NXP	AMP Software	
ADAS VISION	Revision <1.3>	Page 1 of 21
	AMP_SW	

ISP/Sequencer Driver Software User Guide

ABSTRACT:
This is the Software User Guide Document for ISP/Sequencer driver of the Vision SDK.
KEYWORDS:
Sequencer, FDMA, SDI, stream, frame
APPROVED:

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	05-May-16	Tomas Babinec	Document creation.
0.2	31-August-16	Tomas Babinec	Updated for VSDK 0.9.5 release
1.0	06-June-17	Tomas Babinec	Updated for VSDK 1.0 RTM release
1.1	19-January-18	Tomas Babinec	Updated to LLDCMD
1.2	16-March-18	Nguyen Tri Hai	Updated for VSDK 1.1 RTM release
1.3	20-August-18	Khang Ba Tran	Updated for VSDK 1.2 RTM release

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to describe ISP/Sequencer driver user space interface. It is intended to serve as a reference source during the development of VSDK based applications. For exact definitions and implementation details please refer to [3].

1.2 Audience Description

This document is intended for s32v234 Vision SDK users.

1.3 References

Id	***Title***	***Location***
[1]	*S32v234 Reference Manual*	*Sharepoint*
[2]	*ISP Sequencer driver source code documentation*	*Doxygen comments in VSDK*
[3]	*SDI SW User Guide*	*Vision sdk git, folder:* s23v234_sdk\docs\drivers\

Table 1: References

1.4 Definitions, Acronyms and Abbreviations

Term/Acronym	***Description***
API	*Application Programming Interface*
CRAM	*Code and Data Random Access Memory*
CRC	*Cyclic Redundancy Check*
CSE	*Cryptographic Security Engine*
DDR	*Double Data Rate DRAM*
DRAM	*Dynamic Random Access Memory*
fDMA	*fast Direct Memory Access HW block*
HW	*Hardware*
IMEM	*Instruction Memory*

IP	*Intellectual Property*
IPUx	*Image processing unit, as there are IPUS (scalar) and IPUV (vector)*
ISP	*Image signal processor (whole image processing system)*
ISP engine	*Device inside ISP, like camera interface, IPUx, en/decoder, FastDMA*
KRAM	*Kernel Random Access Memory*
LLDCMD	*LowLevel Driver Command*
PRAM	*Parameter Random Access Memory*
SDI	*Sensor Data Interface library*
Sequencer	*Micro processor to control the ISP engines and memory*
Sequencer Graph	*SW description of data flow and processing which can be executed by the ISP*
SoC	*System on Chip*
SRAM	*Static Random Access Memory*
SW	*Software*
TC	*Transfer Channel*
TD	*Transaction Descriptor*
TDM	*Transaction Descriptor Metadata*
TDT	*Transaction Descriptor Table*

Table 2: Acronyms

1.5 Document Location

This document is available in VisionSDK directory structure at the following location:
s32v234_sdk\docs\drivers\ Sequencer_Driver_Software_User_Guide.pdf

2 General Description

The ISP/Sequencer (further referred to as Sequencer) driver software (SW) is intended for kernel space management of the ISP/Sequencer HW block, which is designed to be a part of the S32V234 SoC. An integral part of the driver is also a user space library providing an API for the user applications. This API wraps the kernel space interface of the driver (LLDCMD commands, etc.).

2.1 Sequencer HW

The Sequencer HW was designed for S32V234 SoC to control video data preprocessing pipeline where particular steps of the image processing can be carried out by different HW modules called ISP (Image Signal Processing) engines. The Sequencer is based on an ARM M0+ core that executes SW (Sequencer FW) specialized for the control of various ISP engines. The M0 core is accompanied by Control Block and Event Control block that provide HW support for various event handling.

Main purpose of the Sequencer is to ensure data coherency between the different image data processing steps thus reducing computational load and hard real-time requirements that would be otherwise placed on the Host CPU.

As you can see in Figure 1, the ISP engines the Sequencer is aware of include camera interfaces (MIPI-CSI2 & ViuLite), H.264 encoder/decoder, JPEG decoder, IPUx (scalar and vector Image Processing Units) and fDMA HW block. The ISP engines are expected to work mainly with data located in SRAM block, which is capable to provide high bandwidth access, but is rather limited in memory space (4 MB). Therefore, only a small subset of image data (required for currently ongoing processing) should be stored in SRAM space for each step in the pipeline. The expected granularity of the stored image subsets in SRAM is one image line.

