FastDMA Driver Software User Guide

ABSTRACT:
This is the Software User Guide Document for fDMA driver of the Vision SDK
KEYWORDS:
FDMA, SRAM, Linux, SDI, Sequencer, User guide,
APPROVED:

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	04-May-16	Tomas Babinec	Initial version.
0.2	10-August-16	Tomas Babinec	Fixed review findings.
1.0	16-May16	Tomas Babinec	Updated for VSDK 1.0 RTM release
1.1	19-January-18	Tomas Babinec	Updated to LLDCMD
1.2	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 fast DMA driver user space interface. It is intended to serve as a reference source during VSDK based application development. For exact definitions and implementation details please refer to [2].

1.2 Scope and Objective

This document includes User Guide in the scope of the s32v234 project.

1.3 Audience Description

This document is intended for s32v234 Vision SDK users.

1.4 References

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

Table 1: References Table

1.5 Definitions, Acronyms, and Abbreviations

Term/Acronym	*Description*
API	*Application Programming Interface*
CRC	*Cyclic Redundancy Check*
DDR	*Double Data Rate DRAM*
DRAM	*Dynamic Random Access Memory*
fDMA	*fast Direct Memory Access HW block*
HW	*Hardware*
IP	*Intellectual Property*

ISP	*Image signal processor (whole image processing system)*
SDI	*Sensor Data Interface library*
Sequencer	*M0+ based micro controller managing the ISP engines' events*
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 Table

1.6 Document Location

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

2 General Description

The FastDMA (fDMA) driver software (SW) is intended for kernel space management of the FastDMA 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 (LLDCMDs, etc.).

2.1 Terminology

To facilitate the reader's comprehension of this document, this chapter describes several most important terms that are being used in a consistent manner throughout the document with a specific meaning as explained below.

- "**Host**" or "**Host CPU**" = the main computational unit in the s32v234 SoC (Arm A53).
- **"LLDCMD"** = Low-Level Driver CoMmanD. Provides a way to invoke low-level (kernel) driver functionality from user-space library. It is implemented in OAL library by `OAL_LldCmd()` function with `OAL_LldCmd_t*` attribute.
- "**Transaction**" = a configurable data operation that can be scheduled by the fDMA driver. A Transaction is defined by source address, destination address and amount of the data in bytes. To schedule a Transaction means that the driver writes to the fDMA HW registers to physically queue data operation (data movement, CRC computation or both).
- "**Transfer**" = a series of logically related transactions. For instance to accomplish a Transfer of whole image data a series of Transactions, each working with only one or several (for example 3) lines, has to be scheduled.

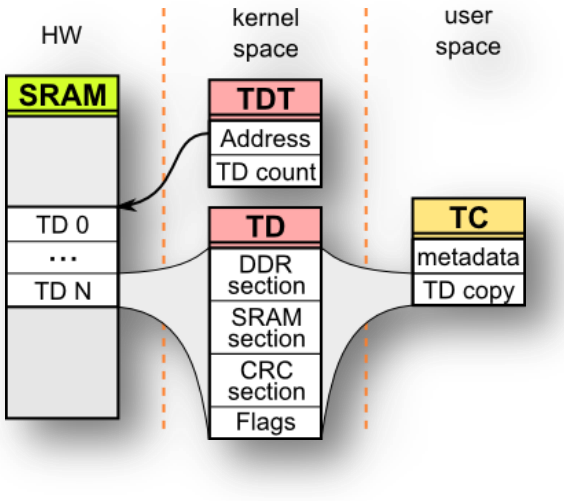

Figure 1: Data Transfer related entities: TD = Transaction Descriptor, TDT = TD Table, TC = Transfer Channel

- "**Transaction Descriptor (TD)**" = a kernel space structure encapsulating one entry in the SRAM Transaction descriptor table (see Figure 1). Contains values (source and destination

addresses, etc. – see [1]) that are necessary to describe one data transaction. Transaction descriptors are equivalent to transfer records in [1].

- "**Transfer channel (TC)**" = a user space type representing one configurable channel for the data flow between SRAM and DRAM. Contains metadata about the corresponding TD (see chapter 3.2.1) and its user space copy (see Figure 1). The same TC can be used to accomplish sequence of several Transactions or even independent Transfers.
- "**Transaction descriptor table (TDT)**" = a kernel space structure encapsulating information (address, number of descriptors) about an array of Transaction descriptors located in SRAM (see Figure 1). In some cases refers directly to the array of Transaction descriptors in SRAM.
- "**Slice**" = a portion of data cut from a data stream (for instance one or multiple lines from an entire image). Also can refer to a memory where such data are stored. Slices in SRAM will be usually stored in ping-pong or circular buffers. For example see Figure 2.

