

This function is called from the Alg_plugin during Algorithm creation during Usecase initialization. The main role of this function should be to:

- Call the Algorithm specific initialization functions
- Allocate memory for Algorithm Handle which the Alg_plugin will use to refer to it during frame processing and Algorithm deletion
- Indicate to the Algorithm Plugin, the nature of the output buffers the Algorithm will populate. Using this information the Alg_plugin will allocate output buffer space.
- Allocate any temporary or scratch buffer memory which is required by the Algorithm.

The input parameters to the Alg_function create are inputs to this interface.

The input parameters are defined in a structure as below:

```
typedef struct
{
   AlgorithmLink_RadarProcessCreateParams baseClassCreateParams;
   /**< Base Class Create Parameters. This should be the first element
    *  of this structure.
```

```
    */
   < Algorithm Function specific Create Params >
} AlgorithmFxn_<Algorithm>CreateParams;
```

As an example refer

vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns\radarframecopy\radarFrameCopy_if.h AlgorithmFxn_RadarFrameCopyCreateParams

For the create function example implementation refer:

vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns\radarframecopy\radarFrameCopy.c

4.2.2 Process Function

This Alg_function is called every time the Alg_plugin receives new buffers to process.

The Alg_function takes the following input:

- Algorithm Handle: Handle created during the Create Function is returned to the Process function.
- In_Bufs: Input System_buffers that need to be processed.
- Out_Bufs: Output Buffers which store the output of the Algorithm.

The input and output buffers can be Video Frame datatype or Meta data type buffers. The Alg_plugin takes care of the necessary cache_invalidate and cache_writebacks required for the input and output buffers.

Once the input and output buffers are obtained with in the Alg_function the developer can typecast the payload of the buffer to the required type and consume the buffers.

An example implementation of the process Alg_function is given in AlgorithmFxn_RadarFrameCopyProcess function in the file vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns\radarframecopy\radarFrameCopy.c

4.2.3 Control Function

This Alg_function is called whenever the usecase would like to change certain algorithm parameters at run time. The Alg_function can be invoked from the usecase by sending calling the following function:

```
AlgorithmFxn_RadarFrameCopyControlParams controlPrms;
controlPrms.baseClassControl.baseClassControl.size = sizeof(controlPrms);
controlPrms.baseClassControl.baseClassControl.controlCmd =
ALGORITHM_LINK_RADAR_PROCESS_CONTROL_CMD;

/* Note above the double "baseClassControl" is intentional as the Radar process Alg_plugin params is the base and this further contains a base class */
```

```
/* Here one can setup the rest of the Algo paramters */

/* Call the command function */
System_linkControl(
                pObj->Alg_RadarProcessLinkID,
                ALGORITHM_LINK_CMD_CONFIG,
                &controlPrms,
                sizeof(controlPrms),
                TRUE
                );
```

The control parameters are defined in a structure as below:

```
typedef struct
{
   AlgorithmLink_RadarProcessControlParams baseClassControl;
   /**< Base class control parameters */
   /** Algorithm Specific Parameters */
   UInt32 reserved;
   /**< Reserved parameter */
} AlgorithmFxn_<Algorithm>ControlParams;
```

As an example refer

vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns\radarframecopy\radarFrameCopy_if.h AlgorithmFxn_RadarFrameCopyControlParams

4.2.4 Delete Function

The Delete function has the following responsibilities:

- Call the Algorithm specific de-initialization functions
- De-Allocate memory for Algorithm Handle.
- De-Allocate any temporary or scratch buffer memory which is required by the Algorithm.

4.3 Registering Alg_Function Interface

The first step to creating the Alg_functions is to create the functions described in Section 4.2. Once these functions are created the next step is to create a table of the function pointers corresponding to these functions.

As an example see below:

```
static AlgRadarProcessFxns_FxnTable gAlgorithmFxn_RadarFrameCopy_fxns =
{
   &AlgorithmFxn_RadarFrameCopyCreate,
```

```
    &AlgorithmFxn_RadarFrameCopyProcess,
    &AlgorithmFxn_RadarFrameCopyControl,
    &AlgorithmFxn_RadarFrameCopyDelete
};
```

It is advisable to create a "Get functions" API as a part of the Alg_functions which returns a pointer to this table. Example:

AlgRadarProcessFxns_FxnTable * AlgorithmFxn_RadarFrameCopy_getFunctions(void)

The Alg_plugin calls a common function AlgRadarProcessFxns_get to be able to get this table of functions. This function is defined in vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns \common\alg_functions.c

Whenever a new Alg_function is created, the developer is expected to update the AlgRadarProcessFxns_get function.