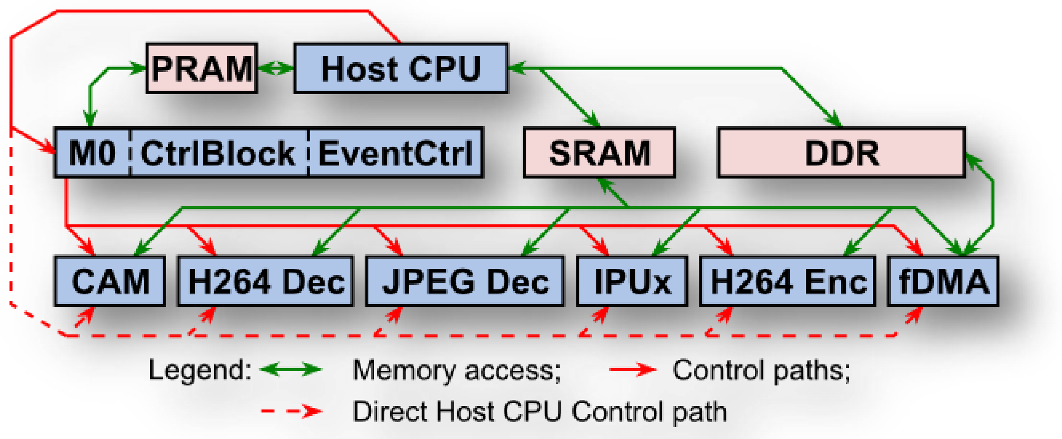

Figure 1: Sequencer HW interactions

Except for the IPUx HW blocks the Sequencer's ability to control the ISP engines is limited in general and to fully set up the preprocessing pipeline a close cooperation with the Host CPU is required. The PRAM block and dedicated set of IRQ channels provide HW support for SW implementation of a communication scheme.

The Host CPU responsibilities include:

- SRAM & DDR buffer allocation.
- fDMA transaction configuration using transaction descriptors in the SRAM. The SRAM (as well as the DDR) memory region is invisible to the Sequencer;
- Camera setup through I^2C bus;
- H.264 encoder/decoder and JPEG decoder configuration through its registers.

The Host CPU interaction is required especially:

- before the Sequencer managed preprocessing pipeline is started,
- in case of an error,
- when the processing pipeline is being changed or terminated,
- periodically on a frame basis. E.g. camera gain control, frame DONE event processing. It is expected to keep the amount of periodic interactions as low as possible.

There is a possibility to dedicate one of the Host CPU cores to control the video data preprocessing pipeline instead of the Sequencer's AMR M0+ core. This feature is not supported by the driver at the moment.

3 Functional Description

The Sequencer driver SW has two layers (see Figure 2). The first layer operates in kernel space and accomplishes most of the driver's functionality.

The second layer is implemented as a user space library creating a thin interface for user level SW (SDI or directly a user application) to access the kernel part functionality. The provided user level API is explained in section 3.4.

To invoke low-level (kernel) driver functionality from user-space library a LLDCMD (Low-Level Driver CoMmanD) interface is used. It is implemented in OAL library by `OAL_LldCmd(OAL_LldCmd_t*)` function.

To inform about asynchronous events (like IRQs), which were detected by the low level (kernel) driver, event signaling and user application hander registering API is implemented.

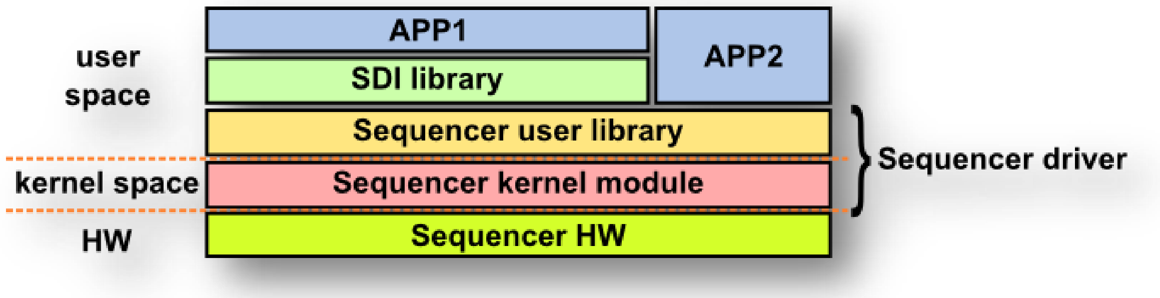

Figure 2: Sequencer driver software layout

3.1 Sequencer HW

The Sequencer HW operation is controlled by the firmware for Arm M0+ core. The firmware can manage broad variety of data preprocessing pipelines which have to be defined in a form Sequencer graphs. Usually one or more IPU engines are involved in the processing. The IPU engine executes programs called IPU kernels.

The Host CPU is responsible for proper configuration of the Sequencer HW including the download of required M0+ firmware & Sequencer graph to CRAM, IPU kernels to KRAM and management of HOST/Sequencer intercommunication data in PRAM.

