Vision SDK

(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

Vision Software Development Kit (SDK) is a multi-processor, multi-channel software development platform for TI family of ADAS SoCs. The software framework allows users to create different ADAS application data flows involving video capture, video pre-processing, video analytics algorithms, and video display.

This document explains procedure for following

1. To develop a use case application using Vision SDK
2. To develop a new algorithm link

This document assumes that the reader is familiar with basics of links and chains architecture used in Vision SDK.

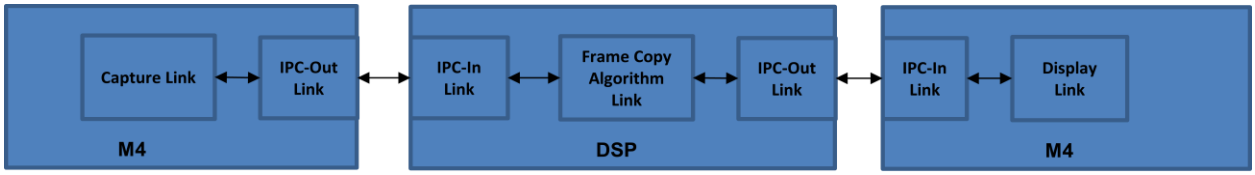
While describing the procedure to develop a use case, as an example, a simple use case is presented. This use case is available as an example in Vision SDK release package.

While describing the procedure to develop and integrate an algorithm link, as an example, two simple algorithms are presented. These algorithms are available as examples in Vision SDK release package.

2 Use Case Development

By use case, we mean the application which connects the links to form a chain and hence build a system.

2.1 Example use case



The example use case shown above consists of image capture (on IPU), followed by an algorithm (on DSP) and then followed by display (on IPU) of the image. For simplicity the example algorithm is chosen to be of a frame copy from one buffer to another.

For the purpose of capture and display the corresponding links on M4 are used.

For the frame copy algorithm, the corresponding algorithm link on DSP is chosen.

Since the use case spans across two different cores, IPC links are chosen for data exchange between cores. In general, a pair of IPC links (IPC-IN/IPC-OUT) is required to enable the data flow across a CPU boundary.

Main implementation of this use case is present in file

\vision_sdk\apps\src\rtos\usecases\vip_single_cam_frame_copy\chains_vipSingleCameraFrameCopy.c

Rest of this document explains various steps involved in building a use case.

2.2 Chain Creation

Use case in also referred to as a chain. Creation of a chain comprises of creation and connection of links. Following are the sub steps involved in creation of a chain.

2.2.1 Directory Structure

It is recommended to follow below directory structure for better clarity and modularity of the code base.

Create a directory, named as per the new use case (say " new_usecase"), under \vision_sdk\apps\src\rtos\usecases\

Create a new source file (.c), named as per the new use case, under \vision_sdk\apps\src\rtos\usecases\new_usecase\

Refer the chains_vipSingleCameraFrameCopy.c under "\vision_sdk\apps\src\rtos\usecases\vip_single_cam_frame_copy" as reference.

For example, in case of "vip_single_cam_frame_copy", use case is implemented in function Chains_vipSingleCameraFrameCopy() in file chains_vipSingleCameraFrameCopy.c

2.2.2 Generating the use-case using the use-case gen tool

Refer to docs\VisionSDK_UsecaseGen_Overview.pdf and docs\VisionSDK_UsecaseGen_UserGuide.pdf for using the use-case generation tool.

Basic steps are,

1. Write the use-case in a .txt file,
   example, chains_vipSingleCameraFrameCopy.txt
2. Run the tool to generate the use-case file as shown in below example,
   > \vision_sdk\build\rtos\scripts\vsdk_win32.exe –img –file chains_vipSingleCameraFrameCopy.txt
3. This will generate the files chains_vipSingleCameraFrameCopy_priv.c, chains_vipSingleCameraFrameCopy_priv.h
4. Write the remaing portion of the use-case, ex, chains_vipSingleCameraFrameCopy.c
5. Compile and run the use-case

2.2.3 Setting of Link ID

NOTE: Setting of link ID code is generated by the use-case gen tool, this section can be referred for informational purposes only.

Each link in the chain has a unique link ID. First step in chain creation is to assign link Ids for all the links present in the use case.

For example use case, setting of link Id is handled in function chains_vipSingleCameraFrameCopy_SetLinkId () [chains_vipSingleCameraFrameCopy_priv.c]

For links which can execute on a particular CPU core only, the Link Id value is just defined by a macro. Ex: For capture link which is present only on M4 core, the link Id is as follows,

pObj->captureLinkId = SYSTEM_LINK_ID_CAPTURE;

Refer file, \vision_sdk\links_fw\include\link_api\system_linkId.h for the available links

For common links, which can execute on any CPU core, the link Id value depends on the processor core on which the link will run and the functionality of the link.

Ex: For IPC In Link which runs on a given CPU, the link Id is obtained as a function of both the proc Id and the link functionality as follows -

pObj->ipcInLink_CPU_Id = SYSTEM_MAKE_LINK_ID(pObj->algLinkProcId, SYSTEM_LINK_ID_IPC_IN_0);

2.2.4 Default Setting of Link parameters

NOTE: Default Setting of Link parameters code is generated by the use-case gen tool, this section can be referred for informational purposes only.

Typically, each link will have certain default values for creation time parameters. Setting of these default values can be done by calling the parameter init functions provided by Vision SDK (Ex: CaptureLink_CreateParams_Init() function performs default parameter initialization for capture link)

Function calls to all the default parameter value initialization can be seen in function chains_vipSingleCameraFrameCopy_ResetLinkPrms () [chains_vipSingleCameraFrameCopy_priv.c]

2.2.5 Setting of Link parameters

The default Parameter values can be modified here as needed by the use case.

This is done in function
chains_vipSingleCameraFrameCopy_SetAppPrms()[chains_vipSingleCameraFrameCopy.c]

This step need to be done for each link individually, if usecase demands different set of values compared with the default values.

The use-case gen tool also generates code to set parameters for some links, so these need not be done by the user. The links for which the parameters are generated by the use-case gen tool are,

- DUP
- MERGE

- IPC OUT
- IPC IN

This is done in function chains_vipSingleCameraFrameCopy_SetPrms() [chains_vipSingleCameraFrameCopy_priv.c]

2.2.6 Connecting Links

NOTE: Connecting links code is generated by the use-case gen tool, this section can be referred for informational purposes only.

This step configures the connection between several links in the desired manner. This is nothing but creating the desired chain.

For example use case, connecting links is done in function
chains_vipSingleCameraFrameCopy_ConnectLinks () [chains_vipSingleCameraFrameCopy_priv.c]

Connecting a link to its previous and next link is by programming values of input and output queue parameters, which are part of Link create parameters.

In example use case, connecting algorithm link to its previous link (IPC in link) and next link (IPC out link) is done as follows:

```
pObj->algLinkCreatePrms.inQueParams.prevLinkId = pObj->ipcInLink_CPU_Id;
pObj->algLinkCreatePrms.inQueParams.prevLinkQueId = 0;
pObj->algLinkCreatePrms.outQueParams.nextLink = pObj->ipcOutLink_CPU_Id;
```

2.2.7 Link creation

NOTE: Link creation code is generated by the use-case gen tool, this section can be referred for informational purposes only.

Links are created using System_linkCreate() API, which takes in link Id, create time parameters and size of create time params structure as arguments. Resources and memory allocation is done in this step.

For example use case, link creation calls are present in function
chains_vipSingleCameraFrameCopy_Create() [chains_vipSingleCameraFrameCopy_priv.c]

Ex: Creation of capture link is done as follows

```
status = System_linkCreate(pObj->captureLinkId, &pObj->capturePrm, sizeof(pObj->capturePrm));
```

Note:

1. Typically, every link takes a few input side create time parameters from the previous link. For example, there would be certain create time parameters, like input color format, input video frame data format, input frame resolution, input pitch etc., which are actually controlled by the previous link. These parameters are not exposed for user configuration. These common parameters of the links, previous link (output) and current link (input) need to be exactly the same. Programming these on both links explicitly could be errors prone and redundant. In order to avoid this, these parameters are programmed for the output side of first (previous) link in the chain and subsequent (next) link in the chain pick the parameters by querying their previous link. This querying of the previous link happens during create phase. For the query to be successful, previous link needs to be successfully created before creating the current link. **Hence creation of the links should be done from source (first) link to destination (last) link of chain in the order, the same direction as data/frames would flow. This link create order is a hard requirement.**
2. Memory and other resources needed for operation of a link will be requested by the links themselves to the resource manager present in Vision SDK. Use case need not be concerned about this request and grant. However, some parameters which control these requests will be link create time parameters and they need to be set up appropriately by the use case at creation phase. For Ex: Number of output buffers for a given link will typically be a create time parameter, which needs to be set up by use case.

With this step creation of a chain is complete

2.3 Starting Execution of Chain

NOTE: Starting Execution of Chain code is generated by the use-case gen tool, this section can be referred for informational purposes only.

To start execution of chain, all the links in the chain need to be started.

For example use case, starting the chain is implemented in the function
chains_vipSingleCameraFrameCopy_Start() [chains_vipSingleCameraFrameCopy_priv.c]

Execution of a link is started by calling the API System_linkStart() which takes in link Id as the argument.

Ex: Staring display link is accomplished as follows,

```
status = System_linkStart(pObj->displayLinkId);
```

With this step, the entire chain would begin execution.

Note:

Typically, you can start the destination link before the source link. The link start order is exactly reverse of the link create order. Again, this is not a hard requirement, but good to follow this guideline

2.4 Stopping and Deletion of Chain

NOTE: Stopping and Deletion of Chain code is generated by the use-case gen tool, this section can be referred for informational purposes only.

Once the chain has started execution, it can be stopped either based on a user input Or based on a timer.

For example use case, stopping and deleting chain is implemented in functions

chains_vipSingleCameraFrameCopy_Stop() [chains_vipSingleCameraFrameCopy_priv.c]

chains_vipSingleCameraFrameCopy_Delete() [chains_vipSingleCameraFrameCopy_priv.c]

Stopping and deleting a chain essentially involves stopping and deleting each link present in the chain. They are accomplished using functions System_linkStop() and System_linkDelete(), which take in link Id as the argument.

Ex: Stopping and deleting algorithm link is done as follows

```
status = System_linkStop(pObj->algLinkId);
status = System_linkDelete(pObj->algLinkId);
```

Note:

1. Stopping of links should follow the same order as create order. So stopping should happen from first to last link. Again, this is not a hard requirement, but good to follow this guideline
2. Deletion can be done in any order.
3. Memory and other resources used by the link for its operation will be released by the links automatically during delete phase.

2.5 Build the new usecase file

Create a new make file (SRC_FILES.MK), under

\vision_sdk\apps\src\rtos\new_usecase\

This make file will include the new use case file into the build flow. Without this the newly created use case file will not get complied and linked

Refer the SRC_FILES.MK file under "vip_single_cam_frame_copy" for details/help

The "Makefile" of \vision_sdk\apps\ also need to be modified, by adding include path for the new usecase

For example use case, “vip_single_cam_frame_copy”, the below line is added,

include $(MODULE_SRC_BASE_PATH)/rtos/usecases/vip_single_cam_frame_copy /SRC_FILES.MK

3 Link Development

3.1 What is a Link

VISION SDK is be based on the "Links and Chains" framework. A link is the basic processing block in a video data flow. A link consists of a OS thread coupled with a message box, implemented using OS semaphores. The message box associated with a link allows user application as well as other links to talk to that link. The link implements a specific interface which allows other links to directly exchange video frames and/or bit streams with the link.

For more information on Links, please refer to the document located here.

Link development involves in creating a few set of files and adding these files into the make file. Although it's not mandatory to develop links in this way, it is quite recommended so as to keep the interfaces clean and easy to understand code flow.

3.2 Link Files

Any link is spread across multiple files. It is important for a developer to have an understanding of what each files contains. This greatly helps in developing links as well as understanding links developed by others. The following table gives a brief of each file

S.No	File Name	Location	Description	Comments
1	<link_name>Link.h (ex: captureLink.h)	\vision_sdk\links_fw\include\link_api	Public Interface File for Link	This file consists of all user/application level configuration related to the link
2	<link_name>Link_priv.h (ex: captureLink_priv.h)	\vision_sdk\links_fw\src\rtos\links_<prcoc_name>\<link_name>\	Private Interface file for link	This file consists of all macros/includes/function API specific to the link. User/application need not care about this file
3	<link_name>_tsk.c (ex: captureLink_tsk.c)	\vision_sdk\links_fw\src\rtos\links_<prcoc_name>\<link_name>\	Task file which waits for commands to be	This file calls driver specific API to achieve a particular

			received from application or other links	task
4	<link_name>_drv.c (ex: captureLink_drv.c)	\vision_sdk\links_fw\src\rtos\links_<prcoc_name>\<link_name>\	Driver specific API are implemented in this file	

3.3 Creating public header file

Every link can be configured through the application/user. Generally a set of link create parameters are encapsulated in an object and these can be passed when creating link.

It is up to the developer to provide the necessary configurability to the link. A developer can expose all configurable parameters of the link in an object and pass it while creating link. Also, input/output queue information is also encapsulated in this object.

For example, In Dup Link, the number of output queues is a configurable parameter and is passed through application.

3.4 Creating private header file

The private header file for the link consists of link specific macros and data structures. Developer can add data structures for the link, generally termed as <link_name>Obj.

This object consists of all information of the link. Some standard contents of this object are,

S.No	Name	Description
1	Create arguments	Needed by link to refer to the previous and next link information
2	Link Info	Current link information
3	Previous Link Info	Previous link Information
4	System Buffers	This is optional. If the link needs to exchange data with other links, these are used. Also payload for these buffers should also be created.

