CDK Plug-in Framework

User Guide

v0.8 / August 2018

Intel® Movidius™
2200 Mission College Blvd
M/S SC11-201
Santa Clara, CA 95054
http://www.movidius.com/

Table of Contents

1 Introduction

1.1 About this document

The purpose of this document is to show Myriad 2 users how to develop camera and sensor driven applications using the Myriad 2 Camera Development Kit (CDK) plug-in framework.

The framework allows multiple components to be integrated together to create ISP and Computer Vision processing applications.

These components perform frame-oriented processing of data which may be sourced from various types of cameras. Components may implement ISP and/or Computer Vision related functionality.

1.2 Scope

This document describes the steps required to create user defined plug-ins and build applications using the CDK Plug-In Framework (PIFW).

The framework is designed to operate in conjunction with a Camera Framework application (incorporating 3A algorithms, focus control etc.) to provide a complete camera solution.

The framework runs on MA2100 and MA2150 silicon.

1.3 Target Audience

This document is targeted at users of the CDK framework wishing to build camera or image processing solutions running on the Movidius Myriad 2 silicon.

The audience of this document includes CDK system integrators, SW Team Leads and SW developers.

1.4 Related Documents and Resources

Documents relating to the CDK can be obtained from http://www.movidius.org. If you do not have access to the documents below, you can request them. Relevant documents include:

1. CDK release package (download from movidius.org)
2. Myriad 2 Platform Databook
3. Myriad 2 Instruction Set Reference Manual
4. Camera Interface Specifications, MIPI Alliance, http://www.mipi.org/specifications/camera-interface#CSI2

1.5 Terms and definitions

Term	Definition
3AFW	3A algorithms Framework (Auto Exposure, Auto White Balance and auto focus)
AP	Applications processor (AKA Host Processor).
API	Application Programmer Interface
CDK	Camera Development Kit
CFA	Color Filter Array
Camera Framework	Main camera control application incorporating 3A functionality, sensor and Autofocus drivers etc. Runs on the **Leon OS** processor.

Term	Definition
Client API	Synonymous with **IPIPE Client API.**
Control Structure	A structure shared between the two RISC processors, used for communication between the Client and the IPIPE. On the Client side, the implementation of the Client API has access to the shared structure, but the IPIPE control application does not access it directly.
CV	Computer Vision
HIF	Host Interface
IPC	Inter-Process Communication
IPIPE	ISP Pipeline framework. An application implemented with IPIPE may incorporate multiple plugin components, such that it may support multiple cameras, and may incorporate Computer Vision and other custom processing.
IPIPE Client	Client application which makes calls to the IPIPE Client API. The **Camera Framework** is such a client application.
IPIPE Client API	API called by the **IPIPE Client Application** to interact with IPIPE.
IPIPE Control Application	Main application running on the **Leon RT** processor, which takes care of plugin instantiation and all **IPIPE** related scheduling and buffer management.
Normal Mode	One of the two primary modes of operation of IPIPE. In Normal mode, a single pipeline processes the sensor output. The Video stream is generated by downsizing the ISP output.
Leon OS	RISC processor (SPARC Leon4) which runs the **Camera Framework** under the **RTEMS** OS environment.
Leon RT	RISC processor (SPARC Leon4) which runs the **IPIPE Control application**, in a bare-metal environment.
Perfect Raw	Data in the same format as the Raw data arriving from the sensor, which has has some processing applied to it, mainly corrections. The image should be free from defective pixel, vignetting, and color shading artifacts.
Plugin	A plugin is a component which plugs into the IPIPE framework to do e.g. ISP or Computer Vision processing.
Sensor Raw	Raw data from a camera sensor which has not undergone any processing on Myriad. May contain defective pixels, and Len Shading Correcting has typically not been applied.
Source	A source of image frames, i.e. a Camera sensor.
Split Mode	One of the two primary modes of operation of IPIPE. In Split mode, there are separate ISP pipelines for Video and Still.
Video Stream	Catchall term for non-Still Image streams. Video Streams typically run at the same frame rate as the sensor, and may subsequently be treated as either Video or Preview streams.
ZSL	Zero Shutter Lag

Table 1: Abbreviations

2 Plug-in Framework Execution Model

2.1 Basic Concepts

The Plug-in Framework (PIFW) allows multiple components to be integrated together to create ISP and Computer Vision processing applications.

These components perform **frame-oriented processing** of data which may be sourced from various types of cameras. Components may implement ISP and/or Computer Vision related functionality.

Applications are constructed by assembling plugins into the application pipelines. Plugins communicate with each other by means of frame pools. The framework provides a mechanism by which frame of data can be transferred among plugins. The framework also provides a mechanism by which the plugins to be used in the application are instantiated and linked together.

Plugins may be implemented using a combination of Leon RT, Shave, SIPP hardware accelerator and DMA controller resources. Plugins may use the SIPP framework to instantiate and control image processing and computer vision pipelines.

A plugin may have zero or more inputs. An input is associated with a frame pool. For each input, the associated frame pool is provided with a plugin-supplied callback, which is called whenever the producer produces a new frame into the frame pool. When the plugin has received the needed frames at its input(s), it typically begins processing, and as a result will eventually produce frame(s) into output frame pools, and/or output **metadata** (non image data) for transmission to the host processor.

Multiple plugins may en-queue frames onto the same **frame pool**. There may be multiple plugins consuming frames from a frame pool. Consumers must register to receive notification that a new frame has been en-queued. Consumers of frames must explicitly release frames they have received, so that the frame can be reused by the producer.

A **source** is associated with a physical camera input. The data arrives physically via either the MipiRX blocks, or via the CIF. Source instances are assigned IDs, which are a set of contiguous integers starting at zero.

A pipeline may be connected directly to a source, such that one of the MipiRX blocks receives data at the start of the pipeline, and frames are not written to DDR prior processing e.g. where there is a single source or multiple sources are in sync. In all other cases, the frames coming from the camera must be buffered in DDR prior to processing. This is because the internal line buffers of the ISP hardware accelerator blocks are limited to 4624 pixels on MA2150, and therefore only frame-granular context switching is supported.

There is also an explicit **trigger** mechanism which may trigger a plugin to perform processing. Plugins which are explicitly triggered typically do not have any inputs associated with frame pools. Instead, the plugin's “trigger” entry point is called. Input frames may be passed to the “trigger” entry point, in addition to a callback which the plugin will call when the processing is complete (which signals that the plugin is ready to accept another call to its “trigger” entry point). This mechanism is typically used to process frames that are located in a ZSL buffer.

A plugin based application is a directed graph of plugins and frame pools. The application links plugins and frame pools together at initialization time.

Multiple consumer plugins may be linked to a frame pool. Also, a plugin may have a producer relationship with more than one frame pool. If a frame pool is linked to more than one producer, those producers must coordinate to ensure that consumer(s) of the frame pool are not overrun.

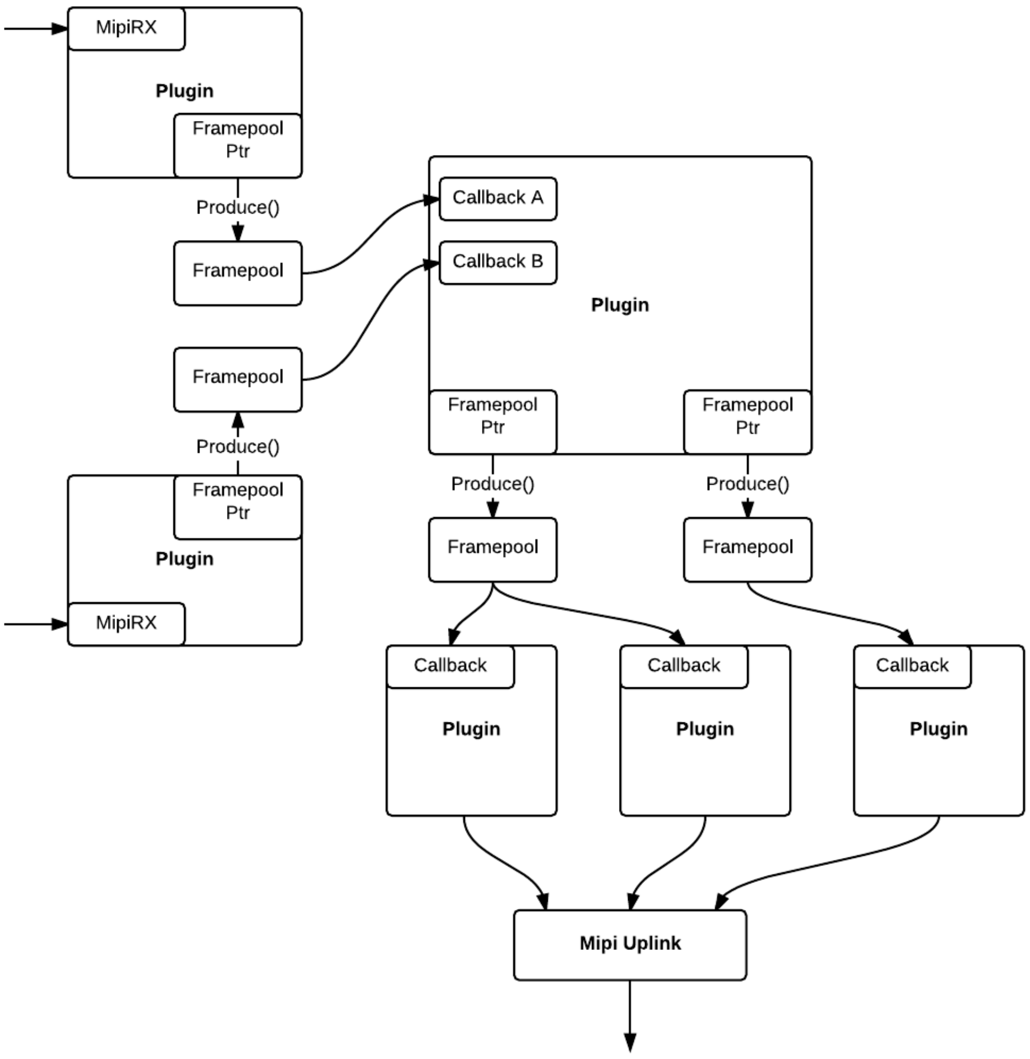

Figure 1: Connection of Plugins and Frame Pools: example app with 6 plugins

Figure 1 shows an example of a Frame Pool with multiple consumers, and of a Plugin consuming frames from multiple frame pools. The application is a directed graph of plugins and frame pools. The application links plugins and frame pools together at initialization time.