With the exception of the IPUx HW blocks the Sequencer does not have the ability to fully configure the ISP engines. Because of this limitation, a HW accelerated communication interface (based on HW accelerated PRAM message exchange) was implemented between the Host CPU and the Sequencer HW.

The s32V234 SoC includes SRAM memory access control block (XRDC) that by default disables write access to SRAM from several ISP engines. The SRAM access control block has to be configured properly to ensure expected ISP preprocessing behavior.

The KRAM region is by default protected by a security engine which disables any write access to it. To successfully execute any ISP based application a security fuse has to be burned in advance.

3.2 Data types

The Sequencer driver introduces the following data types and containers (see [3] for full definitions):

Messaging

- Structure `SEQ_Message`:

  Defines one message used to communicate events between the Host CPU and the Sequencer. The message has a fixed size (currently 64 bytes) with common head (4bytes) and arbitrary data section (60 bytes). The data section can be retyped to various predefined message data structures based on the type member in the head.

- Enums `SEQ_MessageStatus`, `SEQ_MessageType`:

  Define various states and types a message described by the `SEQ_Message` structure is allowed to have.

- Particular predefined message types:

  `SEQM_start`, `SEQM_swreset`, `SEQM_start_node`, `SEQM_echo`, `SEQM_profile`, `SEQM_fdma_sched`, `SEQM_fdma_event`, etc. For more information on predefined messages please see [2] especially the s32v234_sdk/isp/inc/seq_comm.h.

- Structure `SEQ_MessagePool`:

  Defines a pool of messages organized as a circular buffer. It is used to implement a communication channel between the Host CPU and the Sequencer.

Buffers handling

- Structure `SEQ_FrmBuffer`:

  Defines a container of one DDR frame data. Set of these buffers creates a buffer stream which is used for continuous frame grabbing by ISP. For more information please see 3.3.2.

- Structure `SEQ_FrmBufferInfo`:

  Defines a user side handle for a particular `SEQ_FrmBuffer`. For more information please see section 3.3.2.

- Structure `SEQ_FrmStream`:

  Used to manage a set of `SEQ_FrmBuffers` that create a separate ISP stream. For more information please see section 3.3.2.

- Structure `SEQ_BufferRequest`:

  Used to request a particular buffer stream to be initialized with pre-allocated memory addresses. For each address a `SEQ_FrmBuffer` structure is created and assigned to the stream. For more information please see section 3.3.2.

- Enum `SEQ_FrmBufferState`:

  Defines possible states (work, done, user) the SEQ_FrmBuffer can have. For more information please see section 3.3.2.

Driver internal management:

- Structure `SEQ_DrvInfo_t`:

  Collects internal data of the driver.

- Enum `SEQ_FwType`:

  Enumerates possible Sequencer firmware types: `SEQ_FW_CM0` CM0 binary, `SEQ_FW_KERNEL` IPU kernel code and `SEQ_FW_GRAPH` graph.

- Structure `SeqFw`:

  Collects all FW lines from M0 binary S-record.

- Structure `SewFwLine`:

  One line from M0 binary S-record.

- Structure `SEQ_GraphPackage`:

  Describes a preprocessed Sequencer graph that can be downloaded to PRAM.

Event related:

- Function pointer `SEQ_EventHandler_t`:

  Defines a prototype for Sequencer event handler function.

- Class `SEQ_EventHandlerClass`:

  Pure virtual class definition that can be used for Sequencer event handling in C++ based applications.

- Structure `SEQ_AuxData`:

  Defines a buffer for copy of current state of Message auxiliary data in PRAM.

- Structure `SEQ_IpuHist`:

  User space container for histogram values gathered from particular IPU engine.

- Structure `SEQ_IpuHistHead`:

  Kernel space container for histogram values gathered from particular IPU engine.

- Enum `SEQ_LIB_RESULT`:

  Defines return values (`SEQ_LIB_SUCCESS`, `SEQ_LIB_FAILURE`) of most of the Sequencer user library functions.

- Struct `SEQ_RegList_t`:

  Container to store new values for coherent update of IPU registers.

Profiling:

- Structure `SEQ_Profile`:

  Describes a profiling information for the Sequencer firmware.

- Enum `SEQ_IpuEngine`:

States possible IPU engines – defines their index in driver internal structures.

- Structure `SEQ_IpuPerformanceRegs`:

  Contains profiling data for one IPU.

- Union `SEQ_EventDesc`:

  Describes one ISP related event. This includes the event type (IPUx/Other engine signal, Frame done, CSI frame end) and associated register copy if any.

- Structure `SEQ_EventRecord`:

  Combines `SEQ_EventDesc` with the time (micro seconds from reset) the event has been signaled at.