3.5 Creating task file

Every link has an associated task file. The task file acts like a state machine. This task file always runs in an infinite loop waiting for commands from either top level application or from any other link, generally from a previous link. The task file also

has the "init" call which actually registers this link in the framework. This init has to be called at the start up sequence of the framework and this has to be called on all the cores on which this link will be used.

For Example, Dup Link is a generic link which can be used on any core where multiple duplicate output data paths of video is needed from a single input data path. Such a link is needed on any processing core, so it's "init" call has to be called on all needed cores.

Every link has a specific functionality and hence commands to the link may vary from link to link. The following lists the standard commands that most of the links support,

1. SYSTEM_CMD_CREATE :

   This must be the first command to the link. A link upon receiving this command initializes all link related data structures. All profiling variables like frames received, frames processed, latency are initialized. Apart from these, if the link interacts with the hardware using driver calls, then all driver related initializations are also done in this call. Generally, this is done through the standard FVID2 Interface. Please refer to the capture link for more information.

2. SYSTEM_CMD_START :

   This command is optional. Not every link needs an explicit start command. Generally drivers need explicit start command to start the underlying hardware. In such cases, using the standard the FVID2 interface, start command is issued to the driver.

3. SYSTEM_CMD_NEW_DATA :

   Application or previous link posts this command whenever new data is available to this link. Upon receiving this command, the link will process data and puts in the output queue of next link and signals the next link about the availability of data. The processing of data is very much specific to the link and greatly differs from link to link.

4. SYSTEM_CMD_STOP :

   Same as start command, this command is also optional.

5. SYSTEM_CMD_DELETE :

   Delete command must de-initialize all the data structures that have been initialized in the create call. Any memory allocations done have to be freed up in this call. After this call, the task associated with this link also gets deleted. Thus, no other commands to the link can be further issued.

3.6 Creating driver file

Generally most links interact with the hardware to achieve a particular task. For example, VPE link interacts with the hardware to resize an image, to convert de-interlaced video to progressive video. All these functions are implemented by the BSP drivers and can be accessed through the standard FVID2 interface. These functions are called from the task file. See captureLink_drv.c file for sample driver file implementation.

3.7 Initializing link

As stated earlier in section 2.4, all links must be registered in the framework before they can be used. Some connector links like Dup Link, Null Link, Sync Link etc can be run on any core, while links like capture link, VPE link etc can be run on a specific core (here, they can be run on only IPU1-0). For a link to be able to be created on a core,

1. It should be registered with the framework. The following sample code shows how to register DUP link(s) with the framework.

```
for(dupId = 0; dupId < DUP_LINK_OBJ_MAX; dupId++)
{
        pObj = &gDupLink_obj[dupId];

        memset(pObj, 0, sizeof(*pObj));

        pObj->tskId = SYSTEM_MAKE_LINK_ID(procId,
                                          SYSTEM_LINK_ID_DUP_0 + dupId);

        linkObj.pTsk = &pObj->tsk;
        linkObj.linkGetFullBuffers = DupLink_getFullBuffers;
        linkObj.linkPutEmptyBuffers = DupLink_putEmptyBuffers;
        linkObj.getLinkInfo = DupLink_getLinkInfo;

        System_registerLink(pObj->tskId, &linkObj);

        sprintf(tskName, "DUP%u", (unsigned int)dupId);

        /*
         * Create link task, task remains in IDLE state.
         * DisplayLink_tskMain is called when a message command is received.
         */
        status = Utils_tskCreate(&pObj->tsk,
                                 DupLink_tskMain,
                                 DUP_LINK_TSK_PRI,
                                 gDupLink_tskStack[dupId],
                                 DUP_LINK_TSK_STACK_SIZE, pObj, tskName);

        UTILS_assert(status == SYSTEM_LINK_STATUS_SOK);
}
```

2. The above code should be called from the startup sequence of the core on which it is intended to run. The 'DupLink_init()' is called from 'System_initLinks()' and 'DupLink_deInit()' is called from 'System_deInitLinks()'. The System_initLinks() and System_deInitLinks() are defined in the \vision_sdk\links_fw\src\rtos\links_common\system\system_initDeinitLinks.c file which is common for all the cores.

   The 'DupLink_init()' and 'DupLink_deInit()' are called under '`#ifdef links_common_dup`', and this compile flag is defined only when the DUP link code is compiled for the core which is calling the System_initLinks() and System_deInitLinks() functions.

4 Algorithm Link Development

In order to integrate an algorithm into Vision SDK framework, it is required to develop an algorithm link. Once algorithm link is developed, it can be used like any other link in the use case. This chapter describes the procedure to develop an algorithm link.

4.1 Algorithm Link Design Overview

To enable easy and fast development of algorithm links, the link is designed to consist of two portions - Skeleton and plug-in functions

1. Skeletal part of algorithm link:
    a. Comprises of portions of algorithm link implementation, which are common across algorithms
    b. Takes care of generic aspects of link implementation like link creation, link state machine, communication with other links etc.
    c. Provided by TI as part of Vision SDK framework
2. Plug-In Functions:
    a. Comprises of functions which cater to algorithm dependent functionality
    b. Needs to be written, specific to the algorithm being integrated

Skeletal code implementation and communication API is kept same independent of the processing core (EVE/DSP/A15/M4). Skeletal code shall call the plug-in functions based on the state of the algorithm link. Plug-in functions have the implementation to create and use the actual algorithm functions (Provided by the algorithm provider). Plug-In functions can interact with algorithm functions via iVision or any other interface.

With above design, development of a new algorithm link essentially means development of the plug-in functions and their integration.

4.2 Algorithm Link Skeleton

This section provides information about file organization of algorithm link skeletal portion, which is provided by TI. This understanding is necessary for algorithm plug in development

S. No	File Name	Location	Description	Comments
1	algorithmLink.h	\vision_sdk\links_fw\include\link_api	Interface File for Algorithm Link	Developer needs to add algorithm Id for a specific core in this file

2	algorithmLink_algPluginSupport.h	\vision_sdk\links_fw\include\link_api	Interface file for plug-in functions to interact with Algorithm Link	Developer need not modify this file at all
3	algorithmLink_algPluginSupport.c	\vision_sdk\links_fw\src\rtos\links_common\algorithm	Implementation file of the above	Developer need not modify this file at all
4	algorithmLink_cfg.h	\vision_sdk\links_fw\src\rtos\links_common\algorithm	Private API/data structures for Algorithm Link	Developer may modify this to increase/decrease number of algorithm link instances on a core.
5	algorithmLink_cfg.c	\vision_sdk\links_fw\src\rtos\links_common\algorithm	Configuration for Algorithm Link	
6	algorithmLink_priv.h	\vision_sdk\links_fw\src\rtos\links_common\algorithm	Algorithm Link private header file	User need not modify this file at all
7	algorithmLink_tsk.c	\vision_sdk\links_fw\src\rtos\links_common\algorithm	Algorithm Link implementation file	User need not modify this file at all
8	App_init_<CORE>	\vision_sdk\apps\src\common\app_init	Application Init code	Add algorithm plug-in init call in these files.

4.3 Algorithm Link Plug-In Development

List of plug in functions which are typically needed for an algorithm are as follows:

AlgorithmLink_AlgPluginCreate	Plug in function which will perform algorithm instance creation
AlgorithmLink_AlgPluginProcess	Plug in function which will process new data. Internally it will call the process function of the algorithm
AlgorithmLink_AlgPluginControl	Plug in function which will perform Control (Configuration) of the algorithm. Internally it will call the control function of the algorithm.
AlgorithmLink_AlgPluginStop	Plug in function which will perform all functionality which needs to be done at the end of algorithm. Example: If any buffers are locked inside the algorithm, they can be flushed in this function.
AlgorithmLink_AlgPluginDelete	Plug in function which will perform algorithm instance deletion

Rest of this section describes several aspects of algorithm plug-in development

4.3.1 Algorithm ID

Each algorithm is identified by a unique id which we call as Algorithm Id. An algorithm can run on any of the cores depending on the requirements of the algorithm. This is specific to the algorithm and Vision SDK does not restrict algorithm to run on a specific core.

Developer need to add a new algorithm Id to the specific core on which the algorithm is meant to run. This has to added in the file algorithmLink.h present at the following location,

\vision_sdk\links_fw\include\link_api

For example, Color To Gray algorithm is meant to run on DSP, so we add an enum in AlgorithmLink_DspAlgorithmId.

4.3.2 Input and Output Queues

Like any other link, algorithm link can have multiple input and output queues. The number of queues and their properties would depend on the nature of the algorithm.

Plug-In function AlgorithmLink_AlgPluginCreate() need to convey this information to the skeleton via AlgorithmLink_queueInfoInit() API, by populating the AlgorithmLink_InputQueueInfo and AlgorithmLink_OutputQueueInfo

4.3.3 Input and Output Buffers

Like any other link, input buffers for an algorithm link shall come in from previous link, which provides input for algorithm link. Output buffers need to be owned by the algorithm link (Except for the case of In-place computations). Hence AlgorithmLink_AlgPluginCreate() needs to create the output buffers needed for the algorithm.

In some algorithms, it is possible that input and output buffers will have to be locked inside algorithm for more than one frame duration. In such cases the

AlgorithmLink_AlgPluginProcess() plug in function needs to have a suitable API to communicate locking and freeing of buffers with the Algorithm.

Following APIs shall be used by the Process plug-in function to exchange input and output buffers with skeleton / rest of the system:

System_getLinksFullBuffers()	To get input buffers from previous link
AlgorithmLink_getEmptyOutputBuffer()	To get free output buffers from previous link
AlgorithmLink_putFullOutputBuffer()	To pass on the output buffer, which is populated by the algorithm, onto next link
AlgorithmLink_releaseInputBuffer()	To release / free up Input buffer. This needs to be done when algorithm no longer needs this input buffer. Skeletal code shall internally pass on this buffer to previous link.
AlgorithmLink_releaseOutputBuffer()	To release / free up output buffer. This needs to be done when algorithm no longer needs this output buffer.

There are two modes in which an algorithm link can operate based on how the input buffers are handled in the actual algorithm

1. Non In Place mode: In this case output buffers are different from input buffers. And Input buffers are not modified by the algorithm
2. In Place mode: In this case Input buffers are modified and hence same buffer will serve as output buffer to be passed on to next link.

Vision SDK release package has example algorithms for both modes.

Algorithm Mode	Algorithm Name	Description
Non In Place Mode	Frame Copy	Input frames are duplicated and forwarded to the next link. Input frames are not modified.
In Place Mode	Color To Gray	Input Frames are modified. The chroma component of the frame is masked out to make the frame look gray.

Algorithm link has functions to handle both mode of operations. During the create time of the plug-in, user has to set the mode of operation of the input and output

queues. The following two sub sections explains how to set the modes for input and output queues for both modes.

4.3.3.1 Non In Place Mode

Let us consider Frame Copy Algorithm for Non In Place mode. In Frame Copy algorithm input buffers are not modified and output buffers are used. So we need the following mechanisms to manage input and output queues.

1. Managing Input queues

   At the create time of the plug-in, we need to specify the mode of operation as ALGORITHM_LINK_QUEUEMODE_NOTINPLACE. The code snippet is shown below,

```
1. AlgorithmLink_InputQueueInfo  inputQInfo;
2. pInputQInfo.qMode  = ALGORITHM_LINK_QUEUEMODE_NOTINPLACE;
```

   Since Input buffers are not modified, these can be released to the previous link by calling AlgorithmLink_releaseInputBuffer.

2. Managing Output queues

   At the create time of the plug-in, we need to specify the mode of operation as ALGORITHM_LINK_QUEUEMODE_NOTINPLACE. Along with the mode, we also need to specify the queue information. Other members of this structure are don't care for Non In Place mode. The code snippet is given below,

```
1.  AlgorithmLink_OutputQueueInfo  outputQInfo;
2.  outputQInfo.qMode = ALGORITHM_LINK_QUEUEMODE_NOTINPLACE;
3.  outputQInfo.queInfo.numCh = numChannelsUsed;
4.
5.
6.  for(channelId = 0; channelId < numChannelsUsed; channelId++)
7.  {
8.      memcpy((void *)&(outputQInfo.queInfo.chInfo[channelId]),
9.             (void *)&(prevLinkInfo.queInfo[prevLinkQueId].chInfo[channelId]),
10.            sizeof(System_LinkChInfo)
11.           );
12.
13. }
```

4.3.3.2 In Place Mode

Let us consider Color To Gray algorithm for In Place mode of operation. In Color To Gray algorithm, input buffers are modified and output buffers are not created at all. Input buffers from the previous link are sent as output buffers to the next link. So we need the following mechanisms to manage input and output queues.

1. Managing Input Queues

   At the create time of the plug-in, we need to specify the mode of operation as ALGORITHM_LINK_QUEUEMODE_INPLACE. The code snippet is shown below,

```
1.  AlgorithmLink_InputQueueInfo  inputQInfo;
2.  inputQInfo.qMode  = ALGORITHM_LINK_QUEUEMODE_INPLACE;
```

Since the input buffer is modified and acts as the output buffer (which becomes input buffer to the next link), we cannot release this buffer to the previous link immediately. After the algorithm process call is finished we need to give this buffer to the next link by calling AlgorithmLink_putFullOutputBuffer. The code snippet is shown below,

```
1.  Alg_ColorToGrayProcess(algHandle,
2.                         (UInt32 **)pSysVideoFrameBufferInput->bufAddr,
3.                          pInputChInfo->width,
4.                          pInputChInfo->height,
5.                          pInputChInfo->pitch,
6.                          dataFormat
7.                          );
8.  status = AlgorithmLink_putFullOutputBuffer(pObj,
9.                                             outputQId,
10.                                            pSysBufferInput
11.                                            );
12. UTILS_assert(status == SYSTEM_LINK_STATUS_SOK);
```

When the algorithm wants to give back buffers to the previous link, we must call AlgorithmLink_releaseInputBuffer.