Multiple consumer plugins may be linked to a frame pool. Also, a plugin may have a producer relationship with more than one frame pool. If a frame pool is linked to more than one producer, those producers must coordinate to ensure that consumer(s) of the frame pool are not overrun.

An application may have one or more Outputs. An Output can be a Frame Producer or a MetaData Producer. The produced frames or metadata are en-queued to be sent to the Host over the MIPI interface. A plugin may implement 0 or more outputs. An output is normally dependent on a set of sources. If any of the dependent sources are disabled, then the output will not produce any frames/metadata. For example, a plugin that implements stitching of frames coming from multiple cameras to produce a single stream of combined frames (plugin has two inputs and one output) would not produce any output unless both of the camera sources are running.

All outputs may be independently enabled/disabled. A disabled output must not en-queue data for transmission to the Host. A plugin should also endeavor to minimize processing/power consumption insofar as possible, by avoiding unnecessary processing due to outputs being disabled. For example, in an application with two outputs that generates an image stream plus a depth map, power can be saved by disabling the depth map generation if the depth map output is disabled.

3 Plug-in Framework API

3.1 Plug-in Types and Data Structures

3.1.1 PlgTypeS

Type Name	PlgTypeS
Description	A plugin is a passive entity, which normally stays idle until it receives frame(s) from its parent producer(s), or until it receives data from the MipiRX/CIF interfaces. A plugin primary interface to the IPIPE framework and to IPIPE-based applications is via the plg structure.

Field type	Field name	Field description
int32_t (*init)(FramePool *outputPools, int32_t nOutputPools, void *pluginObj);	init	Function pointer to plugin's initialization entry point (see below).
int32_t (*fini)(void *pluginObj);	fini	Function pointer to plugin's de-initialization / cleanup entry point (see below).
PlgStatus	status	Plugin status. Inform running or idle status. After “fini” command will have wait latter for idle status in order to confirm that the plugin was finish execution (this in case of plugin that not allow immediately finish command).
FrameProducedCB*	callbacks	Points to an array of function pointers of type FrameProducedCB. These functions are used to notify the plugin whenever a frame is produced at one of the frame pools from which the plugin consumes frames. A plugin may consume frames from more than one frame pool (i.e. a plugin may have multiple inputs). The number of elements in the array of callbacks is equal to the number of inputs.
int32_t (*trigger)(FrameT *frame, void *params, int32_t (*callback)(int status), void *pluginObj);	trigger	Function pointers to plugin's “trigger” entry point (see below).
int32_t (*resume)(void *pluginObj);	resume	Function pointers to plugin's “resume” entry point function

3.1.2 PlgStatus Enum

Enum Name	PlgStatus
Description	Enumerator – is Plugin idel or running

	Values	Field description
	PLG_STATS_IDLE	= 0
	PLG_STATS_RUNNING	= 1

3.2 Plug-in API

3.2.1 plg.init()

Function	int32_t(*init)(FramePool *outputPools, int32_t nOutputPools, void *pluginObj);		
Description	Called to initialize the plugin.		
	Field type	**Field name**	**Field description**
Parameter 1	FramePool*	outputPools	–
Parameter 2	int32_t	nOutputPools	–
Parameter 3	void*	pluginObj	–
Return value	int32_t	–	–

The plugin initializes itself, and prepares to receive frames via function calls to functions in the plg.callbacks array. A producer may output frames to more than one frame pool. Producers are linked to the frame pools specified via the “outputPools” array pointer. The “nOutputPools” parameter specifies the number of frame pool pointers in the array. When a producer wishes to produce a frame on an output frame pool, it calls FrameMgrAcquireFrame() on the appropriate frame pool before storing data in the frame, then calls FrameMgrProduceFrame() to send the frame to its consumer(s).

The plugin may use Shaves or SIPP hardware resources to perform processing. If the SIPP framework is used, SIPP-related initialization would normally be performed by this entry point.

3.2.2 plg.fini()

Function	int32_t (*fini) (void *pluginObj);		
Description	Frees any resources that were acquired by plg.init(), and resets them to their original state.		
	Field type	**Field name**	**Field description**
Parameter 1	void*	pluginObj	–
Return value	int32_t	–	

3.2.3 plg.trigger()

Function	int32_t (*trigger)(FrameT *frames, void *params, int32_t (*callback)(int32_t status), void *pluginObj);		
Description	Explicitly triggers the plugin to perform frame processing.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	frames	–
Parameter 2	void*	params	–
Parameter 3	int32_t (*callback)(int32_t status)	callback	–
Parameter 4	void*	pluginObj	–
Return value	int32_t	–	

This is typically used for processing frames in a ZSL buffer. Plugins do not need to support the trigger mechanism, in which case this entry point will be NULL in the plg structure. The "frames" argument is a NULL-terminated linked list of frames which are to be processed. A Bayer or Mono ISP pipeline would typically expect only a single frame in the list. In the event that the plugin is an ISP pipeline, the plugin would expect "ispConfig" to point to an ISP configuration structure. The final argument, "callback" will be called by the plugin to indicate that the processing is complete.

The plugin may process the frames in chunks (e.g. tiles or slices). Each time a chunk is processed, it calls the callback with a status of non-zero if there are more chunks to be processed, or zero if the final chunk has just been processed. If there are more chunks to be processed, the plugin suspends processing until its "resume" entry point is called. This allows the application to schedule resources used by the plugin for other purposes, for example if some hardware filters were shared between this plugin and another plugin. An example use case is where a preview or video stream needs to continue to stream in real time, and the plugin processing the realtime pipeline needs to share resources with a plugin which takes a much longer time to process an entire frame.

The plugin owns the frame(s) passed as arguments, until such a time as it calls the callback with a status of zero.

3.2.4 plg.resume()

Function	int32_t (*resume)(void *pluginObj);
Description	This entry point is called to resume multi-chunk processing, in the event that the callback passed to plg.trigger() was called with a non-zero status. Upon resumption of processing, hardware resources potentially need to be re-configured, since another plugin may have used them while processing was suspended.

	Field type	Field name	Field description
Parameter 1	void *	pluginObj	–
Return value	int32_t	–	

4 Frame Manager API

4.1 Frame Manager Overview

The Frame Manager operates on two primary structures: FramePool and FrameT.

Frames have a base class, as defined by the FrameT structure. There is enough information in the base class to allow the frame's contents to be copied, for example, by a DMA engine. To interpret the contents of the frame, the "type" field is used. A superset structure, which includes the FrameT base structure as its first member, contains additional type-specific information which allow the contents of the frame to be interpreted. The "type" field in the base structure determines the type of the superset structure which defines the frame. Frame types are TBD, but will likely be along the lines of `FRAME_TYPE_BASE`, `FRAME_TYPE_RGB_PLANAR` `FRAME_TYPE_YUV_PLANAR`, FRAME_TYPE_YUV_PACKED etc.

The frame manager should be able to run in any OS environment, including bare metal, so all structures are instantiated externally from the frame manager. In general, the Frame Manager is not re-entrant. However, multiple callbacks of type `FrameProducedCB` may be called concurrently, even on the same frame pool. Likewise, there could be multiple concurrent calls to `FrameMgrReleaseFrame()` since it could be called from an interrupt context.

For each frame pool, the application must allocate frame buffers from DDR, and call `FrameMgrCreatePool()` to initialize the frame pool. The size of the frames in the frame pool depends on the output dimensions of each source. This determines the resolution of the frames coming from the sensor, which will in turn determine the size of the frames flowing through each frame pool in the application.

When creating the frame pool, the frame pool is linked to its consumers via the "callbacks" array passed to `FrameMgrCreatePool()`. These callback function pointers can be sourced from the consumer plugin's plg.callback array. The MIPI TX Mux component's function for enqueuing frames for transmission, which has the same prototype, can also be specified.

When the frame pools have been initialized, the app next calls the `init()` entry point of each plugin. Frame pools are linked to producers via the "outputPools" array passed to `plg.init()`.

4.2 Frame Manager Types & Structures

4.2.1 FrameS

NOTE: FrameT is the *typedef* of the FrameS structure. In the following chapters FrameT will be used.

Type Name	FrameS	
Description	–	
Field type	**Field name**	**Field description**
FrameT*	next	Pointer to next FrameT structure in the linked list, or NULL.
uint32_t	type	Type of frame.