- Structure `SEQ_EventLog`:

  Describes an array of `SEQ_EventRecords`.

Types not mentioned in the list above are of minor importance and their purpose can be understood directly from the code.

3.3 Kernel Space

The internal functionality of the Sequencer kernel module and its API manage the low level HW communication and make the Sequencer HW features available for user applications.

3.3.1API functions

This section, Table 3, describes functionality exported by the Sequencer driver module. It is intended to be used by upper layer SW such as LLDCMD in case of Linux environment or directly by the user library in case of a standalone setup. Exact function headers declarations can be found in `seq_func.h`.

In Linux environment, the Sequencer driver is associated with special device file `seq`. Download of the graphs, IPU kernels and M0+ firmware is mediated by mapping of related memory blocks to user spaces (described in 3.4).

Function; LLDCMD	Description

`SEQ_DRV_Setup()`
`---`
	First time use initialization. To enable HW interaction and setup internal structures. In Linux invoked when `seq` device file opened.

`SEQ_DRV_Close()`
`---`
	To terminate driver operations. HW reset, release of all resources, reset of internal structures. In Linux invoked when `seq` device file closed.

`SEQ_DRV_Reset()`
`SEQ_RESET`
	Puts the Sequencer HW to reset. Resets also internal driver data.

`SEQ_DRV_Boot()`
`SEQ_BOOT`
	Brings Sequencer HW out of reset and boots up the Sequencer SW.

	Waits for "Ready after boot" message.

`MsgSend()`
`SEQ_MSG_SEND`
	Sends a message to Sequencer FW.
	
	Parameters: structure `SEQ_Message_t` defining the message and aWait variable. If flag is negative value the call blocks until message was acknowledged by Sequencer FW. If flag is positive value the call blocks until message was acknowledged by Sequencer FW or timeout. If flag is zero non-blocking call, no wait for acknowledge

`FdmaTcScheduleMsgSend()`
`---`
	Schedules a FDMA transaction through Sequencer FW.
	
	Parameters: FDMA_Tc_t structure.
	
	NOT SUPPORTED/IMPLEMENTED

`SEQ_DRV_GraphDownload()`
`SEQ_GRAPH_DOWNLOAD`
	Writes given graph data into PRAM.
	
	Parameters: Structure containing base address of the graph data, its size in bytes and array of buffer lists associated to different fDMA channel numbers.

`PatchListSet()`
`SEQ_PATCH`
	Writes patch-list array to PRAM.
	
	Parameters: Patch-list structure and number of patches included.

`SEQ_DRV_GraphFetch()`
`SEQ_GRAPH_FETCH`
	Loads current state of the Graph from PRAM location.
	
	Parameters: pointer to a graph package structure.
	
	INTENDED FOR DEBUG PURPOSES.

`GraphStart()`
`SEQ_START`
	Sends the `SEQ_MSG_TYPE_START` message to Sequencer. Blocks until the message is acknowledged by Sequencer.
	
	Parameters: Number of frames to be processed. 0 represents no frame count limit.

`SEQ_DRV_GraphStop()`
`SEQ_STOP`
`SEQ_STOP_WAIT`
	Checks for the execution state of the graph.
	
	Sends the `SEQ_MSG_TYPE_STOP` message to Sequencer.
	
	Parameters: If to wait for M0 response (1 = wait, 0 = don't wait).

`SEQ_DRV_GraphStateGet()`
`---`
	Checks for the execution state of the graph.
	
	Returns: 0 if graph is off otherwise 1.

`SEQ_DRV_FrmBuffersRequest()`
`SEQ_BUF_REQ`
	Requests driver internal framebuffers to be created for given stream.
	