2. Managing output queues

   At the create time of the plug-in, we need to specify the mode of operation as ALGORITHM_LINK_QUEUEMODE_INPLACE. Along with the mode we also need to specify few other parameters that are important for INPLACE mode. The code snippet for Color To Gray algorithm is shown below,

```
1.  AlgorithmLink_OutputQueueInfo outputQInfo;
2.  outputQInfo.qMode = ALGORITHM_LINK_QUEUEMODE_INPLACE;
3.  outputQInfo.inputQId = 0;
4.  memcpy((void*)(&outputQInfo.inQueParams),
5.         (void*)(&pColorToGrayCreateParams->inQueParams),
6.          sizeof(outputQInfo.inQueParams)
7.        );
```

4.3.4 Algorithm Internal Memory

Algorithm might need some memory for its operation, which is internal to the algorithm. This memory might be present in DDR / OCMC / L2. Such memory requests for the algorithm needs to be catered to in create plug in function. Algorithm plugin can call mallocs as shown in example to obtain these memories from framework. Algorithm create plug-in can interact with Algorithm using any interface for these memory requests and grants. It is recommended to use XDAIS:memTab as the interface.

All the memories requested during Create phase needs to be freed up in the delete plug-in function.

4.3.5 Cache Operations

In scenarios where buffers are touched by both CPU and DMA, there could be coherence issues. In order to avoid these issues, Cache invalidations / write backs might have to be used. BIOS based cache APIs can be used. These APIs can be called from algorithm plug-in OR directly from the algorithm itself.

Following two APIs are typically used:

Cache_inv() – To invalidate cache for a buffer

Cache_wb() – To write back cache contents into DDR

4.4 Algorithm Link Plug-In Integration

As stated in the previous section algorithms are developed as plug-ins into the algorithm link. Following are the sub steps involved in developing a plug in and integrating it into the SDK.

4.4.1 Creating public header file

Once algorithm plug-in functions are implemented, an API to use the link needs to be defined. So we need to have a public header file which contains,

1. Link API commands : Commands to control plugin behavior
2. Create time parameters : Structure containing create time parameters.
3. Control parameters : Structure containing control parameters
4. Function that registers plugin functions into the function table

Note: For create time parameters and control time parameters, the algorithm skeleton defines a base structure. This base structure can be extended with more elements to cater to the needs of particular algorithm. However, it is to be noted that the first member of the parameters structure must be AlgorithmLink_CreateParams for create time parameters and AlgorithmLink_ControlParams for control parameters . For example, create time parameters of Frame Copy algorithm is shown below,

```
typedef struct
{
    AlgorithmLink_CreateParams baseClassCreate;
    /**< Base class create params */
    UInt32                     maxHeight;
    /**< Max height of the frame */
    UInt32                     maxWidth;
    /**< max width of the frame */
    UInt32                     numOutputFrames;
    /**< Number of output frames to be created for this link per channel*/
    System_LinkOutQueParams    outQueParams;
    /**< Output queue information */
    System_LinkInQueParams     inQueParams;
    /**< Input queue information */
    AlgorithmLink_CopyMode     copyMode;
    /**< CPU or DMA mode of frame copy */
} AlgorithmLink_FrameCopyCreateParams;
```

4.4.2 Plugin Registration

Once a plug-in is developed, it has to be registered with the framework. To register a plug-in, developer has to populate a set of function pointers so that appropriate functions can be called by the algorithm Link. Typically this registration is done in a separate function. Algorithm Id needs to be passed to AlgorithmLink_registerPlugin to register plug-in.

In Color To Gray algorithm, this is done in AlgorithmLink_ColorToGray_initPlugin(). The code snippet is shown below,

```
Int32 AlgorithmLink_ColorToGray_initPlugin()
{
    AlgorithmLink_FuncTable pluginFunctions;
    UInt32 algId = (UInt32)-1;

    pluginFunctions.AlgorithmLink_AlgPluginCreate =
        AlgorithmLink_ColorToGrayCreate;
    pluginFunctions.AlgorithmLink_AlgPluginProcess =
        AlgorithmLink_ColorToGrayProcess;
    pluginFunctions.AlgorithmLink_AlgPluginControl =
        AlgorithmLink_ColorToGrayControl;
    pluginFunctions.AlgorithmLink_AlgPluginStop =
        AlgorithmLink_ColorToGrayStop;
    pluginFunctions.AlgorithmLink_AlgPluginDelete =
        AlgorithmLink_ColorToGrayDelete;

#ifdef BUILD_DSP
    algId = ALGORITHM_LINK_DSP_ALG_COLORTOGRAY;
#endif

#ifdef BUILD_ARP32
    algId = ALGORITHM_LINK_EVE_ALG_COLORTOGRAY;
#endif

#ifdef BUILD_A15
    algId = ALGORITHM_LINK_A15_ALG_COLORTOGRAY;
#endif

    AlgorithmLink_registerPlugin(algId, &pluginFunctions);

    return SYSTEM_LINK_STATUS_SOK;
}
```

This function needs to be called at the initialization of Algorithm Link. So as to make this happen, developer needs to add a call to this function in AlgorithmLink_initAlgPlugins() located at the following location,

\vision_sdk\links_fw\src\rtos\links_common\algorithm\algorithmLink_cfg.c

4.5 Directory Structure and Make File Changes

Plug-in related files have to be placed in proper locations so as to include in the build. Also make file changes have to be done. The below table gives the details of the location of files and corresponding make file changes, if any.

S.No	File Name	Location	MakeFile Change Needed (Yes/No)	Location of MakeFile
1	XxxxLink_algPlugin.c	\vision_sdk\apps\src\rtos\alg_plugins\plugin_name	Yes	\vision_sdk\apps\src\rtos\alg_plugins

5 Porting Vision SDK

The following sections cover different aspects related to porting Vision SDK to custom hardware.

Please note that this section covers only information specific to Vision SDK. Individual components like SBL, BSP etc will need changes as part of the porting activity and these are not covered here.

5.1 Using custom memory map

Memory map for the Vision SDK is defined in the file \vision_sdk\apps\build\tda2xx /mem_segment_definition_<bios/linux>.xs

Important things to pay attention to while porting the map file are:

- DDR, OCMC & DSP/EVE SRAM sizes
- Core specific code/data/vecs sizes
- Size of the shared frame buffer pool

DDR is divided into 2 sections: cached and non-cached. The cached part is used for frame buffer and core specific code/data and other sections. The non-cached part is used for Vision SDK log buffers, HDVPSS descriptors.

5.2 Support for custom core selection

The cores to be included can be controlled using the file vision_sdk/build/Rules.make

Please modify the file by setting the core specific defines (PROC_DSP1_INCLUDE etc) to the appropriate value (yes/no). By default all cores (A15_0, IPU1_0, IPU1_1, DSP1, DSP2, EVE1, EVE2, EVE3 and EVE4) are enabled in the vision_sdk.

5.3 Support for custom board

For adding support for a new board, the board specific init, de-init and probe APIs need to be implemented. Please refer to the files \vision_sdk\apps\src\rtos\board folder for examples related to TI EVM.

Once defined, these specific APIs can be invoked from the use case implementation (chain) based on the required configuration.

5.4 Support for different video capture device

For adding support for a different video capture device, the device specific create, delete & control APIs need to be implemented. Please refer to the files \vision_sdk\apps\src\rtos\devices folder for examples related to video sensor (OV10635) and HDMI receiver.

Once defined, these specific APIs can be invoked from the use case implementation (chain) based on the required configuration.

5.5 Support for different LCD

For adding support for a different LCD, the LCD specific APIs need to be implemented. Please refer to the files \vision_sdk\apps\src\rtos\devices\lcd.c file for example.

Once defined, these specific APIs can be invoked from the use case implementation (chain) based on the required configuration.

5.6 Specifying custom core frequencies to BIOS

API Utils_getClkHz() defined in utils.h is used to read PLL values and then tell BIOS about the core frequency that is programed. BIOS needs to be told exact frequency since it configures its timer based on this frequency. BIOS configuration is done in main() for the specific core via the API Utils_setCpuFrequency(). No action is needed from the user to configure BIOS when PLL settings are changed via SBL **unless reference clock is changed**. By default EVM uses 20Mhz reference clock. If custom board uses another reference clock then change in file utils_clk.c (\vision_sdk\links_fw\src\rtos\utils_common\src) #define UTILS_SYS_CLK1 (20*1000*1000)

5.7 Using custom PLL and clock settings

The PLL and clock settings for different peripherals can be modified as required (based on the input crystal frequency & specifications) either in the SBL (for out of box execution) or gel file (CCS based execution).

Default frequency configurations used as part of vision sdk are captured in the data sheet. TI EVM uses the 20MHz input crystal.

6 Boot time optimizations on TDA3X

This chapter describes various techniques that can be applied through vision_sdk for optimizing boot time. This will be typically helpful for rear view camera systems.

Vision SDK demonstrates boot times as low as **500 to 600 ms** through example.

6.1 Usecase supported for fast boot demonstration

Users can refer to following usecase to tryout demo on fast boot

1 ch ISS capture (OV10640) + Object Detect + Display (LCD 10 inch)

Path in vision_sdk –

\vision_sdk\apps\src\rtos\usecases\fast_boot_iss_capture_isp_simcop_pd_display

This is a special usecase which not listed in Run time Menu. For more details on h/w setup and how to run this usecase please refer VisionSDK_UserGuide_TDA3xx.pdf under vision_sdk\docs folder.

For boot time numbers and time split since boot, please refer respective Data Sheet.

6.2 Optimizations challenges

There are multiple challenges when boot time needs to be reduced, few of them are mentioned below

1. Dividing usecase Data Flow into parts and selectively bringing up s/w modules
2. Delayed loading of CPUs/Cores
3. Synchronization of lately loaded CPUs/cores with early loaded cores
4. Seamlessly switching video data to lately loaded modules (without glitches on display)
5. Reducing initialization time of modules in boot time path
6. Compiler / linker optimizations
7. Reducing image size
8. Reducing sensor initialization time
9. Identifying unnecessary delays and eliminating them
10. Choosing boot medium

6.3 Techniques for boot time optimization (Framework level)

All of the challenges listed above cannot be addressed only through framework, we need support from all levels i.e. hardware, boot loader, frame work. This section in detail covers optimizations at framework level only and briefly introduces to other optimizations (e.g. h/w or boot loader optimizations).

6.3.1 Gate Link

This is a special link in vision_sdk with following features and functionalities

- Gate link is like any other link in vision_sdk but with one input and one output queue always.

- It acts as on/off switch, when the operation mode is "ON" it simply forwards buffers to next link, on the other hand it returns received buffers back to previous link if its "OFF".
- The default the operation mode of Gate Link instance is "OFF".
- The state can be changed by using a system command, while the data flow is running.
- It does not own/manage any output buffer, based on state it will either forward or return data to next/previous link.
- It ensures callbacks are forwarded to previous link of the GateLink while freeing buffers and to GateLink's next Link while putting full buffers.

6.3.1.1 Usage

Essentially Gate Link allows dividing your data flow into two parts

- **UcEarly** – Usecase that needs to come up first
- **UcLate** – Usecase that has no implication on boot time and can be brought up late.

Boot time optimization is one of the features that are implemented using Gate Link. This link can also be used to implement Power Management features as you can selectively turn on and off parts of the data flows.

This link is supported through usecase generation tool and can be used in any usecase from vision_sdk 2.7 onwards as per need.

6.3.1.2 Example

Let's look at example mentioned in section 6.1 and see how Gate Link can be used to achieve *usecase division*.

Typical data flow for Object Detect usecase looks as mentioned in the figure below.

Here, Capture and display is happening on IPU1_0 while processing will happen on DSPs or EVEs available in the system.

If you need to have minimum POR to Display time, you can not waste time in creating other Links/modules which are not in the capture -> display path.

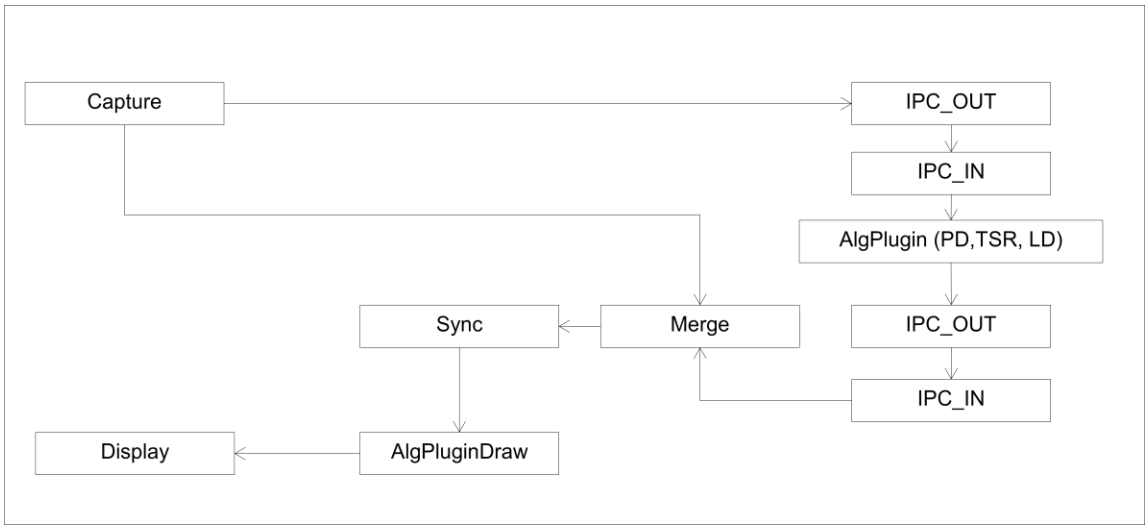


Figure: Object Detect usecase without Gate Links

User can segregate group of links that can be brought up first and group of link that can be brought up later.