Field type	Field name	Field description
uint32_t	refcnt	When consumers are notified that this frame has just been produced by the producer, this field is set to the number of consumers that were notified. Each time a consumer is finished with the frame, it calls FrameMgrReleaseFrame(), which decrements this reference count. If the counter reaches 0, then the frame is marked as free (the notFree flag is set to 0). **NB: The decrement of refcnt, and the test if it has reached 0, must be performed as one atomic operation.**
uint32_t	notFree	FrameMgrAcquireFrame() checks this flag in the frame indexed by FramePool.nextFree. If set, FrameMgrAcquireFrame() returns NULL, otherwise it sets this flag, advances FrameMgrAcquireFrame.nextFree by following the linked list (returning to the start of the list if NULL is encountered) and returns a pointer to the frame.
FramePool*	framePool	Pointer to the FramePool structure which this frame belongs to.
void*	fbPtr[4]	Void * pointers to the planes of frame buffer memory.
uint32_t	stride[4]	Number of bytes from the first byte of line N to the first byte of line N+1, for each plane.
uint32_t	height[4]	Number of lines in each plane.
uint32_t	nPlanes	Number of planes (minimum 1, maximum 4).
uint64_t	timestamp[4]	Time stamp from different events associated with the frame. It is depend by plugins what time to add there.
uint32_t	timestampEvent[4]	Indicates the event associated with the corresponding timestamp in the “timestamp” array; application specific.
uint32_t	timestampNr	Last time stamp index added in frame structure description.
uint32_t	seqNo	Frame sequence number. Each frame pool maintains an incrementing frame counter. Each time a frame is produced, the value of the counter is stored in the frame, and the frame counter is incremented.
void*	appSpecificData	Application/plug-ins specific void* information that needs to be linked with the frames.

4.2.2 FramePoolS

NOTE: FramePool is the *typedef* of the FramePoolS structure. In the following chapters FramePool will be used.

Type Name	FramePoolS	
Description	–	
Field type	**Field name**	**Field description**
uint32_t	nFrames	Number of frames in the frame pool.
FrameT*	nextFree	Pointer to the next frame which will be returned (if free) by `FrameMgrAcquireFrame()`.
FrameT*	frames	Pointer to a NULL terminated linked list of FrameT structures which belong to this frame pool.
FrameProducedCB*	callbacks	Pointer to an array of callbacks of type FrameProducedCB. Callbacks are called to notify consumers that a new frame of data is available to be processed.
uint32_t	nCallbacks	Number of elements in the “callbacks” array.

4.2.3 FrameProducedCB

Type Name	FrameProducedCB	
Description	Callback supplied by a consumer, which is called to notify it that a frame has been produced on the frame pool.	
Field type	**Field name**	**Field description**
FrameProducedCB *	callback	Pointer to the FrameProducedCB type function that will be call at frame produced event.
void*	pluginObj	Plugin that refer to.

4.2.4 FrameProducedCBFunc

Type Name	typedef void (***FrameProducedCBFunc**)(FrameT *frame, **void** *pluginObj);
Description	

Field type	Field name	Field description
FrameT *	frame	Pointer to frame.
void*	pluginObj	Plugin object that it refers to.

4.3 FrameManager API

4.3.1 FrameMgrCreatePool()

Function	int32_t FrameMgrCreatePool(FramePool *pool, FrameT *frames, FrameProducedCB *callbacks, int32_t nCallbacks);		
Description	Create a frame pool.		
	Field type	**Field name**	**Field description**
Parameter 1	FramePool*	pool	pre-allocated FramePool structure.
Parameter 2	FrameT*	frames	Pointer to a NULL terminated linked list of frame descriptor structures.
Parameter 3	FrameProducedCB*	callbacks	an array of callbacks which are to be called to notify consumers whenever a new frame is produced on the frame pool.
Parameter 4	int32_t	nCallbacks	the number callbacks in the array.
Return value	int32_t	A status code.	

The base descriptor structure is generic, so that a DMA engine can be programmed to transfer data to/from the frame, without knowledge of the meaning of the frame contents. All frame descriptors must have their fields populated beforehand, which includes setting pointers to the actual frame buffer memory. The frame manager is not responsible for allocation of the frame buffer memory.

4.3.2 FrameMgrAcquireFrame()

Function	FrameT* FrameMgrAcquireFrame(FramePool *pool);
Description	Called by a producer to acquire a free frame from the frame pool. The producer can store data into the acquired frame if it is successfully acquired. This function is non-blocking. No mechanism is provided to block until a frame

	becomes available, or to receive notification when a frame becomes available.		
	Field type	**Field name**	**Field description**
Parameter 1	FramePool*	pool	–
Return value	FrameT*	**NULL:** thee frame could not be acquired (indicating all frames are currently in use). This should not normally occur. If it does occur, it indicates that consumer(s) are not able to process/consume the frames as quickly as they are being generated. The producer is expected drop a frame if this occurs. Non-NULL: A free frame was successfully acquired and a pointer to the acquired frame is returned.	

4.3.3 FrameMgrProduceFrame()

Function	void FrameMgrProdiuceFrame(FrameT* frame);		
Description	Sends a frame that was previously acquired with FrameMgrAcquireFrame() to its consumers. Consumer notification callbacks associated with the frame pool will be called to notify consumers that a new frame has been produced.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	frame	–
Return value	void	–	

4.3.4 FrameMgrReleaseFrame()

Function	void FrameMgrReleaseFrame(FrameT *frame);		
Description	Called to release a frame back to the frame pool (normally by the consumer), marking it as free. If there are multiple consumers, all consumers must call FrameMgrReleaseFrame() on the frame, before it is released back to the frame pool (a reference count Is used internally. The Reference Counter shall be decremented as an atomic operation, since there could be multiple concurrent calls to this function!).		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT *	frame	–
Return value	void	–	

4.3.5 FrameMgrLockFrame()

Function	FrameT * FrameMgrLockFrame(FramePool *pool, int32_t frameSel, int tsRel);		
Description	Lock a frame for subsequent capture (e.g. for ZSL implementation). A frame within the frame pool is selected for locking. The selected frame shall not be in the acquired state (if the frame has been acquired via FrameMgrAcquireFrame(), but not yet been produced via FrameMgrProduceFrame(), then it is in the acquired state). To ensure that the frame contains valid content, it must have previously undergone a call to FrameMgrProduceFrame(). Once a frame is selected, it is marked as locked, which means it cannot be acquired by a call to FrameMgrAcquireFrame().		
	Field type	**Field name**	**Field description**
Parameter 1	FramePool*	pool	–
Parameter 2	uint32_t	frameSel	–
Parameter 3	int32_t	tsRel	he "tsRel" parameter is analogous to the IC_LOCKZSL_TS_RELATIVE flag. "frameSel" is either a relative timestamp or a frame index, depending on the value of "tsRel". Refer to the description of IC_LOCKZSL_TS_RELATIVE
Return value	FrameT*	selected frame shall be the non-acquired frame in the frame pool which is considered to be the best match in accordance with the values of "frameSel" and "tsRel".	

4.3.6 FrameMgrUnlockFrame()

Function	void FrameMgrUnlockFrame(FrameT* frame);		
Description	Unlocks a frame that was previously locked by FrameMgrLockFrame().		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	frame	_
Return value	Void	_	

4.3.7 FrameMgrIncreaseNrOfConsumer()

Function	void FrameMgrIncreaseNrOfConsumers(FrameT *frame, uint32_t addNr);		
Description	Increase the reference count associated with "frame" by "n". This function is called automatically by the framework when a frame is produced to set the initial reference count equal to the number of consumers, before the consumers are notified of the frame produced event. Each time a consumer calls FrameMgrReleaseFrame() to indicate that it is finished consuming the frame, the reference count is decremented. When the reference count reaches zero, the frame can be re-used by the producer.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	frame	_
Parameter 2	uint32_t	addNr	_
Return value	Void	_	

4.3.8 FrameMgrAndAddTimeStamp()

Function	void FrameMgrAndAddTimeStamp(FrameT *oFrame, icTimestamp timeStamp);		
Description	Add a timestamped event to the specified frame. The event is added to the frame's timestamp history, which may record up to four events.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	oFrame	_
Parameter 2	icTimestamp	timeStamp	_
Return value	Void	_	

4.3.9 FrameMgrAddTimeStampHist()

Function	void FrameMgrCopyTimestamps(FrameT *oFrame, FrameT *iFrame);		
Description	Add a timestamped event to the specified frame. The event is added to the frame's timestamp history, which may record up to four events.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT*	oFrame	–

	Field type	Field name	Field description
Parameter 2	FrameT*	iFrame	–
Return value	Void	–	

5 IPIPE Client-Server

5.1 IPIPE Client API

The Camera application configures and controls the IPIPE via the IPIPE Client API, or simply the Client API. The Client API implementation runs on the OS Leon, along with the Camera Framework, and handles communication and synchronization with IPIPE (which runs on the RT Leon).

5.1.1 Initialization sequence

The camera framework must call icSetup() before any other calls are made to the IPIPE client API.

The Camera Framework and IPIPE need to be in agreement as to how many Sources are present in the system. Therefore, the Camera Framework must be configured in a manner that is compatible with the application/use case that is going to run on LeonRT within the IPIPE framework. However, the Camera Framework dictates the input resolution of each Source.

At initialization time, the Camera Framework informs IPIPE of the maximum possible resolution of each source (by calling `icSetupSource()`). Once the sources are configured, the Camera Framework commits the configuration by calling `icSetupSourcesCommit()`. In response, the IPIPE application will allocate memory buffers of the appropriate size to support the use case at the required resolutions, and prepare the pipelines to run. If this setup is successful (there is enough memory in the system etc.) then the Camera Framework may request that the sources be configured with the rest of the parameters and started. Once the sources are started, the IPIPE application may start enqueuing image and metadata frames for transmission to the Host via the MIPI interface, for any outputs that are not disabled.

5.1.2 Inter-processor communication

Inter-processor communication (via LeonOS and LeonRT) is implemented by means of a global, shared IPIPE Control structure (subsequently referred to as “the Control Structure”). An Event mechanism is used to communicate between the two processors.