	Parameters: Request description structure. If requested

	buffer count is 0, the last ones are removed.
`SEQ_DRV_FrmBufferQuery()` `SEQ_BUF_QRY`	Queries the existence and info for a particular buffer. **Parameters**: `SEQ_FrmBufferInfo` structure describing the buffer.
`SEQ_DRV_FrmBufferPush()` `SEQ_BUF_PSH`	Puts a buffer to the drivers "work queue" to be used for data grabbing. **Parameters**: `SEQ_FrmBufferInfo` structure describing the buffer.
`SEQ_DRV_FrmBufferPop()` `SEQ_BUF_POP`	Attempts to pops a buffer from the drivers "done queue" to make it available for application use. **Parameters**: `SEQ_FrmBufferInfo` structure describing the buffer.
`SEQ_DRV_PramAuxDataGet()` `SEQ_AUX_DATA`	Gets the message auxiliary data structure pointer. **Parameters:** `SEQ_AuxData` structure to be filled in.
`SEQ_DRV_RegListSet()` `SEQ_REGLIST_SET`	Setups a list of IPU registers that should be updated or read by M0 coherently at the frame boundary. **Parameter:** Pointer to `SEQ_RegList_t`. In case previous register list was not used yet the function returns success but the mCnt is reset to 0 to inform upper level SW.
`SEQ_DRV_RegListGet()` `SEQ_REGLIST_GET`	Reads current content of register list in PRAM. **Parameters:** SEQ_RegList_t structure.
`IpuPerformanceInfoRequest()` `SEQ_IPU_PROFILE_REQ`	Instructs Sequencer to IPU performance data collection. **Parameters:** Number of frames to collect the data for.
`IpuPerformanceInfoGet()` `SEQ_IPU_PROFILE_GET`	Attempts to get the IPU profiling data. **Parameters:** Array of `SEQ_IpuPerformanceRegs` to be filled in.
`SEQ_DRV_EventLogGet()` ---	Gets the Event log array. **Parameters:** `SEQ_EventLog` structure. For standalone environment and debug purpose only.
`SEQ_DRV_EventLogEnable()` ---	Enables the Event logging. **Parameters:** `uint32` flag array of what events to use. For standalone environment and debug purpose only.
`SEQ_DRV_EventLogDisable()` ---	Disables the Event logging. For standalone environment and debug purpose only.

`IpuHistogramEnable()`
`SEQ_HIST_ENABLE`
	Enables the use of histogram on particular IPU.
	Parameters: `SEQ_IpuEngine_t` index of the IPU.

`IpuHistogramDisable()`
`SEQ_HIST_DISABLE`
	Disables the use of histogram on particular IPU.
	Parameters: `SEQ_IpuEngine_t` index of the IPU.

`SEQ_DRV_IpuHistogramGet()`
`SEQ_HIST_GET`
	Gets the histogram from specified IPU engine .
	Parameters: Pointer to `SEQ_IpuHistHead_t`.

`SEQ_DRV_IpuStatEnable()`
`SEQ_STAT_ENABLE`
	Enables the use of statistic data from IPUS_7.

`SEQ_DRV_IpuStatDisable()`
`SEQ_STAT_DISABLE`
	Disables the use of statistic data from IPUS_7.

`SEQ_DRV_IpuStatGet()`
`SEQ_STAT_GET`
	Gets the statistic data from IPUS_7.
	Parameters: Pointer to `SEQ_IpuStatHead_t`.

`SEQ_DRV_TimeStatsGet()`
`SEQ_TIME_STAT`
	Gets current state of temporal frame statistics.

Table 3: Sequencer driver API

3.3.2Graph management

The ISP graph that was preprocessed by the SDI is supposed to be downloaded to a predefined location in CRAM using the `SEQ_DRV_GraphDownload()` function. Parameter of the function is a graph package structure pointer which contains Sequencer graph data together with the SRAM buffer list array. Both of these structures are copied to proper CRAM location and the number of lines per frame for the particular graph is remembered by the Sequencer driver.

The graph download functionality will be moved to user library to reduce the number of operations done in the kernel space driver part.

To invoke the graph execution the `SEQ_DRV_GraphStart`() function has to be called. This function has two parameters to setup number of frames to grab and override the number of expected lines per frame at the input of the frame. If the requested number of frames is set to 0 the graph execution continues until a `SEQ_DRV_GraphStop()` is invoked. If the specified number of input lines is 0, the value mentioned in the graph is used.

During the graph execution the graph data in CRAM are being updated. It is possible to create a copy of the current snapshot of the graph by invoking the `SEQ_DRV_GraphFetch()` function.

3.3.3Full-frame handling

The user application executed on the Host CPU is expected to work with the sensor data on a full-frame basis whereas the Sequencer graph running on ISP HW works with individual image lines. Framebuffers located in DDR are the only supported way to exchange full image data between the Sequencer graph and the user application.

In the Sequencer graph the access to the DDR buffers is defined in a FDMA node. The FDMA HW supports N independent channels to be used for DDR->SRAM or SRAM->DDR data transfers. Each channel is described by a TransferDescriptor (TD) structure stored in a table located in

SRAM. Content of a descriptor defines (among other transfer parameters) the source and destination buffer addresses, together with the line indexes to maintain current location in the buffers.

The separate line transfers can be scheduled by the Sequencer FW directly from M0 core and the FDMA HW itself supports an auto-update feature to update the current buffer line indexes in the appropriate TD in SRAM. This way the per line preprocessing is entirely independent of the Sequencer driver, thus saving the Host CPU resource.

The Sequencer graph can generate/consume several independent frame sequences (streams) at a time (e.g.: downscaled grayscale image intended for vision processing, together with RGB HD for display). To enable the use of these streams the driver exports the following functions that implement a mechanism similar to standard V4L (Video for Linux) interface.