Figure below shows how Object Detect is divided into **UcEarly** and **UcLate** using Gate Link instances in between

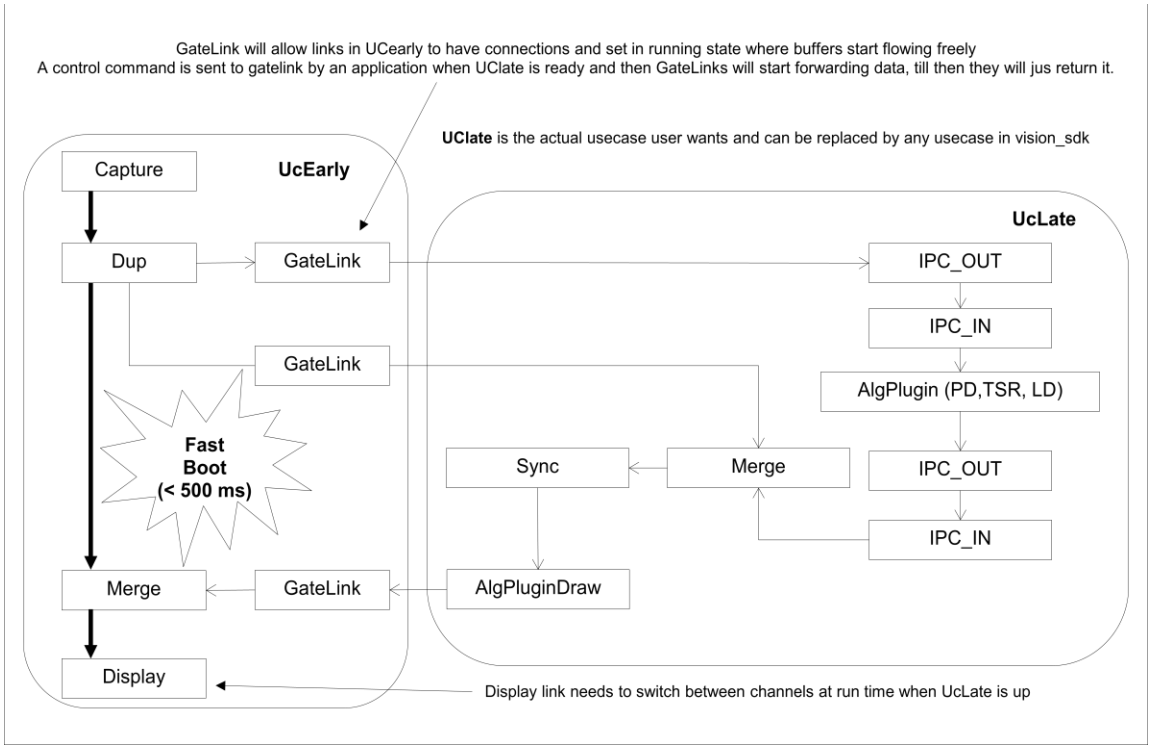


Figure: Object Detect usecase with Gate Links

Idea here is, upon POR UcEarly (IPU1_0) executes and flashes up display with preview in less than 500 ms. In due course of time, IPU1_0 or master core application boot loads other cores and instantiates UcLate. As soon as UcLate is up and running, Gate Link instances are switched "ON" using a system command and then application switches channel at display showing algorithm output.

Users can practically replace UcLate from above figure with any usecase of their choice to achieve fast boot.

6.3.2 Selective loading and Power management of cores

Users need to decide which cores need to boot first and which can be boot later. In TDA3X since IPU1_0 is a master core and owns capture and display subsystem this needs to come up first, in some cases CAN stack comes up on IPU1_1 and CAN response time needs to be minimal so IPU1_1 also comes up along with IPU1_0.

This can be achieved by having two AppImages instead of one.

AppImage_UcEarly_BE – contains IPU1_0 and IPU1_1 images

AppImage_UcLate_BE – contains DSP1, DSP2 and EVE1 images

This is achieved using a script file in vision_sdk MulticoreImageGen_tda3xx_fast_boot.sh or MulticoreImageGen_tda3xx_fast_boot.bat

Upon POR, SBL picks up only AppImage_UcEarly_BE flashed into fixed offset (e.g. 0x80000) in QSPI and loads IPU1_0 and IPU1_1. While AppImage_UcLate_BE is flashed on another location within QSPI known location to application (e.g. 0xA80000). IPU1_0 application after capture + display usecase (UcEarly), parses this AppImage_UcLate_BE and boot loads slave cores using SBL lib.

6.3.2.1 Delayed loading of slave cores using SBL lib

Starterware bootloader (SBL) provides APIs which can be used to selectively load and run DSPs and EVEs from IPU. Application writers can choose to use those directly or call following sequence of functions through vision_sdk app which encapsulates SBL lib APIs

……

```
    Utils_BootSlaves_Params bootParams;
    ddrAddress       =       Utils_memAlloc(UTILS_HEAPID_DDR_CACHED_SR,
MAX_UCLATE_IMAGE_SIZE, 4U);
    Utils_bootSlaves_paramsInit(&bootParams);
    bootParams.offset = IMAGE_UCLATE_OFFSET_QSPI;
    bootParams.ddrAddress = (UInt32)(ddrAddress);
    bootParams.useEdma = TRUE;
    bootParams.loadCode = TRUE;
    bootParams.maxDdrBuffSize = MAX_UCLATE_IMAGE_SIZE;
    bootParams.enableCrc = TRUE;
    bootParams.useEdma = TRUE;
```

```
Utils_bootSlaves(&bootParams);
```

……

The `Utils_bootSlaves()` function ensures all slave cores are loaded and running as it would have been done by SBL with single AppImage!!! It also takes care of their power states and brings them to their respective `main()`.

`bootParams.offset` tells the application where the AppImage_UcLate_BE is flashed in QSPI.

`bootParams.ddrAddress` tells the DDR location where each core's image is copied before parsing.

`bootParams.useEdma` mentions which method to be used while copying sections after AppImage_UcLate_BE is parsed. If this is FALSE, IPU1_0 memcopies sections into respective load addresses otherwise it usecase EDMAs

`bootParams.maxDdrBuffSize` tells the maximum available size of DDR buffer where each core's image is copied before parsing.

`bootParams.enableCrc` mentions whether CRC check should be done on the UC Late Multi-core application image.

For more information on SBL lib APIs users can refer PROCESSOR_SDK_VISION_XX_XX_XX_XX\ti_components\drivers\pdk\packages\ti\boot\sbl_auto\sbl_lib\sbl_lib.h

6.3.2.2 *Syncing up with lately loaded cores*

Just by selectively loading cores will have them working separately but not together. To have them work together users need to sync them with already running cores (IPU1_0 and IPU1_1).

vision_sdk provides a function `Utils_syncSlaves()` that synchronizes early and lately loaded cores for enabling their Inter Processor Communication.

Important Note- If users are using `Utils_bootSlaves()` in the application to selectively boot load their cores they must use `Utils_syncSlaves()` followed by that to synchronize them.

6.3.3 Seamless switch between UcEaly and UcLate

This is very important from user experience perspective, there can be a noticeable glitch when display switches channels from UcEarly to UcLate, this can be mainly caused either by other tasks in the system e.g. GrpxSrc

The task priority maters here, users must carefully choose lower priorities for tasks like GrpxSrc or System tasks which don't have much role to play after initialization.

Using memcpy instead of edma to copy sections slave core's memory can also partially attribute to this.

6.3.4 Other Optimizations

6.3.4.1 H/w changes

VisionSDK_UserGuide_TDA3xx.pdf in detail explains h/w changes to support I2C at 400KHz, if I2C is running at lower speed, it can attribute to higher / worst sensor initialization times.

Users must ensure these changes are done on EVM before any s/w optimization is tried out

6.3.4.2 Sensor initialization time

Depending upon sensor chosen and I2C speed in the system this time varies drastically, it is left to users to optimize this but it contributes to almost 40% boot time for fast boot.

Users should carefully choose only registers that are needed for sensor initialization for the chosen sensor and reduce I2C commands to minimal possible. This will reduce sensor initialization time and give better boot time.

6.3.4.3 Boot media and image size

To achieve fast boot usecase user needs to choose fastest media available on the board as boot media, e.g. QSPI for tda3x evm

The image size is very subjective to application but it can be optimized by excluding modules that are of no relevance to user's usecase. One thing that users need to ensure is time taken to read and write from boot media is as expected.

e.g. If you are flashing image of size 4.5 MB to QSPI, IPU should practically get close to 30 MB/s speed for read, it should not take more than 160-170 ms to read this whole image into DDR. Users need to cross check this for their appImages.

6.3.4.4 Compiler and linker optimizations

Based on the type of master core and its compiler these options vary, users can look at rules_m4.mk to see compiler and linker options used for fast boot (e.g. –O3, --ram_model etc).

Users need to refer respective manuals for compiler and linkers to find more optimization options and use them carefully.

6.4 Steps to convert a usecase into fast boot usecase

1. Identify places in the usecase where GateLink instances need to be introduced into the data flow.
    a. Note: You need an addition Dup (logically after capture) and Merge link (before display) to separate UcLate from UcEarly, as shown in section 6.3.1.2
2. Create the chains_fastBoot_<new_usecase>.txt for input to usecase generation tool, refer example

    \vision_sdk\apps\src\rtos\usecases\fast_boot_iss_capture_isp_simcop_pd_display\chains_fastBoot_issIspSimcop_pd_Display.txt

3. Generate Usecase using usecase generation tool. Refer VisionSDK_UsecaseGen_Overview.pdf
    a. vsdk_linux.out –img –file chains_fastBoot_<new_usecase>.txt
4. Manually modify generated chains_fastBoot_<new_usecase>_priv.c and chains_fastBoot_<new_usecase>_priv.h to split two functions as followed
    a. Split chains_fastBoot_<new_usecase>_**Create()** to chains_fastBoot_<new_usecase>_**Create_UcEarly**() and chains_fastBoot_<new_usecase>_**Create_UcLate**()
    b. Split chains_fastBoot_<new_usecase>_**Start()** to

       chains_fastBoot_<new_usecase>_**Start_UcEarly**() and chains_fastBoot_<new_usecase>_**Start_UcLate**()
    c. Move calls System_linkCreate/Start logically to respective UcEarly or UcLate functions
    d. Modify chains_fastBoot_<new_usecase>.c (note this is not generated one) to split chains_fastBoot_<new_usecase>_StartApp() to split into

       chains_fastBoot_<new_usecase>_**StartApp_UcEarly**() and chains_fastBoot_<new_usecase>_**StartApp_UcLate**()

       Ensure functions newly create in 2.C above are called from respective StartApp_UcEarly() / StartApp_UcLate().
5. Refer \vision_sdk\apps\src\rtos\usecases\fast_boot_iss_capture_isp_simcop_pd_display\chains_fastBoot_issIspSimcop_pd_Display.c

   to understand modifications needed in respective chains_fastBoot_<new_usecase>.c file
    a. Modify Chains_fastBoot<new_usecase>(), main usecase function to call
        i. newly created functions in 4 above
    b. Modify Chains_fastBoot<new_usecase>(), main usecase function to call
        i. Utils_bootSlaves() and Utils_syncSlaves()
    c. Add System commands to switch "ON" Gate Links for all the instances of gates in your usecase
    d. Add System command to switch display channel from preview to algorithm output
6. Modify Chains_main() in \vision_sdk\apps\src\rtos\common\chains_main_bios.c to call Chains_fast<new_usecase>() with right parameters.

7 Power Optimization in Vision SDK

This section describes the ways in which the Vision SDK can be power optimized to make sure the thermal dissipation of the device is within the budget.

7.1 Putting CPUs to Low Power when not used

Different CPUs in the system can be configured to go to their different low power modes when they have nothing to execute:

1. A15 C0 to Retention (C1 is placed to forced off state from SBL) - `\vision_sdk\links_fw\src\rtos\utils_common\src\utils_idle_a15.c`
2. M4 C0 and C1 (IPU) to Auto Clock Gate. - `\vision_sdk\links_fw\src\rtos\utils_common\src\utils_idle_m4.c`
3. DSP 1 and DSP2 to Auto Clock Gate - `\vision_sdk\links_fw\src\rtos\utils_common\src\utils_idle_c66x.c`
4. EVE 1/2/3/4 to ARP32 to Auto Clock Gate - `\vision_sdk\links_fw\src\rtos\utils_common\src\utils_idle_arp32.c`

7.1.1 Setting up the CPUs for Low Power

In order to place any of the cores in the desired low power mode when the CPU does not have anything to execute requires two steps:

- Configuration of the PRCM and the subsystem to allow the subsystem to enter the desired power state when the CPU executes IDLE or WFI Instruction. (`Utils_idlePrepare`)
- CPU executing the Idle/WFI instruction. (`Utils_idleFxn`)

The `Utils_idlePrepare` function needs to be called before BIOS_Start to initialize the CPU correctly before the BIOS Scheduler schedules tasks on the CPU.