5.1.3 Events

The primary inter-processor communication mechanism is Events. Events are enqueued by adding them to a ring buffer. There are two event ring buffers, one in each direction. An event is transmitted by filling in the next free slot in the event ring buffer, and raising an interrupt to notify the other processor. On receiving the interrupt, the other processor will then check the event ring buffer for new events, consuming any new events and marking the ring buffer entries as available for reuse.

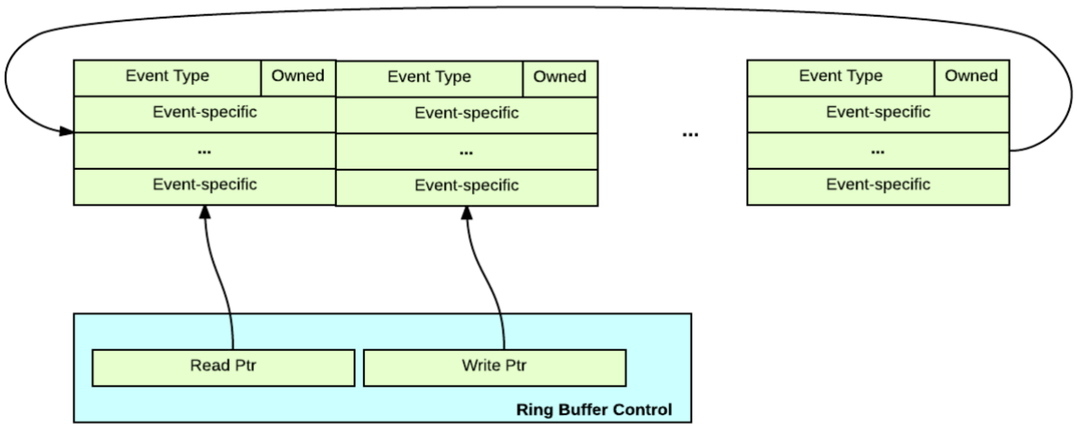

Figure 2: Event Ring Buffer

Each event has an event type, and an ownership bit. If the ownership bit is 0, then the event is available to be used for transmission of a new event, i.e. the event is owned by the transmitter. If the ownership bit is 1, then a previously transmitted event has not yet been consumed by the receiver, i.e., the event is owned by the receiver.

Each ring buffer is controlled by a Write Pointer, and a Read Pointer. **The Write Pointer is solely for the use of the transmitter, and the Read Pointer is solely for the use of the receiver**. The Write Pointer is the index into the ring buffer where the next event is to be enqueued. Both pointers must be set to 0 initially.

To transmit an event, the transmitter uses the Write Pointer to locate the entry in the ring buffer which will store the new event. It first checks the ownership bit. If this bit is 1, it means the ring buffer is full. **This should not happen during normal operation.** Since events are consumed by interrupt handlers, they should be consumed very quickly, and the ring buffer size should be chosen such that the arrival of several events in quick succession can never cause the queue to fill completely. Therefore the check for the ownership bit to be 0 should be an ASSERT(), and failure is considered to be a fatal error.

After checking the ownership bit, the transmitter populates the event structure with the relevant data: the event type plus any event-specific information. Next, the Ownership bit is set to 1, and the interrupt to the receiving processor is raised. **The transmitting processor should not modify the event structure after the ownership bit is set.** The transmitter then advances the ring buffer write pointer, wrapping back to index 0 if necessary.

On the receiver side, upon receipt of the interrupt, the receiver uses the Ring Buffer Read Pointer to locate the entry in the ring buffer where the next incoming event is stored. It first checks the ownership bit to see if there really is a new event in the queue. If the ownership bit is set, it takes whatever action is necessary to handle the event (for example, it could simply set a flag, or copy data from event structure to a private queue for deferred processing by another thread). The receiver should then set the ownership bit to 0, meaning the event has been consumed, and the ring buffer entry can be reused. The receiver will then increment the Ring Buffer Read Pointer, wrapping back to index 0 if necessary. The receiver should loop if the ownership bit at the new index is 1, until all enqueued events in the buffer have been consumed.

NOTE: The SPARC MEMBAR instruction will likely be needed in the implementation, for example, to ensure that the event structure fields are written to memory before the ownership bit is set.

5.1.4 Client-generated events

In response to client API calls by the application, the following events may be sent to IPIPE:

Event	Description
IC_EVENT_TYPE_CONFIG_GLOBAL	Global configuration. This event is sent when icSetup() is called, and specifies the index of the IPIPE application (appID) to be run. It also specifies the range of DDR to be used for the allocation of frame buffers.
IC_EVENT_TYPE_SETUP_SOURCE	Setup camera-specific parameters. Does not take effect until an IC_EVENT_TYPE_SETUP_SOURCE_COMMIT event is issued. API must be in the “configuring” state before sending this event.
IC_EVENT_TYPE_SETUP_SOURCES_COMMIT	Commits the source configurations that were issued since the “configuring” state was entered. An event of type IC_EVENT_TYPE_SETUP_SOURCES_RESULT will be subsequently issued to the client in response.
IC_EVENT_TYPE_CONFIG_SOURCE	Configure camera-specific parameters. API must be in the “configured” state before sending this event.
IC_EVENT_TYPE_CONFIG_SOURCE_DYNAMIC	Configure camera-specific parameters that may be modified even if the camera is started. Does not wait for IC_EVENT_TYPE_CONFIG_SOURCES_COMMIT before taking effect.
IC_EVENT_TYPE_CONFIG_ISP	Configure ISP pipeline parameters
IC_EVENT_TYPE_START_SOURCE	Start a specific camera source
IC_EVENT_TYPE_STOP_SOURCE	Stop a specific camera source
IC_EVENT_TYPE_LOCK_ZSL	Lock a frame in the ZSL buffer for subsequent capturing
IC_EVENT_TYPE_CAPTURE	Process the frame that is locked in the ZSL buffer
IC_EVENT_TYPE_UNLOCK_ZSL	Unlock the frame in the ZSL buffer, if any, that is currently locked.
IC_EVENT_TYPE_ZSL_ADD	Add N frames to the ZSL buffer associated with the specified camera source. Zero buffers may be added, in order to elicit an IC_EVENT_TYPE_ZSL_ADD_RESULT event from IPIPE, without actually adding any frames.
IC_EVENT_TYPE_RESET	Resets the entire IPIPE. Any active sources will be stopped. Any in-progress operations will be halted. Any pending operations will be canceled, and event queues will be flushed. IPIPE will be reset to its initial state.
IC_EVENT_TYPE_TEARD_DOWN	Tears down the entire IPIPE. Any active sources will be stopped. Any in-progress operations will be halted. Any pending operations will be canceled, and event queues will be flushed. IPIPE will be reset to its initial state. Also Leon RT will be stopped.
IC_EVENT_TYPE_MEM_ALLOCATED	#TBD

Event	Description
IC_EVENT_TYPE_OUTPUT_DATA_RECEIVED	Announces server that the respective frame is processed, so the corresponding memory can be reused if necessary. Response to IC_EVENT_TYPE_SEND_OUTPUT_DATA event.

Table 2: Client Events handled by PIPE

5.1.5 IPIPE-generated events

IPIPE may generate the following events:

Event	Description
IC_EVENT_TYPE_READOUT_START	The first line of data for a given frame has arrived from a specific sensor
IC_EVENT_TYPE_READOUT_END	The first line of data for a given frame has arrived from a specific sensor
IC_EVENT_TYPE_LINE_REACHED	Line number N for a given frame has arrived from a specific sensor, where N was specified by the Client as a per-camera configuration parameter
IC_EVENT_TYPE_ISP_START	A specific frame has commenced being processed by a specific ISP pipeline
IC_EVENT_TYPE_ISP_END	A specific frame has finished being processed by a specific ISP pipeline
IC_EVENT_TYPE_ZSL_LOCKED	A request to lock a frame in the ZSL buffer has complete
IC_EVENT_TYPE_STATS_READY	Provides the Client with Autofocus and AE/AWB statistics for a given frame
IC_EVENT_TYPE_SOURCE_STOPPED	A request to stop a given camera has completed
IC_EVENT_TYPE_ISP_CONFIG_ACCEPTED	An ISP configuration structure sent from the client in no longer required, and can be reused by the client.
IC_EVENT_TYPE_ERROR	An error occurred
IC_EVENT_TYPE_LEON_RT_READY	The LeonRT has started, and the client may begin sending events.
IC_EVENT_TYPE_SOURCE READY	Announce client that respective source configuration has finished.
IC_EVENT_TYPE_SETUP_SOURCES_RESULT	Status message in response to IC_EVENT_TYPE_SETUP_SOURCES_COMMIT. Contains a status code, and indicates how much DDR is still free for allocation to ZSL buffers.
IC_EVENT_TYPE_ZSL_ADD_RESULT	Status message in response to IC_EVENT_TYPE_ZSL_ADD. Contains a status code, and indicates how much DDR is still free for allocation to ZSL

Event	Description
	buffers, and also how many frames are in the relevant ZSL buffer.
IC_EVENT_TYPE_ALLOC_MEM	#TBD
IC_EVENT_TYPE_SEND_OUTPUT_DATA	Announce client that a frame is ready to be sent further (e.g.: through MIPI)
IC_EVENT_TYPE_WAS_RESET	Response to IC_EVENT_TYPE_RESET event.
IC_EVENT_TYPE_TORN_DOWN	Response to IC_EVENT_TYPE_TEARD_DOWN event.

The above events are received by the Camera Framework's event thread (which receives events by calling `icGetEvent()`).

5.1.6 Source (per-camera) Parameters

Per-source parameters include the parameters required to configure the MIPI Rx block and allocate frame buffers, along with other camera-specific parameters. They are divided into two groups:

- Static parameters – these may not be modified while the camera is running. These parameters are configured via `icConfigureSource()`.
- Dynamic parameters – these may be modified at any time, even if the camera is running. These parameters are configured via `icConfigureSourceDynamic()`.
- #TBD

5.1.7 ISP Parameters

There may be multiple ISP pipelines in the system, and each may be configured independently. Sometimes an ISP may only implement a subset of functionality.