The user application is responsible for allocating the contiguous DDR buffers. Then the driver has to be informed that the buffers were prepared for a particular stream by filling in a `SEQ_BufferRequest_t` structure and calling `SEQ_FrmBuffersRequest()`. The driver generates `SEQ_FrmStream_t` structure for internal handling of the buffers. This structure includes two queues:

- "Work queue" – buffers in this queue are or will be provided to the Sequencer to write/read the data.
- "Done queue" – buffers that have been finished by the Sequencer.

At the beginning both queues are empty and all the buffers are set to "User state". The buffer lifecycle is depicted in Figure 3. It can assume the following states:

- "User state" – the buffer is safe to be used by the user application until pushed back to the work queue by calling `SEQ_FrmBufferPush()`.
- "Work queue" – the buffer is scheduled to be used by the Sequencer and must not be accessed from the user application. Otherwise image artefacts can be observed caused by the Sequencer/Host simultaneous access.
- "Done queue" – the buffer was finished by the Sequencer and is ready to be made available to the user application after calling `SEQ_FrmBufferPop()`.

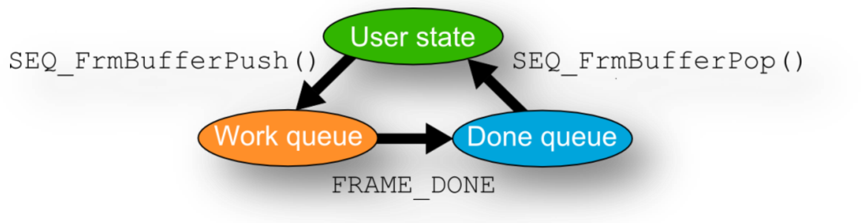

Figure 3: Framebuffer lifecycle

To address a specific buffer from a user application a `SEQ_FrmBufferInfo_t` structure has to be used. To get the state and size information for any buffer a `SEQ_FrmBufferQuery()` function can be called provided the BufferInfo structure with prefilled stream and buffer index.

To enable the data grabbing all the buffers are supposed to be initially queried (to confirm the buffer exists) and then pushed to the "Work queue". If there is at least one buffer in the "Work queue" the Sequencer processing can be initiated.

The transition of a buffer from Work to Done queue will happens if there was a FRAME_DONE message received from the Sequencer FW and there are at least two buffers in the Work queue at the moment. Once the Graph processing has been started the Work queue has to always contain at least one frame. Because of this it is recommended to keep at least two frames to be ready in the Work queue. From this perspective the minimum recommended number of buffers per stream is 3 but 4 or more buffers should provide better stability.

3.4 User Space

The Sequencer driver SW includes a thin user space library to abstract kernel space driver from user space. The diagram in Figure 4 depicts sequence of user space library function calls required for a typical CSI camera frame grabbing. The available data types have been mentioned in section 3.2 and functions exported by the user space library are listed in Figure 4. For more information please refer to [3].