The `Utils_idleFxn` function is registered with the BIOS scheduler as one of the Idle tasks. This is done in each of the CPUs BIOS Configuration Files in the directory \vision_sdk\links_fw\src\rtos\bios_app_common\<SOC>\<CPU CORE>.

```
/* Add an idle thread 'Utils_idleFxn' that monitors interrupts. */
    var Idle = xdc.useModule("ti.sysbios.knl.Idle");
    Idle.addFunc('&Utils_idleFxn');
```

7.1.2 BIOS Tick and Time Stamp Provider Concerns

When the CPU is placed into a low power mode when the Idle Task is hit the CPU clocks are gated which causes any BIOS tick clock or the Time stamp provider clocks to be gated if internal CPU Timers are used. The impact of this is that the Task_sleep(); and CPU Load calculation functionalities would be broken. In order to avoid this, the system level GP Timers can be used to provide BIOS Tick and Time Stamp Provider Proxy. This configuration as well can be placed in the CPU specific BIOS configuration file. An example configuration for A15 is as shown below. The portion highlighted in yellow configures GP Timer 2 for BIOS Ticks and the portion highlighted in green configures GP Timer 3 for Time Stamp Provider used for CPU Load calculation.

```
/*********************************************
*           Timer Module Configuration         *
*********************************************/
/* Assign GPTimer2 to be used for Timestamp */
/* Set to 1-ms Tick and Enable Wakeup for OVF interrupt */
```

```
var Timer = xdc.useModule('ti.sysbios.timers.dmtimer.Timer');
var timerParams = new Timer.Params();
timerParams.period = 1000;
timerParams.twer.ovf_wup_ena = 1;
timerParams.tiocpCfg.emufree = 1;
timerParams.tsicr.posted = 0;
/* Timer ID = 1 for GPTimer2 and input clock runs at 20 MHz */
Timer.intFreqs[1].hi = 0;
Timer.intFreqs[1].lo = 20000000;
Timer.create(1, '&mainA15TimerTick', timerParams);

/* Assign GPTimer3 to be used for Timestamp */
/* Timer ID = 2 for GPTimer3 and input clock runs at 20 MHz */
var DMTimer = xdc.useModule('ti.sysbios.timers.dmtimer.Timer');
var timerParams2 = new DMTimer.Params();
timerParams2.tsicr.posted = 0;
DMTimer.intFreqs[2].hi = 0;
DMTimer.intFreqs[2].lo = 20000000;
var DMTimestampProvider =
xdc.useModule("ti.sysbios.timers.dmtimer.TimestampProvider");
DMTimestampProvider.timerId = 2;
DMTimestampProvider.useClockTimer = false;
var Timestamp = xdc.useModule("xdc.runtime.Timestamp");
Timestamp.SupportProxy = DMTimestampProvider;

/* Indicate GPT2 & GPT3 are used */
var TimerSupport =
xdc.useModule('ti.sysbios.family.shared.vayu.TimerSupport');
TimerSupport.availMask = 0x0006;
```

The following Table defines which timers are used for which CPUs

CPU	TDA2xx Timers	TDA2ex Timers	TDA3xx Timers
A15_0	GP Timer 2 and 3	GP Timer 2 and 3	NA
IPU1_0	GP Timer 9 and 11	GP Timer 9 and 11	GP Timer 3 and 4
IPU1_1	GP Timer 9 and 11	GP Timer 9 and 11	GP Timer 3 and 4
DSP1	GP Timer 5 and 6	GP Timer 5 and 6	GP Timer 1 and 2
DSP2	GP Timer 5 and 6	NA	GP Timer 1 and 2
EVE1	GP Timer 13 and 14	NA	GP Timer 7 and 8
EVE2	GP Timer 13 and 14	NA	NA
EVE3	GP Timer 13 and 14	NA	NA
EVE4	GP Timer 13 and 14	NA	NA

7.1.3 Disabling CPU Idle for Debug

Optionally if one wants to disable the CPU Idle from making the CPUs go into their respective low power states one can configure vision_sdk/build/Rules.make to update the following build configuration to ‘no’

#

```
# Used to enable or disable CPU idle functionality in SDK
# By Default CPU idle is enabled
#
CPU_IDLE_ENABLED=yes

ifeq ($(PROFILE),debug)
CPU_IDLE_ENABLED=no
endif
```

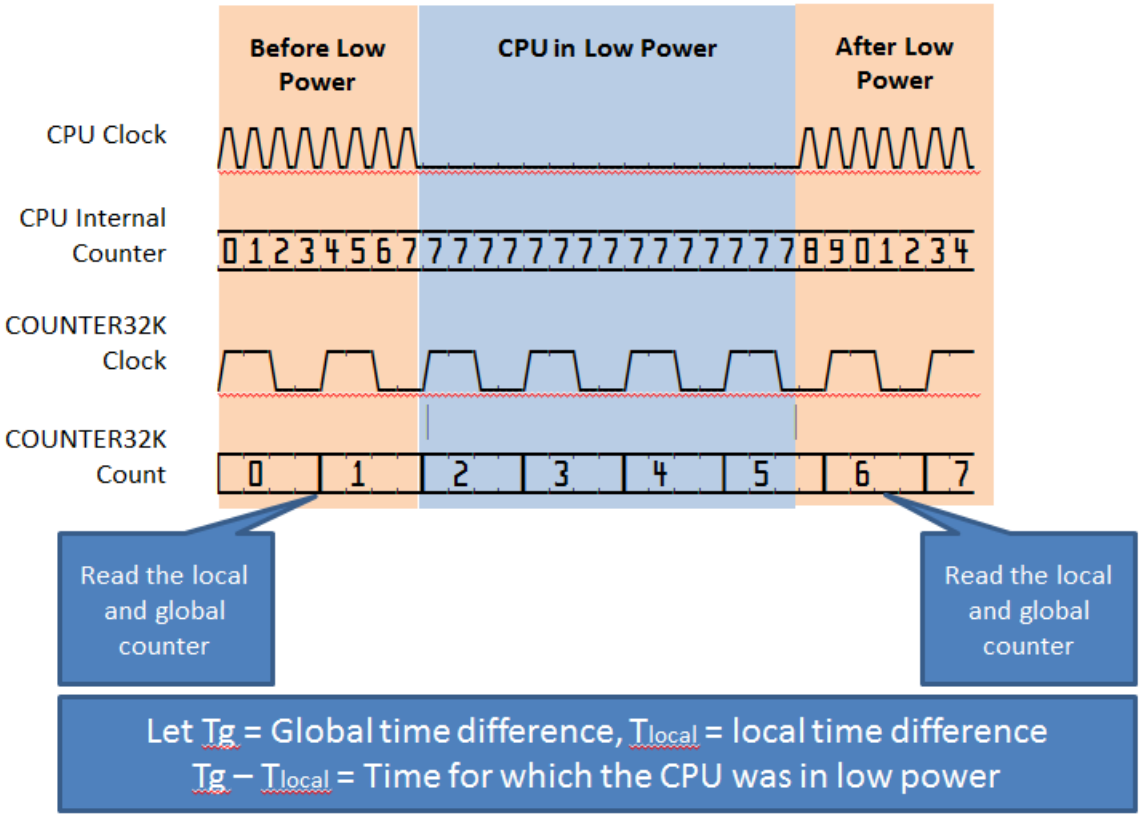
For more information on the APIs used to achieve the CPU Idle functionality refer the ADAS Power Management Application Note.

7.1.4 Knowing the time for which the CPU is in Idle

With the goal of functional safety, it is often important to know the amount of time the CPU has been in low power. The state of the system, dependencies to different clock domains and pending interrupts on the CPU can disallow the CPU to go to low power mode.

A mechanism has been designed to understand the time for which the CPU is in low power. This mechanism utilizes the COUNTER32K module which does not get clock gated when the CPU is in low power and an internal counter/timer. The internal counter or timer is chosen such that the counter/timer is within the CPU subsystem such that when the CPU is clock gated the counter/timer stops counting.

The basic premise of the mechanism is as shown below:



The code for the time difference calculation is given in the function `Utils_idleFxn`.

7.2 Limp Home Mode

When the temperature of the device becomes too high it is often required the software start taking steps to cool the device by making sure the power dissipation of the device is lowered. The Single Channel FC Analytics use case has been modified to showcase this feature of limp home mode which causes the capture FPS to drop to 10 fps from the default 30 fps when the temperature hot event is hit and then again re-configure the fps to 30 fps when the device has cooled down and the thermal cold event is reached.

7.2.1 Initializing the system for Limp Home Mode

\vision_sdk\links_fw\src\rtos\utils_common\src\utils_temperature.c has the necessary API calls to configure the on chip temperature sensors to provide thermal alerts and read the current temperature. Essentially the Utils_tempConfigInit function configures the on chip BGAP Temperature sensors to provide a thermal hot event at 80 deg C (interrupt is generated when the chip temperature goes higher than 80 deg C) and a thermal cold event at 10 deg C (interrupt is generated when the chip temperature is lower than 10 deg C). The interrupt handler (`Utils_tempBgapEventIsr`) is also registered in this function which reads the status of the BGAP registers to find which temperature sensor has raised the thermal event. (There are 5 temperature sensors in TDA2xx and 1 in TDA3xx).

Note: IPU Interrupt number 61 is used to register the Thermal event.

The Utils_tempConfigInit function should be called from the Usecase initialization function (`chains_vipSingleCameraAnalytics_tda2/3xx_StartApp`). The application should also register the thermal handler in the System Link during application initialization. e.g.

```
SystemLink_registerHandler(Chains_vipSingleCameraAnalyticsTda2/3xx_EventHandler);
```

This handler is called when the thermal event is received and the ISR (`Utils_tempBgapEventIsr`) calls the System Call to invoke the thermal Handler. The flow is described in the subsection below. The calls for the Thermal system initialization and the system event handler registration can be seen in the files:

```
apps/src/rtos/usecases/vip_single_cam_analytics_tda2xx/chains_vipSingleCameraAnalytics_tda2xx.c
```

```
apps/src/rtos/usecases/vip_single_cam_analytics_tda3xx/chains_vipSingleCameraAnalytics_tda3xx.c
```

7.2.2 Handling Thermal Events

When a hot event occurs the Temperature ISR `Utils_tempBgapEventIsr` is called. This ISR determines which voltage domain has crossed the thermal threshold by reading the BGAP registers through the PMHAL APIs and then makes a system call which gives the voltage domain ID and the type of the thermal event (eg. HOT/COLD).

```
System_linkControl(SYSTEM_LINK_ID_IPU1_0,
                   UTILS_TEMP_CMD_EVENT_HOT,       /* Event Type */
                   &gUtils_tempObj[voltId].voltId, /* Voltage ID */
                   sizeof(pmhalPrcmVdId_t),
```

```
                    FALSE);
System_linkControl(SYSTEM_LINK_ID_IPU1_0,
                    UTILS_TEMP_CMD_EVENT_COLD,      /* Event Type */
                    &gUtils_tempObj[voltId].voltId, /* Voltage ID */
                    sizeof(pmhalPrcmVdId_t),
                    FALSE);
```

The `UTILS_TEMP_CMD_EVENT_HOT` or `UTILS_TEMP_CMD_EVENT_COLD` is defined in the header file `\vision_sdk\links_fw\src\rtos\utils_common\include\utils_temperature.h`. The System Link call does not wait for an acknowledgement from the registered handler to ensure the Interrupt context is not maintained for a long time.

Once the thermal handler `Chains_vipSingleCameraAnalyticsTda2/3xx_EventHandler` the function first checks which event type is it and then based on whether the event is a HOT or COLD one it calls the appropriate hot event handler (`ChainsCommon_tempHotEventHandler`) or cold event handler `(ChainsCommon_tempColdEventHandler)`. These functions take in parameters of the captureLink ID and the pointer to the voltage domain ID which is sent by the temperature ISR. These functions are defined in the file `vision_sdk/apps/src/rtos/usecases/common/chains_common_fc_analytics.c`.

The hot or cold event handler in its current implementation only handles the thermal events generated by VD_CORE. This is sufficient while trying to implement thermal handling coarsely as the other temperature sensors are often +/- 5 deg C difference from the VD_CORE temperature.

Based on the capture sensor the event handler decides the necessary frameSkipMask which is a capture driver parameter which decides which frames to drop to achieve the desired FPS of capture. For the frames which are skipped the VPDMA does not copy these frames to DDR. This is useful as the DDR IO power is saved as memory transactions are minimized by dropping frames at the capture thread.

The `CaptureLink_tsk` has been modified to handle the requests from the thermal handler or any other handler to change the frame skip parameters on the fly. A new command `CAPTURE_LINK_CMD_SET_FRAME_SKIP_MASK` has been added to achieve this functionality defined in `include/link_api/captureLink.h`. A function `CaptureLink_drvUpdateFrmSkip` has been added to the file `vision_sdk/links_fw/src/rtos/links_ipu/vip_capture/captureLink_drv.c` to make the driver IOCTL call which updates the frameSkip parameter for all the capture streams and instances.

The handler then modifies the thermal temperature thresholds by calling the `Utils_tempChangeHotThreshold` and `Utils_tempChangeColdThreshold`. The new threshold values can be determined by the application. As an example the thresholds in the current implementation has been changed to the current temperature + step size for the next hot event and current temperature – step size for the next cold event. This method of using the current temperature to generate the next thermal thresholds helps in tracking the temperature to handle events of slowly rising or falling ambient temperatures. The APIs `Utils_tempChangeHotThreshold` and

`Utils_tempChangeColdThreshold` re-enable the thermal interrupt to service future thermal events.