5.1.8 Synchronization and Frame Sequence Numbers

The application must be able to synchronize between the Camera Sensors, the ISP pipelines, and the control algorithms which use statistics derived from specific frames. For example, it needs to know what ISP configuration parameters were in effect when a specified frame was processed by the ISP, and what sensor parameters (e.g. exposure and gain) were in effect when that frame was captured.

To enable this, various facilities are provided:

- Each frame is assigned a Frame Sequence Number. Frame Sequence numbers start at 0, and increase by 1 between consecutive frames. Sequence numbers are tracked independently for each camera source.
- The application is notified via a callback about various events, such as a camera output reaching a specific line, or ISP completion of a specific frame. These events contain the sequence number of the frame to which the event pertains, along with a “userData” parameter (which was supplied by the client along with the ISP configuration parameters that were applied to the relevant frame).
- When an ISP parameter update takes effect, the client will be notified via an IC_EVENT_TYPE_ISP_CONFIG_ACCEPTED event. This event contains a “userData” pointer which

allows the Client to associate the event with a particular ISP configuration request. IPIPE should not reference the fields of the ISP configuration structure after it has been released.

5.2 Pipeline control API

5.2.1 icSetup()

5.2.1.1 Prototype

```
icCtrl *icSetup(int32_t appID, uint32_t ddrStart, u32 ddrSize);
```

5.2.1.2 Description

Initialize the client API and select the target application. Must be called before calling any other Client API call. A pointer to a structure is returned, which must be passed to subsequent client API calls.

Subsequent calls to this function are prohibited, unless `icTeardown()` is called first.

"appID" is a zero-based index selecting which IPIPE-based application is to be run. Multiple applications may be available for selection at runtime, to support different use cases. An application has a fixed set of plugins, linked together in a fixed way. It is possible to switch to a different application, by calling `icTeardown()`, and then calling `icSetup()` with a different application ID.

5.2.2 IcQuery()

5.2.2.1 Prototype

```
void *icQuery(icCtrl *ctrl, icQueryType queryType, int32_t index);
```

5.2.2.2 Description

Queries information about a Source, ISP, or Output. Valid value of "queryType", along with the type of structure that the returned pointer points to, are given in the following table:

queryType	Returned structure type
IC_QUERY_TYPE_SOURCE	icQuerySource
IC_QUERY_TYPE_ISP	icQueryIsp
IC_QUERY_TYPE_OUTPUT	icQueryOutput

The icQuerySource structure has the following members:

Field	Type	Description
attrs	uint32_t	Camera attribute flags.
TBD	uint32_t	Add anything we want.

The icQueryIsp structure has the following members:

Field	Type	Description
outputType	IcMipiRxDataTypeT	Describes sensor input format: IC_IPIPE_RAW_8 IC_IPIPE_RAW_10 IC_IPIPE_RAW_12 IC_IPIPE_RAW_14

Field	Type	Description
attrs	uint32_t	ISP attribute flags.
slaveConfig	int32_t	Specifies the index of an ISP instance to which this pipeline's configuration is slaved. If the ISP may be configured (i.e. it is not a slave with respect to configuration), the value of this field will be the ISP's own index. Otherwise, the ISP may not be configured independently, and uses the same configuration as the ISP with this index.

The icQueryOutput structure has the following members:

Field	Type	Description
attrs	uint32_t	Output attributes.
dependentSources	uint32_t	Bitmap of sources that this output is dependent on. If any of these sources are disabled, the output is implicitly disabled.

5.2.2.3 Return code

Returns a pointer to a structure whose type depends on the queryType parameter that was specified. Returns NULL if the Source, Isp or Output instance corresponding to "index" does not exist. The caller may query all instances for a given type of query, by calling this function in a loop, starting with an index of 0, and incrementing it each time, until NULL is returned.

5.2.2.4 Example

```
icQueryOutput *outputInfo = icQuery(ctrl, IC_QUERY_TYPE_OUTPUT, 0);
```

5.2.2.5 Implementation note

This function does not need to exchange messages with IPIPE. Instead, the information may be retrieved via pointers from the icCtrl, which were set up when icSetup() was called. The information returned is static, and will not change unless icTeardown() is called, followed by icSetup() with a different appID.

5.2.3 icSetupSource()

5.2.3.1 Prototype

```
void icSetupSource(icCtrl *ctrl,
    icSourceInstance srcIdx, icSourceSetup *setup);
```

5.2.3.2 Description

Configure the specified source with the specified configuration information. "srcIdx" is a zero-based index which selects the camera source to be configured. **The API must be in the "Configuring" state when this function is called**, which implies that all sources are stopped.

The fields of the icSourceSetup structure are defined as follows:

Field	Description
maxWidth	Maximum possible width for specified camera source.
maxHeight	Maximum possible height for specified camera source.
maxBpp	Maximum possible bits per pixel for specified camera source.

Field	Description
maxPixels	maxWidth multiplied by maxHeight

5.2.4 IcSetupSourcesCommit()

5.2.4.1 Prototype

```
void icSetupSourcesCommit(icCtrl *ctrl);
```

5.2.4.2 Description

Commits the setup of sources specified by calls to icSetupSource(). This function will cause IPIPE to allocate frame buffer memory in accordance with the parameters that were specified in the icSetupConfig structures. This function will block until an IC_EVENT_TYPE_SETUP_SOURCES_RESULT event is returned to the client in response. The results of the response will be stored in the structure which “result” points to.

The API must be in the “Configuring” state when this function is called, which implies that all sources are stopped.

Any source for which icSetupSource() was not called remains marked as “unused”. No frame buffer memory needs to be allocated for unused sources. Subsequent attempts to start an unused source will fail, and Outputs that are dependent on a source marked as unused will not produce any data.

If the response indicates that the configuration was successful, then sources may be started (except those which are marked as unused).

5.2.5 icConfigureSource()

5.2.5.1 Prototype

```
void icConfigureSource(icCtrl *ctrl,
    icSourceInstance srcIdx, icSourceConfig *config);
```

5.2.5.2 Description

Configure the specified source with the specified configuration information. “srcIdx” is a zero-based index which selects the camera source to be configured. **The API must be in the “Source Setup” state when this function is called**, which implies that all sources are stopped.

The fields of the icSourceConfig structure are defined as follows:

Field	Description
cameraOutputSize	Width and height of the frames that the camera is configured to output.
cropWindow	IPIPE can crop the sensor output at the very beginning of the pipeline. This field specifies the cropping window. The window must fit within the dimensions of the camera output frame size.
bayerFormat	Specifies the Bayer pattern. May be one of IC_BAYER_FORMAT_GRBG, IC_BAYER_FORMAT_GBRG, IC_BAYER_FORMAT_RGGB or IC_BAYER_FORMAT_BGGR.
bitsPerPixel	Bits per pixel of the camera output data. If a value of 8 or less is specified, the data is assumed to be 1 byte per pixel. Otherwise, the data is assumed to be 2 bytes per pixel. The value must be in the range [6, 16].

Field	Description
mipiRxData	MIPI inteface configuration parameters – icMipiConfig structure (controller number, number of lanes, lane rate (in Mbps), Data Type and Data Mode).
inFileNme	Input file name used just for PC run version.

5.2.6 icConfigureSourceDynamic()

5.2.6.1 Prototype

```
int32_t icConfigureSourceDynamic(icCtrl *ctrl,
    icSourceInstance source, icSourceConfigDynamic *config);
```

5.2.6.2 Description

Configure the specified source with dynamically-configurable parameters, in accordance with the specified configuration information. This function may be called regardless of whether the source is in the stopped state or note.

Field	Description
notificationLine	When the sensor has output the this line number, a notification event (IC_EVENT_TYPE_LINE_REACHED) is sent to the client. Specify -1 if the event is not required.

5.2.7 icStartSource()

5.2.7.1 Prototype

```
int32_t icStartSource(icCtrl *ctrl, icSourceInstance source);
```

5.2.7.2 Description

Starts the specified source (camera). When a source is started, video frames from that source will be captured and made available to other plugins in the application.

A source may only be started if it was configured by calling icSetupSource() and icConfigureSource() during the source configuration phase.

5.2.8 icStopSource()

5.2.8.1 Prototype

```
int32_t icStopSource(icCtrl *ctrl, icSourceInstance source);
```

5.2.8.2 Description

Stops a source that was previously started with icStartSource(). If the specified source was not already started, this call will be ignored.

5.2.9 icConfigureIsp()

5.2.9.1 Prototype

```
int32_t icConfigureIsp(icCtrl *ctrl, int ispIdx, void *cfg);
```

5.2.9.2 Description

Configures the specified ISP instance according to the specified parameters structure. The ISP instance is specified by “ispIdx”, which is a zero-based index.

The “cfg” parameter points to a structure containing various ISP parameters, as well as some control settings. The relevant control settings are described in this section.

The “cfg” configuration structure contains an “enableFlags” field in the parameters structure, which allows various pipeline blocks to be enabled/disabled.

The “cfg” configuration structure also contains a “userData” field. This field may be populated by the caller, and may be used to point to some user-specific data associated with the configuration set. This pointer will be included in subsequent events sent to the client, so that the client can keep track of what configuration settings are in effect during the processing of a given frame.

The new ISP configuration will take effect the next time processing of a frame by the specified ISP is commenced. The updates will be applied to complete frames: they will not take effect during the processing of a frame. The updated parameters will be applied atomically, that is, partial application of the updated parameters to any subsequently processed frame will not occur. If multiple calls are made to this function before processing of a new frame is commenced, the later calls will override the earlier ones, i.e. when processing of a frame begins, the ISP instance will use the latest configuration received.