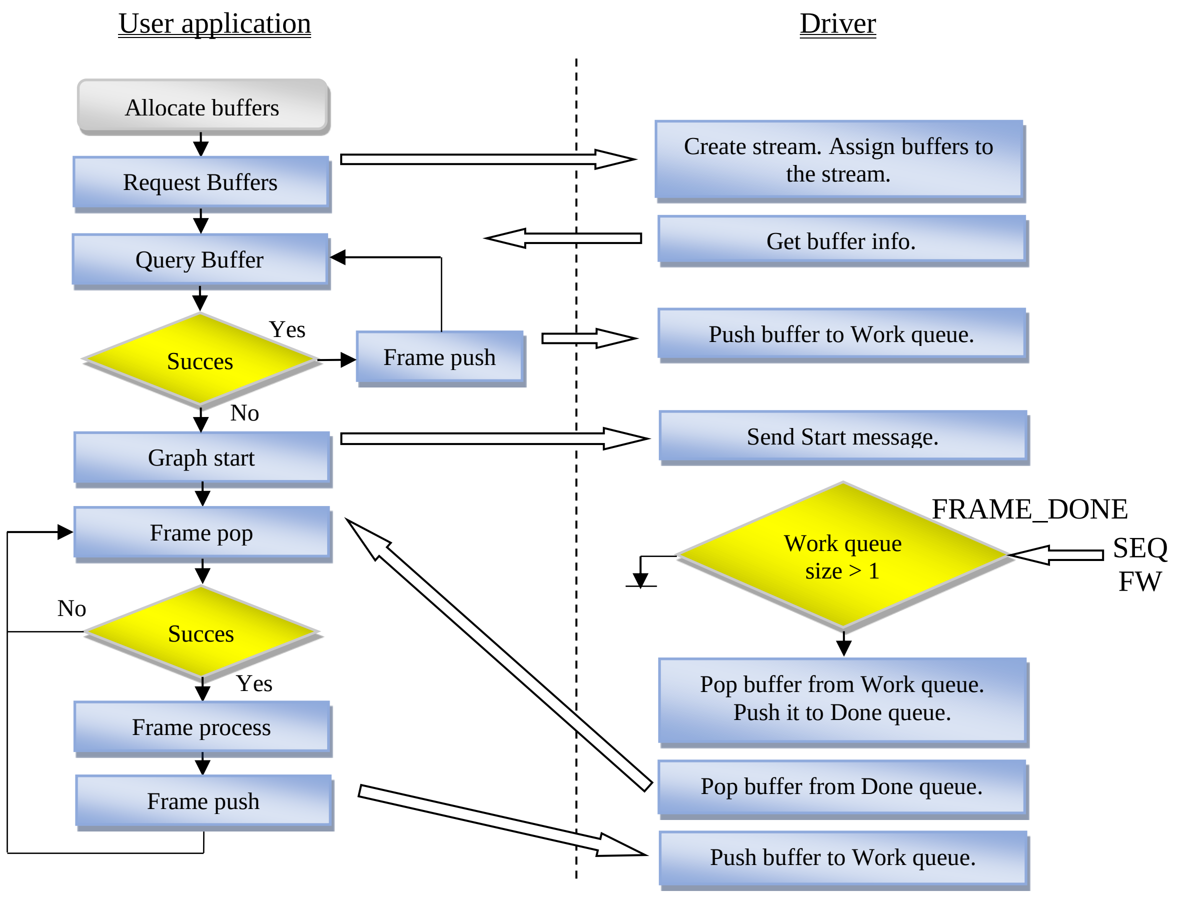

Figure 4: Buffer stream workflow

3.4.1API

Besides the user-space counter parts of `Reset()`, `Boot()`, `MsgSend()`, `GraphDownload()`, `PatchListSet()`, `GraphStart()`, `GraphStop()`, `GraphFetch()`, `PramAuxDataGet()`, `FrmBuffersRequest()`, `FrmStreamsReset()`, `FrmBufferQuery()`, `FrmBufferPush()`, `FrmBufferPop()`, `RegListSet()`, `RegListGet()`, `IpuPerformanceInfoRequest()`, `IpuPerformanceInfoGet()`, `EventLogEnable()`, `EventLogDisable()`, `EventLogGet()`, `IpuHistogramEnable()`, `IpuHistogramDisable()`, `IpuHistogramGet()`, `IpuStatEnable()`, `IpuStatDisable()`, `IpuStatGet()`, `TimeStatsGet()` functions, the Sequencer user library exports functionality as summarized in the Table 4. Please see [3] and `isp_seq.h` for implementation details.

Function	Description
`SEQ_Reserve`	Opens Sequencer special device file on Linux or calls `SEQ_Setup` in case of standalone environment.
`SEQ_Release`	Closes Sequencer special device file on Linux or calls `SEQ_Close` in case of standalone environment.
`SEQ_GraphAddressGet`	Gets the offset from PRAM base address at which the graph will be stored – required for successful graph packing.
`SEQ_FwArrPreProc`	Downloads the specified firmware
`SEQ_FwFileDownload`	Downloads the specified FW file to appropriate memory region
`SEQ_FwArrDownload`	Downloads the specified FW code to appropriate memory region.
`seq_swreset`	Instructs the Sequencer to commence SW reset.
`SEQ_EventHandlerSet`	Installs a user provided function or object to handle the Sequencer asynchronous events.
`SEQ_EventLogProccess`	Processes the provided ISP event log to a formated string that can be stored as csv file.
`seq_startNode`, `seq_checkNode`, `seq_setReg`, `seq_getReg`, `seq_echo`	Utility functions to start specific graph nodes execution, set/get IPU register values and send echo messages.
`ISP_LutSet`	Copies provided 16bit values to IPU LUT at specified offset. LUT is available only in IPUs 2 and 3. One IPU has 4k (4096) of 16bit values
`ISP_LutGet`	Copies 16bit values from IPU LUT at specified offset to array. LUT is available only in IPUs 2 and 3. One IPU has 4k (4096) of 16bit values

`ISP_DumpPerfCounters`	Prints out IPU performance counters.
`ISP_DumpIpuState`	Prints out IPU state
`ISP_PrintGraphStat`	Prints out Graph bufers statics

Table 4: Sequencer user library exported functions

3.4.2Firmware handling

Before the M0 can be booted and the ISP graph executed the various firmware parts have to be placed in proper memory locations. The ISP firmware includes the following parts:

- M0 binary defining the Sequencer SW behavior,
- ISP graph defining the image processing steps,
- IPUx kernels defining particular algorithms to be executed on specific IPU engines as part of the ISP graph image data preprocessing.