Note: Since the APIs `Utils_tempChangeHotThreshold` and `Utils_tempChangeColdThreshold` re-enable the IPU interrupts for thermal events it is essential to maintain a certain order of calling these APIs in the thermal event handlers to ensure we do not get false/double interrupts. For instance in the Hot event handler the change of thersholds should be HOT first then COLD to ensure the hot temperarture threshold is changed before the interrupt is enabled. In the case of the cold event handler the COLD threshold should be changed before hot. If the other way around is done the IPU may receive an extra interrupt which is caused by the interrupt being enabled but the temperature threshold being unchanged.

When the Limp Home actions have been taken, it is important to notify the user regarding the state of the system (Limp Home or Not). The Handler also additionally updates the thermal state variable to indicate this by calling the function

Utils_tempUpdateAllVoltLimpHomeState(UTILS_TEMP_LIMP_HOME_ACTIVE);

in the Hot event handler and calling the function

Utils_tempUpdateAllVoltLimpHomeState(UTILS_TEMP_LIMP_HOME_INACTIVE);

in the Cold event Handler.

Additionally the display is updated with the mode by sending a GRPX_SRC_LINK_CMD_PRINT_STRING message to the GRPX Link as below:

```
snprintf(printPrms.stringInfo.string,
                sizeof(printPrms.stringInfo.string) - 1,
                "                          \n");
  printPrms.stringInfo.string[
            sizeof(printPrms.stringInfo.string) - 1] = 0;
  printPrms.duration_ms = LIMP_HOME_DISPLAY_DURATION_MS;
  printPrms.stringInfo.fontType = LIMP_HOME_DISPLAY_FONTID;
  printPrms.stringInfo.startX  = pObj->displayWidth/2 + 200;
  printPrms.stringInfo.startY  = pObj->displayHeight-100;

  status = System_linkControl(IPU1_0_LINK(SYSTEM_LINK_ID_GRPX_SRC_0),
                    GRPX_SRC_LINK_CMD_PRINT_STRING,
                    &printPrms,
                    sizeof(printPrms),
                          TRUE);
```

7.2.3 Run-time configurability

The use case menu has been modified to demonstrate the thermal management capability of the software. In the use case menu an extra option 't' has been added as shown below:

```
====================
 Chains Run-time Menu
 ====================

 0: Stop Chain

 p: Print Performance Statistics

 t: Show Thermal Configuration Menu
```

```
 Enter Choice:
```

Once the ‘t’ option is chosen the following menu options are shown:

```
==============================
Thermal Management Description
==============================

The Thermal Management of the device involves reducing the power
consumption when the temperature of the device becomes hotter than
the desired THOT temperature. When the temperature becomes lower than
TCOLD temperature the device power consumption can be restored.

The control of power consumption is done by reducing the FPS of
of the usecase by dropping frames in the capture thread.

After every thermal event the temperature thresholds would be changed
to make sure THOT and TCOLD are as below:

     THOT Temperature    ------------------------
                                 ^
                                 |
                            Step Size
                                 |
                                 V
     Current Temperature ------------------------
                                 ^
                                 |
                            Step Size
                                 |
                                 V
     TCOLD Temperature    ------------------------

 ============================
 Thermal Menu Options:
 ============================

 1: Change THOT Temperature
 2: Change TCOLD Temperature
 3: Show current THOT Temperature
 4: Show current TCOLD Temperature
 5: Change Threshold Step Size
 6: Show Limp Home Status
 x: Exit Thermal Menu
 Enter Choice:
```

This menu option allows the user to on the fly change the HOT and COLD thresholds, change the step size for the handler to change the thersholds when the thermal action is taken and also read the current hot and cold threshold and Limp Home status (ACTIVE/INACTIVE).

Note: The current temperature is printed out in the “`p: Print Performance Statistics`” menu option.

7.3 DSP and EVE run time off and on

In order for the application to go into an analytics off low power mode where only the capture and display threads are running the DSP and EVE can be made to go to power domain off state at run time. Once the DSP and EVE are switched off and the application wants to restart the analytics the DSP and EVE has to be re-booted. A demonstration of the analytics off and on scenario has been integrated to the TDA3xx Fast boot use-case (Section 6).

7.3.1 Usecase supported for DSP and EVE off and on

Users can refer to following usecase to tryout demo for DSP and EVE off and on

1 ch ISS capture (OV10640) + Object Detect + Display (LCD 10 inch)

Path in vision_sdk –

\vision_sdk\apps\src\rtos\usecases\fast_boot_iss_capture_isp_simcop_pd_display

This is a special usecase which not listed in Run time Menu. For more details on h/w setup and how to run this usecase please refer VisionSDK_UserGuide_TDA3xx.pdf under vision_sdk\docs folder.

7.3.2 Optimization challenges

There are multiple challenges when the DSP and EVE are switched off and have to be rebooted in between running the usecase:

1. Dividing usecase Data Flow into parts and selectively putting down and bringing up s/w modules
2. Avoiding code loading every time DSP and EVE need to be re-booted.
3. Re-enabling IPC and re-Synchronization of DSP and EVE on re-boot.
4. Seamlessly switching video data when DSP and EVE are switched off and re-booted.
5. Avoid On CPUs kept on from trying to access switched off CPUs.
6. Ensure initialization state of global data structures for the DSP and EVE cores when re-booted.
7. Ensure clean switch off and memory free of DSP and EVE usecase structures to avoid memory leak when DSP and EVE are switch off and then on again.

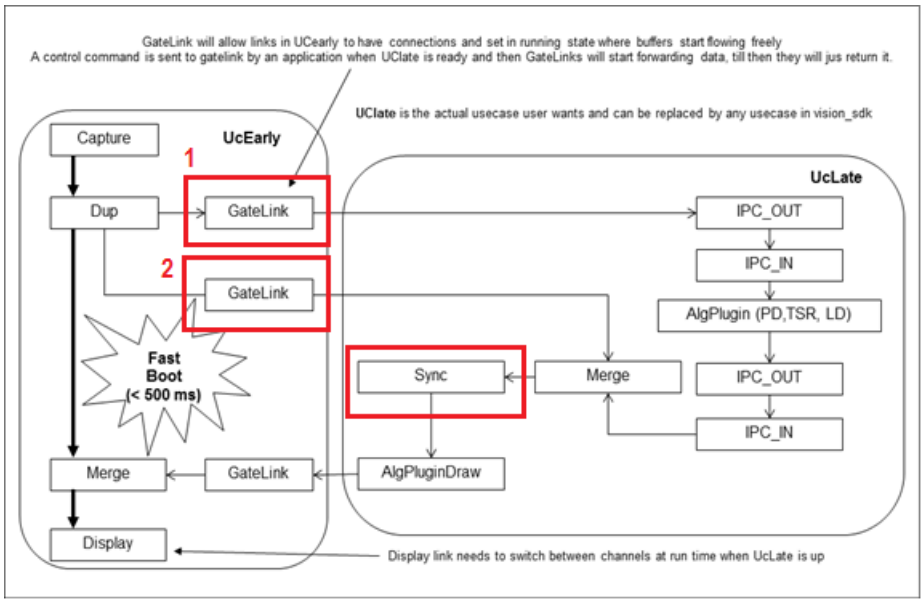
7.3.3 Techniques for DSP and EVE power off and on (Framework Level)

7.3.3.1 Usage of Gate Link

The Gate Link shown in Section 6.3.1 was used to cleanly separate the data flow between the DSP and EVE operation and the capture and display threads to allow seamless switching between analytics on and analytics off state. The UCLate portion of the application was shut down and brought up when the DSP and EVE CPUs were powered off and re-booted.

The key challenge addressed in repeated switching on and off the gate link was to ensure no buffers are held locked by the chains following the gate link before switching off the DSP and EVE. This was done by reading the bufCount of the GateLink and waiting for all the downstream chains after the GateLinks complete their task before switching off cores. This ensured no stale buffers are held by software components in UClate.

In cases where there are two input gate links to the section being switched off and we are switching the gates in the order 1 followed by 2 as shown in the figure below, care should be taken especially in presence of a sync link which waits for the inputs from both the gate links. It could be possible that the buffer from Gate Link 2 passes through before Gate Link 2 is made off whereas the buffer from Gate Link 1 does not pass through. This leads to a situation where the sync link waits till a timeout value before discarding the buffer which has sneaked in while turning the gates off. A finite value of a timeout must be kept to avoid a deadlock situation where the sync keeps waiting infinitely for the second buffer and the logic to turn off the cores keep waiting for the sneaked in buffer to be free.



Refer the functions:

`chains_fastBoot_switchDspEveOn` and `chains_fastBoot_switchDspEveOff`

in the file

`apps/src/rtos/usecases/fast_boot_iss_capture_isp_simcop_pd_display/chains_fastBoot_dspEvePowerDown.c`

for the exact implementation to switch off the Gates, wait for the buffers to get freed and then stop and delete the late usecase.

7.3.3.2 *Handling CPU IPC*

While powering down EVE and DSP, the application must ensure the IPC is cleanly exited to allow re-attaching the IPC when the DSP and EVE are brought up. The first step to do when the Gate links have been switched off and the late usecase components are stopped and deleted is to ensure the DSP and EVE application exit requests are placed. While exiting the DSP and EVE core application software the `System_ipcDetach();` should be called which allows the IPC to detach cleanly between DSP ←→ EVE, DSP→IPU and EVE→IPU. For the implementation of this function refer `src/rtos/links_common/system/system_ipc_notify.c`. The method to perform Ipc_detach is to perform the detach going from a higher value of PROC_ID to a lower value.

Once the DSP and EVE applications have exited the IPU can detach the IPU→DSP and IPU→EVE IPC using the following sequence:

```
cookie = Hwi_disable();
System_ipcNotifyDeInit();
System_ipcDetach();
System_ipcStop();
Hwi_restore(cookie);
```

This sequence is shown in the function `Utils_bootPowerDownSlaves` in the file `src/rtos/utils_common/src/tda3xx/utils_boot_slaves.c`

7.3.3.3 *Turning off and turning on Cores*

The DSP and EVE CPUs are switched off and on using the PMLIB System Config APIs. A sample sequence for DSP1 being switched off is as shown below:

```
if(System_isProcEnabled(SYSTEM_PROC_DSP1))
{
  do
  {
      pmlibSysConfigPowerStateParams_t inputTableDsp1[] =
      {{PMHAL_PRCM_MOD_DSP1,        PMLIB_SYS_CONFIG_DISABLED}};
      status = PMLIBSysConfigSetPowerState(inputTableDsp1, (UInt32) 1,
                                           PM_TIMEOUT_INFINITE,
                                           NULL);
      status = PMLIBSysConfigGetPowerState(PMHAL_PRCM_MOD_DSP1,
                                           &currentState, NULL);
  } while ((PM_SUCCESS == status) &&
           (PMLIB_SYS_CONFIG_DISABLED != currentState));
  if (PM_SUCCESS != status)
  {
     Vps_printf(" UTILS: BOOT SLAVES: Powering down DSP failed!!\n");
  }
}
```

7.3.3.4 *Turning off commands from On CPUs to Off CPUs*

As an example to not allow the IPU to not communicate with the DSP and EVE while they are off the CPU load calculations are also switched off on the DSP and EVE side while they are off. This is specifically a key care about for EVE as the EVE subsystem mailbox is used for communication between IPU and EVE. If the EVE subsystem is off the mailbox inside the EVE subsystem is not accessible and can lead to a potential IPU crash if the IPU tried to access the register space while trying to communicate with EVE.

The function `chains_fastBoot_stopDspEveLoadCalculation` was added to this effect.

7.3.3.5 *Compiler Options for DSP and EVE code*

When the DSP and EVE codes are not re-loaded from QSPI during re-boot there are some key care abouts to allow the DSP and EVE code to re-boot correctly. The assumption made here is that all the data structures stored in the DSP and EVE subsystem internal memories are scratch and do not require any reloading when the DSP and EVE are re-booted.

Note: In the circumstance that the DSP and EVE have some important data in the internal memories that cannot be considered scratch, a save of these memories should be performed before switching off these cores and the contents should be restored once the subsystem is brought up.

Global variables and structures often are initialized to a certain value and can be changed during the program execution. When the DSP and EVE are powered off stale

state of the global variables are left behind which can cause undesired code behavior when the DSP and EVE codes are re-started. To ensure the global variables/structures are re-initialized to their desired state, the following compiler options were modified.

DSP Linker Flags: `build/rtos/makerules/rules_66.mk`

LNKFLAGS_INTERNAL_COMMON = --reread_libs --warn_sections -q -e=_c_int00 --silicon_version=6600 **--rom_model --zero_init=off**

EVE Linker Flags: `build/rtos/makerules/rules_arp32.mk`

LNKFLAGS_INTERNAL_COMMON = --warn_sections -q -e=_c_int00 --silicon_version=arp32 -c -x **--zero_init=off --rom_model**

Specifically for DSP the --dynamic flag was removed to allow creating cinit tables which hold the initialization .data section values and copy them to the actual .data section variable addresses before reaching the application main function. For more details refer the TMS320C6000 Assembly Language Tools User's Guide Literature Number: SPRU186W.

7.4 Reading Power State and Clock Frequency of the system

The power state of the modules and clock frequency of the clocks in the system can be read using the PM APIs. The file `src/rtos/utils_common/src/utils_prcm_stats.c` gives the functions which read the PRCM status using PM STW APIs.

The functions help print the following information regarding PRCM and temperature:

1. DPLL Configuration and Status : `Utils_prcmPrintAllDpllValues()`
2. Voltage Values of different voltage rails: `Utils_prcmPrintAllVoltageValues()`
3. Temperature of different voltage rails: `Utils_prcmPrintAllVDTempValues()`
4. Parsed power state of each module in the system: `Utils_prcmPrintAllModuleState()`
5. CPU clock frequencies : `Utils_prcmPrintAllCPUFrequency()`
6. Peripheral Clock Frequencies: `Utils_prcmPrintAllPeripheralsFrequency()`
7. PRCM Register Dump: `Utils_prcmDumpRegisterData()`

8 Memory Allocation

This section describes the different methods by which memory is allocated in the Vision SDK framework. The Vision SDK framework also support static memory allocation. This section also describes how users can configure their system for static memory allocation.