The AlgRadarProcessFxns_get takes an input parameter of Alg_function name. The developer is free to add logic to this function to call the Alg_function Get API call based on the comparison of this name provided. This name is a Create time parameter to the Radar Process Alg_plugin.

Example:

```
AlgRadarProcessFxns_FxnTable *AlgRadarProcessFxns_get(char *name)
{
    AlgRadarProcessFxns_FxnTable *fxns = NULL;
    #if defined (ALG_FXN_radarframecopy)
    if (strcmp(name, "ti.radar.framecopy") == 0)
    {
        AlgRadarProcessFxns_FxnTable
             * AlgorithmFxn_RadarFrameCopy_getFunctions(void);
        fxns = AlgorithmFxn_RadarFrameCopy_getFunctions();
    }
    #endif
    return fxns;
}
```

4.4 Usecase Level Considerations

At the usecase level the Alg_function name and parameters need to be populated for the Alg_Plugin initialization and create.

In order to do this the Alg_function specific create time parameters need to be allocated space in the Usecase Application Object.

Example:
vision_sdk\apps\src\rtos\radar/src/usecases/radar_capture_process/chains_radarcaptureprocess.c

typedef struct {

```
…
   /* Algorithm Function Create Parameters */
   AlgorithmFxn_RadarFrameCopyCreateParams radarFrmCpyParams;

} Chains_radarcaptureprocessAppObj;
```

In the usecase, before calling the usecase create it is mandatory to populate the RadarProcess Algorithm Create params pointer with the address of the Alg_function create params.

Example:

```
chainsObj.ucObj.Alg_RadarProcess_framecopyPrm =
      (AlgorithmLink_RadarProcessCreateParams *)&chainsObj.radarFrmCpyParams;
```

In the usecase Set App Params, the alg_function parameter initialization needs to be performed. This will populate the Alg_function params with the required input parameters which will be used during Alg_function create.

Example:

```
Void chains_radarcaptureprocess_SetAppPrms(chains_radarcaptureprocessObj
*pUcObj, Void *appObj)
{
….
/* Algorithm Function Initialization */
   AlgorithmFxn_RadarFrameCopy_Init(&pObj->radarFrmCpyParams);
   pObj->radarFrmCpyParams.imgFrameWidth    = pObj->ar12xxCfg.csi2OutWidth;
   pObj->radarFrmCpyParams.imgFrameHeight = pObj-
>ar12xxCfg.csi2OutHeight;
   pObj->radarFrmCpyParams.outputBufferpitch = pObj->ar12xxCfg.csi2OutWidth;
   pObj->radarFrmCpyParams.dataFormat = SYSTEM_DF_BAYER_BGGR; ….
}
```

4.5 Radar Algorithms with WorkQ

For details regarding Low Latency Inter-processor communication mechanism based on Work Queues kindly refer VisionSDK_UserGuide_WorkQ.pdf in the Feature Specific user guides document section.

The Radar FFT algorithm has been implemented using work queues. Refer to the implementation in PROCESSOR_SDK_RADAR\ vision_sdk\apps\src\rtos\radar\src\alg_plugins\alg_fxns\radarfft2

The Host thread of the FFT processing runs on the DSP and the Work Thread runs on the EVE.

The Alg_function create function, process, control and delete functions run on the DSP.

The FFT Alg_function create function calls the work queue create using the function AlgorithmFxn_RadarFftDoWorkCreate to create the work thread on the EVE processor and sets up the parameters using AlgorithmFxn_RadarFftSetAlgProcessParams.

The FFT Alg_function process submits the work to EVE using AlgorithmFxn_RadarFftDoWorkProcess and waits for completion and checks the status of the operation.

The FFT Alg_function delete calls the AlgorithmFxn_RadarFftDoWorkDelete function in order to delete the work thread instance on EVE.

Note: When using the work queue, the usecase chain is defined using the Host thread as the link and within the host thread the communication to the work thread is implemented. Refer PROCESSOR_SDK_RADAR\ vision_sdk\apps\src\rtos\radar\src\usecases\radar_capture_fft_null usecase for usecase development with work queues.

5 Revision History

Version	Date	Revision History
0.10	29 Sept 2016	First Draft
0.20	30 Sept 2016	Addressed Review Comments
0.30	30 Jan 2017	Updates based on 2.12 Release.
0.40	5th July 2017	Updates based on 3.00 Release.
0.50	14th Oct 2017	Updates based on 3.01 Release
0.60	21th Dec 2017	Updates based on 3.02 Release

«««« § »»»»