The application is responsible for management of ISP configuration parameter structures. The application should not overwrite members of the parameters structure while the application of the parameters is still pending. Typically, the application will maintain a pool of parameter structures. IPIPE will send the IC_EVENT_TYPE_ISP_CONFIG_ACCEPTED event to indicate that the structure is free to be reused by the application.

5.2.10 icDataReceived()

5.2.10.1 Prototype

```
void icDataReceived(icCtrl *ctrl, FrameT *dataBufStruct);
```

5.2.10.2 Description

Announces server that the respective frame is processed, so the corresponding memory can be reused if necessary.

5.2.11 icLockZSL()

5.2.11.1 Prototype

```
int32_t icLockZSL(icCtrl *ctrl,
    icSourceInstance source, u32 frameSel, icLockZSLFlags flags);
```

5.2.11.2 Description

Selects a frame in the ZSL associated with the specified source, and locks it for subsequent Capture processing. A timestamp (which can relative) is provided with the Lock request. The frame is selected based on the frameSel parameter.

Once the frame is locked, an event is sent to the Client side, containing the userData associated with the locked buffer. The Client may then issue as many Capture requests as it likes, which will operate on the locked frame. Each capture request is accompanied by an ISP config struct. In order to perform multiple captures, a flag must be passed with the capture request to indicate that the frame should not be automatically unlocked in the ZSL buffer after the capture. Additionally, there is an explicit unlock event which may be sent to IPIPE, to release the frame (if any) currently locked in the ZSL buffer.

Supported flags are:

- IC_LOCKZSL_CLEAR_RAW
- IC_LOCKZSL_TS_RELATIVE – Relative timestamp vs. sequence no. If set, the 'frameSel' parameter specifies a relative time, in microseconds, indicating how long ago the desired frame should have been captured (timestamp of the frame selected from the ZSL buffer should be as close as possible to the current time, minus 'frameSel'). If not set, the 'frameSel' parameter specifies a frame sequence number. The sequence number of the frame selected from the ZSL buffer should be equal to the specified frame sequence number. If that frame is no longer in the ZSL buffer, then the oldest frame in the ZSL buffer will be selected. If the specified sequence number is newer than the newest frame in the ZSL buffer, then the newest frame in the buffer will be selected.

5.2.12 icTriggerCapture()

5.2.12.1 Prototype

```
int32_t icTriggerCapture(icCtrl *ctrl,
    icSourceInstance source, void * buff, void *cfg, icCaptureFlags
flags);
```

5.2.12.2 Description

Triggers processing of a frame stored in the ZSL buffer associated with the specified source. The frame must already have been locked in the ZSL buffer by a successful call to icLockZSL(). If no frame is currently locked in the ZSL buffer, then the call will have no effect.

5.2.13 icUnlockZSL()

5.2.13.1 Prototype

```
int32_t icUnlockZSL(icCtrl *ctrl, icSourceInstance source, FrameT * buff);
```

5.2.13.2 Description

Release the frame (if any) currently locked in the ZSL buffer.

5.2.14 icTeardown()

5.2.14.1 Prototype

```
int32_t icTeardown(icCtrl *ctrl);
```

5.2.14.2 Description

Halt all processing and streaming, and release all processor resources associated with IPIPE. The specified control structure, “ctrl” is no longer valid after this call. icSetup() needs to be called again prior to making any other client API calls.

5.2.15 icIsEventPending()

5.2.15.1 Prototype

```
int32_t icIsEventPending(icCtrl *ctrl);
```

5.2.15.2 Description

This function is non-blocking, and checks if an event is currently pending on the event queue from IPIPE. Returns 0 if no event is pending, otherwise returns 1.

5.2.16 icGetEvent()

5.2.16.1 Prototype

```
int32_t icGetEvent(icCtrl *ctrl, icEvent *ev);
```

5.2.16.2 Description

This function blocks until such a time as an event from IPIPE arrives. The Client is expected to dedicate a thread to receiving events from IPIPE. The clients event thread is expected to call this function in a loop. When an event arrives, the structure pointed to by “ev” is populated with a copy of the event's contents, and the function call returns. The event is removed from the event queue at this time. After handling the event (or passing it off to another thread to be handled), this function can be called again to get the next event.

The following types of event may be returned:

- IC_EVENT_TYPE_READOUT_START – This event is triggered when a camera has output the first line of data in a frame. The instance of the source and the sequence number of the frame coming from the camera are included in the event structure.
- IC_EVENT_TYPE_READOUT_END - This event is triggered when a camera has output the last line of data in a frame. The instance of the source and the sequence number of the frame coming from the camera are included in the event structure.
- IC_EVENT_TYPE_LINE_REACHED – this event is triggered when a camera has output the line specified by the “notificationLine” field of the icSourceConfigDynamic structure that was passed to icConfigureSourceDynamic(). The instance of the source and the sequence number of the frame coming from the camera are included in the event structure.
- IC_EVENT_TYPE_ISP_START – an ISP instance has started processing a frame. The instance of the ISP and the sequence number of the frame being output are included in the event structure.
- IC_EVENT_TYPE_ISP_END – an ISP instance has finished processing a frame. The instance of the ISP and the sequence number of the frame being output are included in the event structure.
- IC_EVENT_TYPE_ZSL_LOCKED – a frame has been locked in the ZSL buffer (in response to a call to icLockZSL) and is ready for Capture processing.
- IC_EVENT_TYPE_STATS_READY – Statistics for Auto Focus, Auto Exposure and Auto White Balance are available for a frame. The instance of the ISP and the sequence number of the frame being output are included in the event structure. A pointer to the statistics structure is contained in the event. The statistics structure is double-buffered, meaning the application has 1 frame interval to copy or use the data before it gets overwritten.
- IC_EVENT_TYPE_SOURCE_STOPPED – a source has stopped. The instance of the source is included in the event structure.
- IC_EVENT_TYPE_ISP_CONFIG_ACCEPTED – an ISP configuration structure has been released, due to a previous call to icConfigureIsp(). The “userData” value that was originally passed in the ISP configuration

structure is contained in the event.

- IC_EVENT_TYPE_ERROR – An error has occurred.
- IC_EVENT_TYPE_LEON_RT_READY – The LeonRT has started, and the client may begin sending events.
- IC_EVENT_TYPE_SOURCE_READY – Announce client that respective source configuration has finished.
- IC_EVENT_TYPE_SETUP_SOURCES_RESULT – Status message in response to IC_EVENT_TYPE_SETUP_SOURCES_COMMIT.
- IC_EVENT_TYPE_ZSL_ADD_RESULT – Status message in response to IC_EVENT_TYPE_ZSL_ADD.
- IC_EVENT_TYPE_ALLOC_MEM – #TBD
- IC_EVENT_TYPE_SEND_OUTPUT_DATA – Announce client that a frame is ready to be sent further (e.g.: through MIPI).
- IC_EVENT_TYPE_WAS_RESET – Response to IC_EVENT_TYPE_RESET event.
- IC_EVENT_TYPE_TORN_DOWN – Response to IC_EVENT_TYPE_TEARD_DOWN event.

6 Creating application specific Plug-ins

6.1 Creating a Plug-in

6.1.1 PlgTemplate

6.1.1.1 Description

This plugin copies a frame from input place to output place.

6.1.1.2 Plugn specific structure

```
typedef struct PlgTemplateStruct {
    // generic component interface
    PlgType plg;

    // specific component interface
    // specific plugin command function, init, configs ...
    void    (*init)          (icSize iframeSize, void *pluginObject);
    // callback possible to be linked to the plugin,
    void    (*procesStart)   (uint32_t seqNr);
    void    (*procesEnd)     (uint32_t seqNr);

    /// Private members. All data structures have to be internal
    FramePool               *outputPools;
    icSize                  privateCfg;
    volatile int32_t        crtStatus; // internal usage
    FrameProducedCB         cbList[1];
} PlgTemplate;
```

6.1.1.3 APIs

6.1.1.3.1 PlgTemplateCreate()

Function	void PlgTemplateCreate(void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObject	Array of type PlgTemplate; number of array elements depend on number of cameras used
Return value	void	–	

The function sets some internal links and values, and initializes some required fields.
Some fields may be overwritten from the application afterward.

NOTE: Calling this API is mandatory for using this plugin

6.1.1.4 Plugin specific functions

6.1.1.4.1 Generic functions

The generic functions, which are necessary for all plugins, are described in chapter 3.2 Plug-in API.

6.1.1.4.2 Initialization

Function	void specificPlginit(icSize iframeSize, void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	icSize	iframeSize	Frame size described in width and height
Parameter 2	Void *	pluginObject	Array of type PlgTemplate; number of array elements depend on number of cameras used
Return value	void	–	

The function sets the parameters necessary for the plugin.

NOTE: Calling this API is mandatory for using this plugin

6.1.1.4.3 Callback

Function	static void producedInputFrame(FrameT *frame, void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	FrameT *	frame	Configuration parameters
Parameter 2	Void *	pluginObject	Array of type PlgTemplate; number of array elements depend on number of cameras used
Return value	void	–	

The function sets the parameters necessary for the plugin.

NOTE: Calling this API is mandatory for using this plugin

6.2 Predefined Plug-ins

6.2.1 PlgSource

Configure and control MIPI input source. Control MipiPhy, MipiControler, and receiver (sippRx and cif).

6.2.1.1 Description

Plugin configures and controlls MIPI and SippRx/Cif.
APIs need to be called for each active input(camera).
Output of one plugin instance is one frame pool containing new frames received from one camera.