Memory in Vision SDK framework is allocated for the following purposes

Purpose	Region in memory used for allocation	Type of allocation (Dynamic, Static)
External Buffer memory for storing algorithms results and/or HW engine results	SR1_FRAME_BUFFER_MEM	Dynamic (heap based) and/or Static
Internal Buffer memory for storing algorithms results and/or HW engine results	OCMC_RAM	Dynamic (heap based) and/or Static
Notify Shared region – ONLY used during Notify setup (IPC_Start()), not used later	SR0	Dynamic (heap based)
Temporary scratch memory in internal memory for algorithms results	DMEM in EVE L2SRAM in DSP	Dynamic (non-heap, linear allocation)
Shared memory for remote core print logs	REMOTE_LOG_MEM	Static
Shared memory for link statistics	LINK_STATS_MEM	Static
Shared memory for inter processor communication	SYSTEM_IPC_SHM_MEM	Static
VPDMA descriptor memory for VIP, VPE HW engines	HDVPSS_DESC_MEM	Static
CPU specific memory for BIOS objects like semaphores, tasks, interrupts, clocks	CPU specific data section	Static

The subsequent section provide more details on each type of memory allocation

In the below description,

<soc> = tda2xx, tda2ex, tda3xx

<ddr_size> = 64mb, 256mb, 512mb, 1024mb

<os_type> = bios, linux

8.1 External Buffer Memory Allocation

8.1.1 Location where memory is specified

- The memory region used for external buffer memory allocation is specified via the below file
    - File: vision_sdk\apps\build\<soc>\mem_segment_definition_<os_type>.xs for TDA2XX/TDA2EX
    - vision_sdk\apps\build\<soc>\mem_segment_definition_<ddr_size>.xs for TDA3X
    - Variable SR1_FRAME_BUFFER_SIZE
- The heap is defined only on IPU1-0 CPU, all other CPUs send a command to IPU1-0 to allocate memory. This is done internally inside the Utils_memAlloc APIs.

8.1.2 API to allocate and free memory

- Below APIs are used to allocate and free memory
    - FILE: \vision_sdk\links_fw\src\rtos\utils_common\include\utils_mem.h
    - API:
        - Utils_memAlloc() with heapId as UTILS_HEAPID_DDR_CACHED_SR
        - Utils_memFree() with heapId as UTILS_HEAPID_DDR_CACHED_SR
        - Utils_memGetHeapStats() with heapId as UTILS_HEAPID_DDR_CACHED_SR
- Other APIs from this file are not recommended to be used by users and are used internally by the framework

8.1.3 Changing the size of memory region

- Modify region size in .xs file mentioned in section 8.1.1
- Modify heap segment size #define in .h file mentioned in section 8.1.1

8.1.4 Using static memory allocation

- When a system wants to use static memory allocation and avoid this heap, it should set the size of this heap segment as 0 by modifying the #define in .h file mentioned in section 8.1.1
- Define static memory objects (arrays, data structures) in IPU1-0 use-case file. Make sure the objects are placed in data section ".bss:heapMemDDR" via #pragma as shown in section 8.1.1
- The links which support static memory allocation allow passing of memory region pointers from use-case file via System_LinkMemAllocInfo data structure
- When creating a link from a use-case, user should now pass memory pointer allocated statically from use-case file. This prevents the link for allocating memory internally. Thus dynamic memory allocation is avoided
    - See capture link "captureLink.h" for example
    - See use-case "\vision_sdk\apps\src\rtos\usecases\vip_single_cam_view" for sample usage of passing user pointer to a link
    - NOTE: In the use-case the memory allocation is still done using Utils_memAlloc APIs. In a fully static memory system, this API wont be used by the user.

- The links assert is the memory segment size passed to it is smaller than what is needs. In this case, it also reports the size required by the link.
- When creating user specific AlgPlugins same mechanism should be used, i.e algorithm plugin should take memory pointer passed from use-case file rather than allocating memory internally. See "Capture" link for example
- The below links support passing of user pointer from the use-case
  - Capture
  - IssCapture
  - IssM2mIsp
  - IssM2mSimcop
  - VPE
  - Algorithm Plugin: IssAewb
  - Algorithm Plugin: CRC
- Other links and algorithm plugins are not modified to take user memory pointer as input.

8.2 Internal Buffer Memory Allocation

8.2.1 Location where memory is specified

- The memory region used for buffer memory allocation is specified via the below file
  - File: vision_sdk\apps\build\<soc>\mem_segment_definition_<os_type>.xs for TDA2XX and TDA2EX
  - vision_sdk\apps\build\<soc>\mem_segment_definition_<ddr_size>.xs for TDA3XX
  - Variable OCMC1_SIZE
- The heap from which memory is allocated is defined in file
  - FILE: vision_sdk\src\utils_common\src\utils_mem_ipu1_0.c
    - #pragma DATA_SECTION(gUtils_memHeapOCMC, ".bss:heapMemOCMC")
  - FILE: vision_sdk\src\utils_common\include\utils_mem_cfg.h
    - #define UTILS_MEM_HEAP_OCMC_SIZE
  - This heap is placed in "OCMC" section via the IPU1-0 cfg file
    - FILE: vision_sdk\src\main_app\<soc>\ipu1_0\Ipu1_0.cfg
      - Program.sectMap[".bss:heapMemOCMC"] = "OCMC_RAM";
- The heap is defined only on IPU1-0 CPU, all other CPUs send a command to IPU1-0 to allocate memory. This is done internally inside the Utils_memAlloc APIs.

8.2.2 API to allocate and free memory

- Below APIs are used to allocate and free memory
  - FILE: \vision_sdk\links_fw\src\rtos\utils_common\include\utils_mem.h
  - API:

- Utils_memAlloc() with heapId as UTILS_HEAPID_OCMC_SR
- Utils_memFree() with heapId as UTILS_HEAPID_OCMC_SR
- Utils_memGetHeapStats() with heapId as UTILS_HEAPID_OCMC_SR

- Other APIs from this file are not recommended to be used by users and are used internally by the framework

8.2.3 Changing the size of memory region

- Modify region size in .xs file mentioned in section 8.1.1
- Modify heap segment size #define in .h file mentioned in section 8.1.1

8.2.4 Using static memory allocation

- When a system wants to use static memory allocation and avoid this heap, it should set the size of this heap segment as 0 by modifying the #define in .h file mentioned in section 8.2.1
- Define static memory objects (arrays, data structures) in IPU1-0 use-case file. Make sure the objects are placed in data section ".bss:heapMemOCMC" via #pragma as shown in section 8.2.1
- Currently none of the links and algorithm plugins implemented by TI use OCMC memory, hence if user wants to statically allocate from OCMC memory they should pass the pointer to the OCMC memory as "create" parameters to the respective algorithm plugin that they have implemented.

8.3 IPC Notify Memory

8.3.1 Location where memory is specified

- The memory used by Notify module of IPC package is specified in below files
  - FILE: vision_sdk\apps\build\<soc>\mem_segment_definition_<os_type>.xs for TDA2XX and TDA2EX
  - vision_sdk\apps\build\<soc>\mem_segment_definition_<ddr_size>.xs for TDA3XX
  - Variable SR0_SIZE
- IPC and IPC Notify configuration is done in below file
  - FILE: vision_sdk\src\main_app\<soc>\cfg\IPC_common.cfg

8.3.2 API to allocate and free memory

- Users are recommended to not allocate memory from this region since this is dedicated for IPC Notify module

8.3.3 Changing the size of memory region

- Modify region size in .xs file mentioned in section 8.3.1

8.3.4 Using static memory allocation

- Currently SR0 is MUST for using IPC and IPC Notify module. It does some dynamic allocation from SR0 during Notify setup. This cannot be avoided.

8.4 Temporary Scratch memory for algorithms

8.4.1 Location where memory is specified

- This memory region is applicable only for DSP and EVE
- The memory region from which memory is allocated is defined in file
    - FILE: \vision_sdk\links_fw\src\rtos\utils_common\src\utils_mem.c
        - #pragma DATA_SECTION(gUtils_memHeapL2, ".bss:heapMemL2")
    - FILE: \vision_sdk\links_fw\src\rtos\utils_common\include\utils_mem_cfg.h
        - #define UTILS_MEM_HEAP_L2_SIZE
    - For DSP, this memory is placed in "L2SRAM" section via the DSP cfg file
        - FILE: \vision_sdk\links_fw\src\rtos\bios_app_common\<soc>\dsp<n>\Dsp<n>.cfg
            - Program.sectMap[".bss:heapMemL2"] = "L2SRAM";
    - For EVE, this memory is placed in "DMEM" section via the EVE cfg file
        - FILE: \vision_sdk\links_fw\src\rtos\bios_app_common\<soc>\eve<n>\Eve<n>.cfg
            - Program.sectMap[".bss:heapMemL2"] = "DMEM";

8.4.2 API to allocate and free memory

- Below APIs are used to allocate and free memory
    - FILE: \vision_sdk\links_fw\src\rtos\utils_common\include\utils_mem.h
    - API:
        - Utils_memAlloc() with heapId as UTILS_HEAPID_L2_LOCAL
        - Utils_memFree() with heapId as UTILS_HEAPID_L2_LOCAL
        - Utils_memGetHeapStats() with heapId as UTILS_HEAPID_L2_LOCAL
- Other APIs from this file are not recommended to be used by users and are used internally by the framework

8.4.3 Changing the size of memory region

- Modify heap segment size #define in .h file mentioned in section 8.4.1

8.4.4 Using static memory allocation

- This memory region does not contain a heap data structure.
- Memory is allocated using a linear allocator, where in a "offset" is incremented each time Utils_memAlloc() is called. Utils_memFree() resets the "offset" to zero.
- Each algorithm would typically reset the "offset" and use alloc APIs to get "offsets" into the memory region
- Multiple algorithms would share the same memory region, hence all data in this region should be treated as "scratch" or temporary by algorithms. Contents of the memory would be lost when control switchs from one algorithms to another
- In Vision SDK control would switch from one algorithm to another at buffer/frame processing boundary.

8.5 Memory for Remote Log, Link Statistics, Interprocessor communication, VPDMA Descriptors

8.5.1 Location where memory is specified

- The memory region used for these sections are specified via the below file
    - File: vision_sdk\apps\build\<soc>\mem_segment_definition_<os_type>.xs for TDA2XX and TDA2EX
    - vision_sdk\apps\build\<soc>\mem_segment_definition_<ddr_size>.xs for TDA3XX
    - Variable "REMOTE_LOG_SIZE" for Remote Log memory
    - Variable "SYSTEM_IPC_SHM_SIZE" for interprocessor communication
    - Variable "LINK_STATS_SIZE" for Link Statistics
    - Variable "HDVPSS_DESC_SIZE" for VPDMA descriptors
- The data structure definition is done in below files,
    - For Remote log memory
        - FILE: \vision_sdk\links_fw\src\rtos\utils_common\src\utils_remote_log_server.c
    - For Interprocessor communication
        - FILE: \vision_sdk\links_fw\src\rtos\utils_common\system\system_ipc.c
    - For Link Statistics
        - FILE: \vision_sdk\links_fw\src\rtos\utils_common\src\utils_link_stats_collector.c
    - For VPDMA descriptors
        - Various files in PDK

8.5.2 API to allocate and free memory

Not applicable, since this is statically allocated memory and users cannot allocate memory from this region

8.5.3 Changing the size of memory region

- Modify #define in .c file mentioned in section 8.5.1

8.5.4 Using static memory allocation

- All memory in this region is static memory and cannot be dynamic

8.6 Memory for BIOS Objects

8.6.1 Location where memory is specified

- The memory region used for these sections are specified via the below file
    - File: \ti_components\drivers\pdk\packages\ti\drv\vps\src\osal\tirtos\bsp_osal.c

- File: \ti_components\drivers\pdk\packages\ti\drv\vps\include\osal\bsp_osalCfg.h

8.6.2 API to allocate and free memory

- See APIs defined in below file
  - FILE: \ti_components\drivers\pdk\packages\ti\drv\vps\include\osal\bsp_osal.h

8.6.3 Changing the size of memory region

- Modify #define in bspOsalCfg.h file mentioned in section 8.6.1

8.6.4 Using static memory allocation

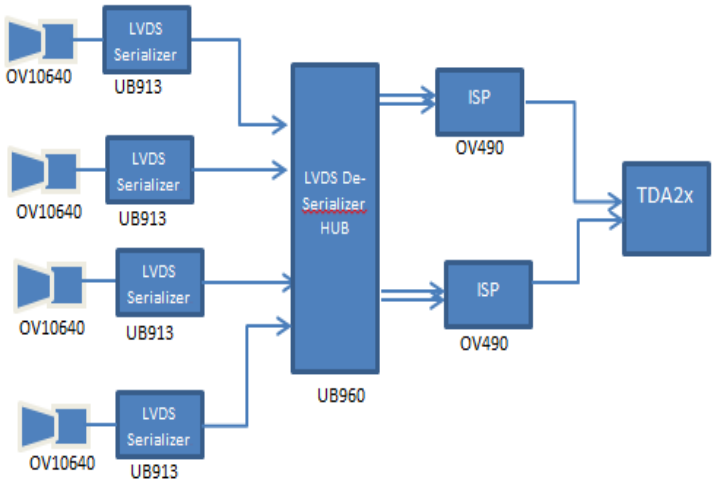
- All memory in this region is static memory and cannot be dynamic