The Sequencer firmware including M0 binary and IPUx kernels is downloaded directly from user-space library. The `SEQ_FwArrDownload` function and its overloads are intended to be used for this purpose. Sequencer driver `memmap` functionality in case of Linux or direct memory access in case of standalone target are used to get access to the particular memory segments. In other hand, the Sequencer firmware and the graph data can be re-downloaded by using `sdi_process:Set()` function. This feature is very useful in case having two cameras and two graph with different setting and algorithm.

The ISP graph is currently being downloaded through call to `SEQ_GraphDownload` which passes the SDI preprocessed graph structure to the Sequencer driver kernel module that accomplishes the final copy of the data into CRAM.

3.4.3Event handling

The Sequencer user library provides mechanism for registering user defined functions as handlers for asynchronous events (frame done and error signaling) generated by the Sequencer HW block. For more details please refer to `SEQ_EventHandlerSet` functions in [2].

3.5 Sequencer user level workflow

From the user application point of view there are two ways to use the Sequencer functionality:

- through the SDI library. This is a preferred method.
- directly through the Sequencer driver user-space library.

To enable the use of the Sequencer driver services on Linux the Sequencer kernel module has to be loaded (`insmod` command). The Sequencer driver module has to be loaded prior to FDMA driver to have exported symbols available.

When the `insmod` command has succeeded the user applications can begin to use the Sequencer HW accelerated data preprocessing.

To start using the Sequencer driver an application has to call `SEQ_Reserve` function. On Linux this function opens the driver special device file and stores its descriptor for future use. On standalone environment the `SEQ_Setup` function is called directly. If succeeded the application must provide the firmware for M0, IPU code and Sequencer graph by invoking the `SEQ_FwDownload` function. Also, DDR buffers for images have to be provided by calls to the `SEQ_FrmBuffersRequest`, `SEQ_FrmBufferQuery`, `SEQ_FrmBufferPush` (see 3.3.2).

After that the Sequencer can be booted up using the sequence of calls to `SEQ_Reset()` and `SEQ_Boot()` functions. To start the graph execution a `SEQ_GraphStart` call has to be triggered.

Now the sensor data preprocessing pipeline has been executed and is controlled by the M0 core according to the graph content. The Host CPU (A53) interaction is required only on a frame basis mostly for DDR buffer related management (fDMA TD updates), exposure control loop or other tasks that cannot be accomplished by the M0 itself.

While the graph is running the application can call `SEQ_FrmBufferPop` function to get exclusive access to finished images and use `SEQ_FrmBufferPush` to return the no longer need buffers back to the driver to be reused for data storage.

To end the Sequencer graph execution `SEQ_GraphStop()` should be used to request graceful stop of the graph execution after the current image has been finished.

There is also a `SEQ_Reset()` function available to force reset of the Sequencer HW and driver internal data. To get performance related data the `SEQ_IpuPerformaceInfoRequest/Get` function can be used.

For full function API description please refer to [2].

4 High Level Design

4.1 System Decomposition

The Sequencer HW and its driver belong to the complex data preprocessing subsystem of the s32v234 SoC that is wrapped and controlled by the SDI library. Part of this subsystem is visualized in Figure 1. For more information about SDI and data preprocessing please refer to [3].

The preferred way to use the Sequencer functionality in a user application is to use Sequencer graphs together with the SDI library services. The SDI library provides complete abstraction of the Sequencer driver interface and thanks to utilization of the Sequencer HW the data flow management load for the host CPU is minimized.

4.2 File Structure

Sequencer driver code is located in VSDK under s3234_sdk/libs/isp/sequencer folder. Internally it has the following structure:

- kernel
    - build-v234ce-gnu-linux-d – build folder for Linux kernel module
        - Makefile
    - include
        - seq_func.h – declaration of Sequencer driver functionality
        - seq_lldcmd.h – declaration of LLDCMD codes
        - seq_sa_kernel_api.h – definition of Linux kernel symbols missing in standalone environment to facilitate bare metal build.
        - seq_types.h – declaration of Sequencer related data types
        - seq.h – general Sequencer related declarations/definitions
        - vdlist.h – definition of double linked list of void*
    - src
        - seq_core.c – Linux module related functionality
        - seq_func.c – definition of the Sequencer driver functionality
        - seq_lldcmd.c – definition of LLDCMD handling
- user
    - build-* – build folders for supported platforms (standalone and Linux)
        - Makefile
    - src
        - seq_user.cpp – definition of user space level public API,
    - BUILD.mk – defines build details
- Public headers (s32v234_sdk/include)

- isp_seq.h – declaration of user space level public API,

4.3 Module Usage

< This section will contain module usage hints and examples.>