6.2.1.2 APIs

6.2.1.2.1 PlgSourceCreate()

Function	void PlgSourceCreate(void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObject	Array of type PlgSource; number of array elements depend on number of cameras used
Return value	void	–	

The function sets some internal links and values, and initializes some required fields.
Some fields may be overwritten from the application afterward.

NOTE: Calling this API is mandatory for using this plugin.

6.2.1.2.2 PlgSourceStart()

Function	void PlgSourceStart(void *pluginObject, icSourceConfig *sourceConfig, uint32_t outFmt)		
Description	Create a frame pool.		
	Field type	**Field name**	**Field description**
Parameter 1	void *	pluginObject	User defined structure of type PlgSource
Parameter 2	IcSourceConfig *	srcCfg	Pointer to a fixed sensor

	Field type	Field name	Field description
			configuration; application specific
Parameter 3	uint32_t	outFmt	Enable/Disable packed data storage; possible values: //Possible pixel formats **#define** SIPP_FMT_8BIT 0x1 **#define** SIPP_FMT_16BIT 0x2 **#define** SIPP_FMT_32BIT 0x4 **#define** SIPP_FMT_PACK10 0x5 //packed type **#define** SIPP_FMT_PACK12 0x3 //packed type
Return value	void	–	

Start source plugin, which will also start the other plugins (if it's the case) through the callbacks chain.

NOTE: Calling this API is mandatory for using this plugin.

6.2.1.2.3 PlgSourceSetLineHit()

Function	void PlgSourceSetLineHit(void *pluginObject, uint32_t lineNo)		
Description	Configures a callback to be triggered when the desired number of lines is received.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObject	User defined array of type PlgSource
Parameter 2	uint32_t	lineNo	Number of lines after which the application needs to be notified.
Return value	void	–	

NOTE: Calling this API is optional.

6.2.2 PlgFifo

Serialize maximum 6 inputs source.

Assume X sources call Y consumer plugins. But that some consumers use same HW resources, in consequence they are not allowed to run in parallel. The approach is to use a FIFO list, and when the resources are free, call the trigger function of the FIFO plugin. This function will call the associated consumer callbacks, if FIFO plugin has a frame in the frame list.

Where X={0, …,6}, Y>0

6.2.2.1 Description

Plugin has several input frame pools and several output frame pools.

APIs need to be called only once for all inputs, with respect to the number of inputs used.

Writing in the fifo buffer is done in background, whenever the input frame pools require it. It is not required that the inputs are synchronized, meaning they can require to add frames to FIFO array in any order (as you can see in Figure 3).

In order to write in the output frame pool(s), the plugin has to be externally triggered through the “trigger” function call (as you can see in Figure 4). This should be done only when the consumer is ready to accept a new frame.

Output of plugin consists on frame pools which can be further used by other plugins, or they can be stored or transmitted.

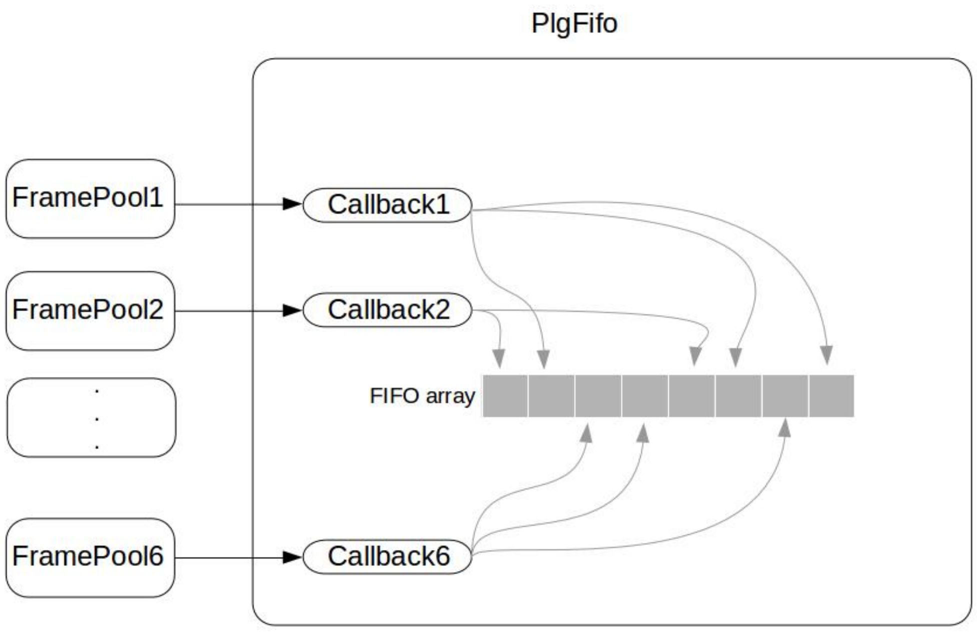

Figure 3: Example of adding elements to FIFO array

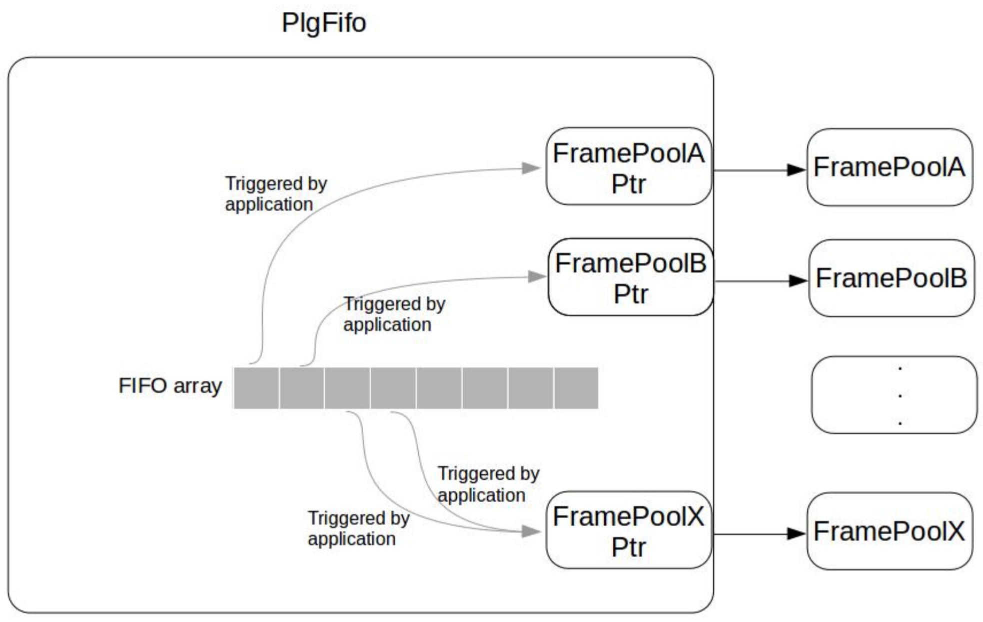

Figure 4: Example of using elements from FIFO array

6.2.2.2 APIs

6.2.2.2.1 PlgFifoCreate()

Function	void PlgFifoCreate(void *pluginObj)		
Description	Initialize pluginObj fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObj	Array of type PlgFifo; number of array elements depend on number of inputs used
Return value	void	–	

The function sets some internal links and values, and initializes some required fields.
Some fields may be overwritten from the application afterward.

NOTE: Calling this API is mandatory for using this plugin.

6.2.2.2.2 PlgFifoConfig()

Function	void PlgSourceStart(void *pluginObj, uint32_t NoOfInputs)		
Description	Set callbacks based on number of active cameras.		
	Field type	**Field name**	**Field description**
Parameter 1	void *	pluginObj	User defined array of type PlgSource; number of array elements depend on number of inputs used.
Parameter 2	uint32_t	NoOfInputs	Maximum number of inputs.
Return value	void	–	

NOTE: Calling this API is mandatory for using this plugin.

6.2.3 PlgFullIsp

Apply full Movidius ISP over a raw input image.

6.2.3.1 Description

Plugin can process one image at a time.

APIs need to be called once for each input.

Output of this plugin is a processed image, which is sent to a frame pool.

The image can then be stored, processed by other plugins, or used directly by the application (e.g.: sent via MIPI or via HDMI).

6.2.3.2 APIs

6.2.3.2.1 PlgIspFullCreate()

Function	void PlgIspFullCreate(void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObject	Array of type PlgIspFull; number of array elements depends on number of cameras used
Return value	void	–	

The function sets some internal links and values, and initializes some required fields.
Some fields may be overwritten from the application afterward.

NOTE: Calling this API is mandatory for using this plugin.

6.2.3.2.2 PlgIspFullConfig()

Function	void PlgSourceStart(void *pluginObj, uint32_t NoOfInputs)		
Description	Set callbacks based on number of active cameras.		
	Field type	**Field name**	**Field description**
Parameter 1	void *	plgObject	User defined array of type PlgSource; number of array elements depend on number of cameras used
Parameter 2	icSize	frameSz	Frame size described in width and height
Parameter 3	uint32_t	inFmt	Pixel format: packed or unpacked; possble values: **#define** SIPP_FMT_8BIT 0x1 **#define** SIPP_FMT_16BIT 0x2 **#define** SIPP_FMT_32BIT 0x4 **#define** SIPP_FMT_PACK10 0x5 // packed type **#define** SIPP_FMT_PACK12 0x3 // packed type
Parameter 4	uint32_t	prevAble	Enable image preview
Return value	void		

This API actually creates the Opipe.

NOTE: Calling this API is mandatory for using this plugin.

6.2.4 PlgSadDm

Calculate a dense depth map, based on the sum of absolute difference filter..

6.2.4.1 Description

Plugin applies algorithm based on 2 input frame pools, containing frames received from 2 different cameras.

Output of this plugin is a processed image, which is sent to a frame pool.