8.7 Known issues and limitations for Static memory allocation system

Below are the known issues and limitations in configuring the Vision SDK system in static memory allocation mode

Limitation	Cause	Solution
BspOsal_taskCreate() results in two "alloc" for "hook function" object alloc	If any "hook function" is registered to task module, the memory for hook function object gets allocated using the default heap. By default in Vision SDK, task level load measurement is registered as a hook function	Disable task, Hwi, Swi load measurement by making below change FILE: \vision_sdk\links_fw\src\rtos\bios_app_common\<soc>\cfg\BIOS_common.cfg Load.swiEnabled = false; Load.hwiEnabled = false; Load.taskEnabled = false; Note, in this case, task specific CPU load is invalid. However total CPU load measurement is still valid.
Modules in vision_sdk\apps would use "alloc"	Modules in vision_sdk\apps are not modified to remove dynamic memory alloc since this example code and is not expected to be used as-is in customer products	If customer plans to use code from this folder in end product then they need to modify to use static memory allocation
NDK uses dynamic memory "alloc"'s	NDK network stack does not support static memory allocation	NDK can be disabled by making NDK_PROC_TO_USE=none in Rules.make
IVA links used for video encode/decode use dynamic memory alloc	IVA links do not support static memory allocation	IVA can be disabled by making IVAHD_INCLUDE=no in Rules.make
Ipc_start() results in memory "alloc"	IPC module does not support static memory allocation	None. All memory allocation happens before System_main() of Vision SDK framework is called. Later during run-time no alloc happens

9 Surround view use-case using TIDA00455/OV490

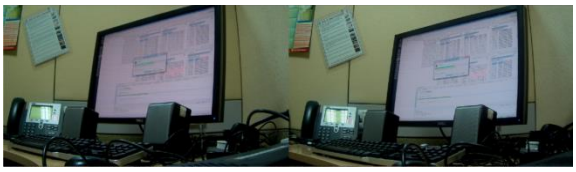


This use-case is used to demonstrate a prototype low-cost surround view platform using OV10640 cameras UB960 de-serializer hub chip and OV490 ISP. This platform reduces the overall system BOM by replacing the four de-serializers with one de-serializer hub chip and using two ISPs (OV490) instead of four.

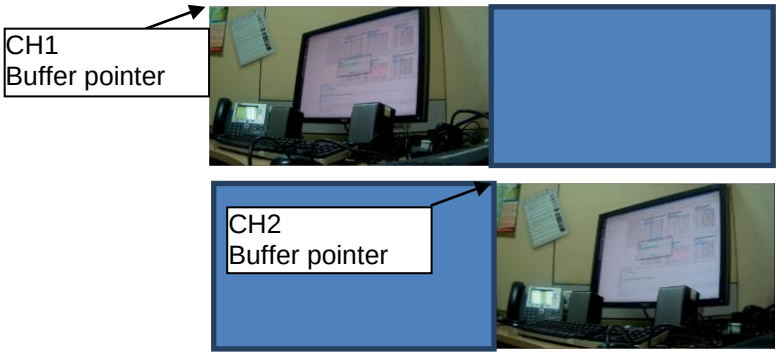
Each OV10640 sends raw video of resolutions up to 1280x880. UB960 sends two CSI2 streams – each with two virtual channels of video data – to the two OV490s. Each OV490 then stitches the two video inputs horizontally and sends out 2560x880 resolution to TDA2x.

Video driver in VisionSDK captures the 2560x880 video. SPLIT link in VisionSDK provides splits one channel of 2560x880 into two channels of 1280x880 – this is done using address manipulation only without using any additional video buffer memory.

Capture driver output – single channel 2560x880 (pitch = 2560)



Split link output – two channels 1280x880 (pitch = 2560)



Actual use-case uses only 2560x720 resolution as required by the surround view algorithm.

10 Usage of Windowed Watchdog Timer feature in TDA3x

TDA3x SoC consists of five RTI (Real Time Interrupt) modules, viz. RTI1 to RTI5. Each of these provides Windowed Watchdog Timer (WWDT) functionality. Please refer to TDA3x TRM for detailed description of this module.

VisionSDK provides an example usage of RTI modules to monitor operation of different cores in the system. This is implemented in all single channel frame-copy use-cases (options 2/3/4 within the single channel use-case section).

RTI-Link is implemented in the `\vision_sdk\apps\src\rtos\modules\rti\src` folder. The same code is used on all cores – core specific variations are handled run-time by using the `System_getSelfProcId()` framework API.

10.1 RTI link – Summary

The basic idea of this link is to associate each core with a different RTI module. Each core must service the associated WWDT in a timely fashion. If this is not done, the RTI module can generate an interrupt or reset the SoC – this is user configurable. In the VisionSDK implementation of RTI-link, RTI modules are associated with different cores as follows:

1. RTI2 -> IPU1-Core0
2. RTI3 -> DSP1
3. RTI4 -> DSP2
4. RTI5 -> EVE

This mapping is done using the `gRtiLink_obj.procMap[]` variable.

The WWDT period can be configured only once after a cold-boot. RTI1 WWDT gets used during the boot-up process and is configured with a period of ~3min. Since this is not changeable, VisionSDK does not use the RTI1 module in the RTI-link. If the 3min period is acceptable for some use-case, users can make use of this RTI1 module as well.

10.2 WWDT expiry handling

If WWDT is not processed on time by DSP1/DSP2/EVE, the corresponding RTI modules are configured to generate an interrupt which is routed via the interrupt crossbar to all cores. This is treated as a core expiry event and IPU1-Core0 will reset the corresponding core in this case. Other cores detect this event and stop sending messages to the expired core to prevent a system hang. This mechanism allows users to execute use-cases which do not use this expired core safely.

In case WWDT corresponding to IPU1-Core0 expires, the corresponding RTI module will generate a SoC Warm Reset.

Refer to the function `rti_registerInterrupts()` for interrupt routing and `rti_getCoreRtiReaction()` RTI reaction configuration.

10.3 RTI link – Task description

`rti_wwdtProcess()` is the task associated with this link. The task is kept at lowest priority to minimize interference to the system.

`rti_setup()` sets up each of RTI module. The default configuration for each WWDT is a period of 4 seconds (as set in `rti_getCoreRtiTimeoutVal()`) and a window size of 50% (as set in `rti_getCoreRtiWindowSize()`)

The link is enabled or diasabled via the `System_rtiEnableAll()` or `System_rtiDisableAll()` API from the use-case. This function causes the link to start executing the rti_service() function. For a WWDT configuration with period of 4 seconds and window size of 50% (2 seconds), this function will sleep for start the WWDT, sleep for 2 seconds, wake-up and service the WWDT. It will continue to do these actions in a loop until a `System_rtiDisableAll()` function call occurs.

When WWDT servicing and monitoring is disabled via `System_rtiDisableAll()`, WWDT expiry events continue to occur but are safely ignored by the framework.

10.4 WWDT configuration and reconfiguration

We support WWDT monitoring on a use-case basis. This requires multiple reconfigurations of the RTI modules. The WWDT counter cannot be stopped once it is started. To ensure correct reconfiguration, following sequence is followed.

a. One-time configuration of clock source and timer period
b. Set window size to required value
c. Start WWDT counter
d. Service WWDT within service window until WWDT servicing is requested to be stopped by use-case using the `System_rtiDisableAll()`
e. Change Window size to 50%
f. Wait for service window and service the WWDT. Change reaction type to interrupt immediately (outside service window) to ensure the change takes effect immediately.
g. Sleep for 1ms and change window size to 100% to allow reconfiguration at any time in the future
h. When a new `System_rtiEnableAll()` function call is made, change window size to the new requested window size.
i. Clear older expiry events generated when WWDT was not being monitored and service the WWDT to ensure the new window size takes effect.
j. Update the reaction type for WWDT expiry as required.
k. Loop back to step **d**

Refer to `rti_setup()` and `rti_service()` for more details.

11 Usage of filesystem with Vision SDK

FAT Filesystem can be used with MMC/SD as storage media with Vision SDK.

This section describes the integration of FAT filesystem with Vision SDK

11.1 Features

- FAT FS implementation taken from http://elm-chan.org/fsw/ff/00index_e.html
- Integrated with MMC/SD as storage media
- Supports EDMA mode of operation of MMCSD. EDMA operates in interrupt mode.
- Works on IPU1-0
- Tested on TDA3x EVM and TDA2x EVM
- Thread safe APIs when used via "\vision_sdk\links_fw\src\rtos\utils_common\include\file_api.h"

11.2 Known Limitations

NOTE: See "\ti_components\drivers\pdk\packages\ti\drv\stw_lld\fatlib\fatfs\ffconf.h" for config options used to configure the FAT filesystem.

- Works on one partition only
- Long filenames, Unicode char support not enabled
- Run-time Card removal / insertion not supported. Card MUST be inserted before starting the application
- Card formatting not supported
- Date / time not supported
- NDK/NSP is disabled when FAT FS is enabled. FAT FS is disabled when NDK/NSP is enabled. See Rules.make to enable NDK or FATFS. Do "gmake config" to confirm is FAT FS is enabled with current options specified in Rules.make

11.3 Integration Details

FAT FS integration involves below files

File / Folder	Purpose
\ti_components\drivers\pdk\packages\ti\drv\stw_lld\fatlib\fatfs	FAT Filesystem code, taken from http://elm-chan.org/fsw/ff/00index_e.html
\ti_components\drivers\pdk\packages\ti\drv\stw_lld\fatlib	MMC SD driver used to integrate with FAT FS
\ti_components\drivers\pdk\packages\ti\drv\stw_lld\fatlib\fatlib_edma	EDMA integration with MMCSD driver – ONLY used when Starterware ONLY examples are used. With Vision SDK the EDMA integration is in vision_sdk folder
\vision_sdk\links_fw\src\rtos\utils_common\include\file_api.h	API user should use to do File IO
\vision_sdk\links_fw\src\rtos\utils_common\src\file_api.c	File system, MMC SD init, board/pinmux, clock init for MMCSD and filesystem integration with Vision SDK
\vision_sdk\links_fw\src\rtos\utils_	EDMA integration with Vision SDK

File / Folder	Purpose
common\src\file_api_dma.c	

11.4 Using FAT filesystem

- Make sure FAT FS is enabled in \apps\configs\$(MAKECONFIG)\cfg.mk via FATFS_PROC_TO_USE variable. By default FATFS_PROC_TO_USE is set to ipu1_0
  - When NDK is enabled via NDK_PROC_TO_USE, FATFS gets disabled. \apps\configs\$(MAKECONFIG)\cfg.mk over rides the variable value
  - To confirm FAT FS is enabled, do "gmake config" and check if FATFS_PROC_TO_USE reflects the value set in \apps\configs\$(MAKECONFIG)\cfg.mk
- Make sure SD card is inserted in the EVM before starting the application
  - Currently FAT FS with MMCSD is tested on TDA3x EVM and TDA2x EVM only
- FAT FS and MMCSD is initialized by calling File_init() in ChainsCommon_Init() [\vision_sdk\apps\src\rtos\usecases\common\chains_common.c]
  - This called by default based on FATFS_PROC_TO_USE flag set in \apps\configs\$(MAKECONFIG)\cfg.mk
- Use File IO APIs defined in file_api.h [\vision_sdk\links_fw\src\rtos\utils_common\include] to start using the filesystem in the SD card
  - See Utils_fileReadFile() and Utils_fileWriteFile() in file_api.c [\vision_sdk\links_fw\src\rtos\utils_common\src] for example usage of file IO APIs
- Additional APIs of FAT FS not defined in file_api.h can be used via APIs defined in \ti_components\drivers\pdk\packages\ti\drv\stw_lld\fatlib\fatfs\ff.h
  - Note, however that these APIs wont be thread safe, so user to make sure these APIs are called from a single thread/task
  - For detailed API documentation of these APIs refer to http://elm-chan.org/fsw/ff/00index_e.html

12 Frequently Asked Questions

Q. How will I Know, which link can run on which processor core?
A. There are certain links which can run on any core. These are called as common links and they are present in the folder \vision_sdk\links_fw\src\rtos\links_common. There are certain links which can run only on a particular processor core. These are present in core specific folders. Ex: \vision_sdk\links_fw\src\rtos\links_ipu consists of links which can run on IPU only.
Q. How will I know if the steps during use case creation or execution happened successfully?
A. All the system APIs used will return status value. By examining the return status value, we can understand if a particular step happened successfully or not.
Q. Why do we have single vs multi mailbox in Link Task? And, why do we enable multimbx on DSP and EVE but not IPU and A15?
DSP and EVE mainly run algorithms. The algorithms don't really preempt until a frame processing is complete. On M4 and A15 where we use HWs like say VPE or Ethernet, while HW processes a frame, task can switch to some other work. So on DSP and EVE a single task is enough to handle all algos. MultiMbx is to use a single underlying task but multiple logical links on top. On EVE particularly we want algo stack to be in DMEM so using a single task to run multiple algos becomes very important since we have very limited DMEM.

13 Revision History

Version	Date	Revision History
0.10	02 Oct 2013	First Draft
0.11	09 Oct 2013	Reviewed and modified
0.50	30 Oct 2013	Merged alg link and use case doc into one doc
1.00	26 Feb 2014	Merged Link development guide
1.01	4 Mar 2015	Updated to add details about use-case gen tool
2.0	6 July 2015	Updated for Vision SDK v2.7. Added sections on fast boot in TDA3x, power optimization, memory allocation
2.1	14 Oct 2015	Updated based on Vision SDK v2.8 features
2.9	4 April 2016	Updates based on Vision SDK v2.9 features
3.0	05th July 2017	Updated for Vision SDK release 3.0

««« § »»»