The image can then be stored, processed by other plugins, or used directly by the application (e.g.: sent via MIPI or via HDMI).

6.2.4.2 APIs

6.2.4.2.1 PlgDmCreate()

Function	void PlgDmCreate(void *pluginObject)		
Description	Initialize pluginObject fields which do not depend on external configuration.		
	Field type	**Field name**	**Field description**
Parameter 1	Void *	pluginObject	Type PlgDm
Return value	void	–	

The function sets some internal links and values, and initializes some required fields.

Some fields may be overwritten from the application afterward.

NOTE: Calling this API is mandatory for using this plugin.

6.2.4.2.2 PlgDmSetParams()

Function	void PlgDmSetParams(void *pluginObject, icSize frameSz, uint32_t firstShave, uint32_t lastShave)		
Description	Load code to specified number of shaves.		
	Field type	**Field name**	**Field description**
Parameter 1	void *	pluginObject	User defined array of type PlgDm.
Parameter 2	icSize	frameSz	Frame size described in width and height.
Parameter 3	uint32_t	firstShave	Number between 0 and 12; specifies 1st shave in which code will be loaded.
Parameter 4	uint32_t	lastShave	Number between 0 and 12; specifies last shave in which code will be loaded; should be grater then

	Field type	Field name	Field description
			firstShave.
Return value	void	–	

This API sets the transmitted parameters and loads the code which calculates the depth map in the specified shaves.

NOTE: Calling this API is mandatory for using this plugin.

7 Ipipe2 Pipelines

7.1 Description

An Ipipe2 pipeline is a specific case of plugin connections.

7.2 Pipeline example: IspUpTo3Cams

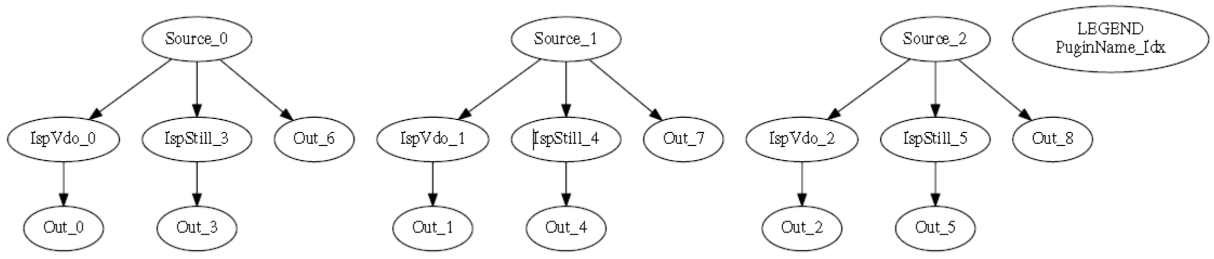

Figure 5: Pipeline with up to 3 cameras

Each sensor uses:

- An instance of Source plugin.
- FIFO plugin.
- Two instances of ISP plugin (one video and one for still).

8 Creating Applications with Plug-ins

8.1 Adding a plugin to an already existing pipeline

NOTE: The situation described below refers to a plugin which has one input point (one callback) and one output point (one frame pool).

1. Include plugin header file:

   ```
   #include "PlgTemplateApi.h"
   ```

2. Add path to makefile:

3. Define the plugin specific type variable:
   This will be the communication "channel" between the application and the plugin, sending information in both directions.

   ```
   PlgTemplate        plgTemplate SECTION(".cmx.cdmaDescriptors") ALIGNED(8);
   ```

   NOTE: Each plugin has its own structure definition, but it is mandatory to contain the PlgType structure.

4. Define the frame pool of the plugin:
   This is the output of the plugin, where the processed data is stored.

   ```
   FramePool       frameMgrPoolTemplate;
   ```

   NOTE: FramePool type is common for all plugins used

   a. Define the frame manager, which is actually a pointer to the first element of the list of frames.

   ```
   FrameT          *frameMgrFrameTemplate;
   ```

   b. Define number of frames of the plugin output buffer (usually 3).

   ```
   #define NR_OF_BUFFERS_PER_TEMPLATE_OUT 3
   ```

5. Create plugin:

   ```
   PlgTemplateCreate(void *plgTemplate);
   ```

6. Add application callback only if the plugin is the last one (no other plugin processes the output frame pool):

   NOTE: This step is not mandatory, it depends on the plugin scope.

Old pipeline:

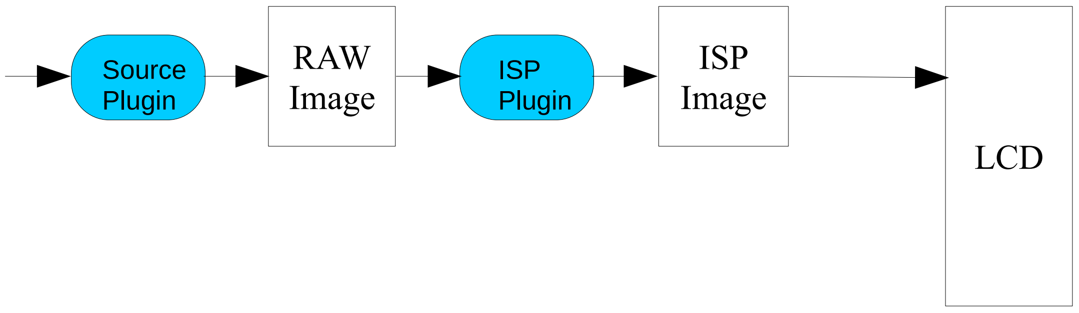

Figure 6: Original ISP Pipeline

Case A – Add the callback to the application callback list:

```
        FrameProducedCB cbOutputList[...] = yourCallback;
```

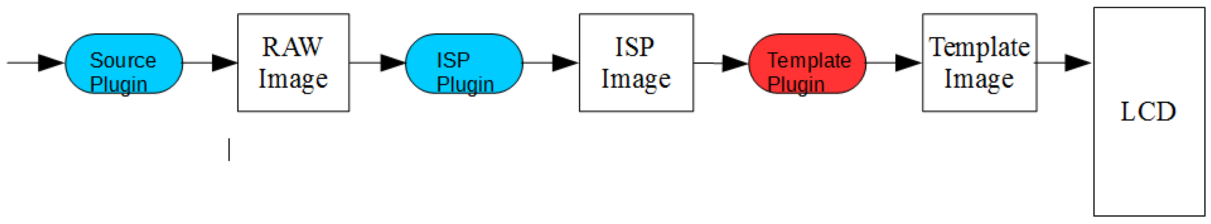

Figure 7: Pipeline structured with plug-ins

Case B – Create a new callback list (if old output is still needed):

```
        FrameProducedCB cbOutputListPlgTemplate[2];
```

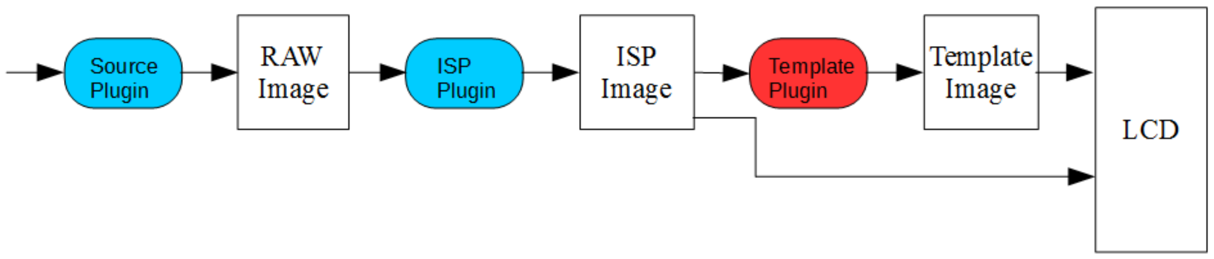

Figure 8: Pipeline with re-directed output

Any case:

Define the callback:

```
void yourCallback(FrameT *frame, void *pluginObj) {
   // ...
}
```

7. Create the frame pool of the plugin:

```
FrameMgrCreatePool(&frameMgrPoolTemplate, frameMgrFrameTemplate,
&cbOutputList[4], 1);
```

NOTE: If the plugin is not the last one, the frame pool will be connected to the next plugin through its callback.

`&plgNextPlugin.plg.callbacks instead of the application callback &cbOutputList[4]`

8. Create the frame list:
A buffer of NR_OF_BUFFERS_PER_TEMPLATE_OUT elements. By creating the frame list, the frame manager is also initialized.

```
// Allocate frames buffers memory
         frameMgrFrameTemplate =
ipServerFrameMgrCreateList(NR_OF_BUFFERS_PER_TEMPLATE_OUT);
```

9. Allocate memory for the frame list:

```
ipServerFrameMgrAddBuffs(frameMgrFrameTemplate,maxOutputFrameSizeLayer
1,maxOutputFrameSizeLayer2, maxOutputFrameSizeLayer3);
```

10. Set the Frame Manager parameters:

```
FrmMgrUtilsInitList(frameMgrPoolTemplate, frameSize, frameFormat);
```

11. Initialize the plugin:

NOTE: Additionally a configuration API should be called, only if provided in the header file of the plugin.

```
plgTemplate.plg.init(&frameMgrPoolTemplate, 1, (void*)&plgTemplate);
```

12. Check the status of the plugin in order to decide if an action can be performed or not:

NOTE: This step is not mandatory, it depends on the plugin scope.

```
if(0 == plgTemplate.plg.status)
```

13. Trigger the plugin:

NOTE: This step is not mandatory, it depends on the plugin scope.

```
            plgTemplate.plg.trigger(frame, params, NULL, &plgTemplate);
```

14. Stop the plugin:

```
              plgTemplate.plg.fini(&plgTemplate);
```