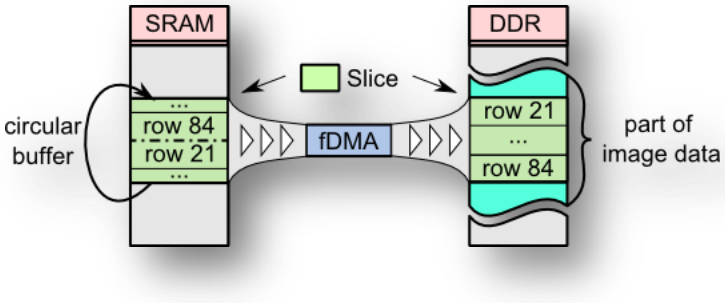

Figure 2: Slice of data (fDMA reads image data from a slice in the SRAM circular buffer and constructs a whole frame in the DDR)

2.2 FastDMA HW

The fDMA HW was designed for s32v234 SoC, which is aimed at vision ADAS applications. The involved image processing requires high bandwidth data transfers and large memory areas to be available for the data processing and storage. To achieve this, the data being processed are distributed between two distinct memory blocks:

- SRAM with high speed access but limited memory space (4MB) allowing only the immediate input/output data (several image lines at most) to be stored there;
- DDR with much lower bandwidth but enough space to store series of whole image frames.

To use this setup efficiently, the data has to be in constant flow between the SRAM and DDR. The fDMA HW is specifically designed to accomplish the actual data transfers thus significantly reducing the possible load for the host CPU and Sequencer HW.

To help ensure functional safety the fDMA HW supports CRC data protection that can be applied to both SRAM and DDR data without the need to actually write the data to different location.

Figure 3 shows fDMA HW related interactions. The Sequencer HW block has an important role here. In general the Sequencer is dedicated to the management of the image data preprocessing pipeline. Among other features, the Sequecer has the ability to schedule fDMA transactions and receives the transaction done signaling.

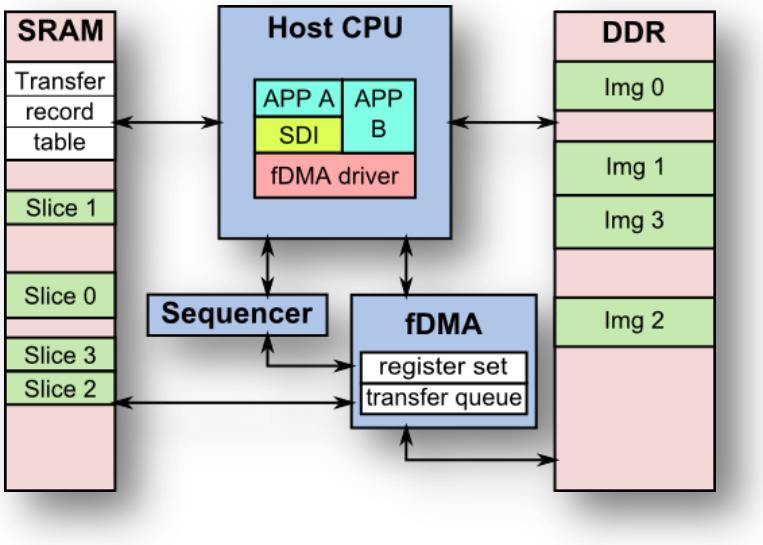

Figure 3: FastDMA HW interactions

Based on the Sequencer HW block activity the s32v234 vision related subsystem (HW & SW) can be configured to operate in two distinct modes:

- **The Sequencer-based mode**
- In this mode, the host CPU (A53 core cluster) will initiate all transactions through the Sequencer messaging interface. The host does only the initial set-up of the FastDMA HW block. After that only the Sequencer is allowed to access the fDMA HW registers to rule out race conditions between the Sequencer and the host CPU.
- The Sequencer functionality is available and its driver is loaded. This is considered the preferred mode of operation. In this operation mode, the Sequencer will also be responsible for mediating the transaction scheduling and event signaling between host CPU and fDMA HW, which will be routed through the Sequencer block driver (sending/receiving messages through the Sequencer parameter RAM). Please see Figure 4 – blue control paths.
- This approach will allow the host CPU to mask/ignore the fDMA IRQs and process only Sequencer messages. Since the host driven fDMA transfers are expected to be rare compared to the Sequencer driven ones, the host CPU load caused by the fDMA IRQ filtering and handling will be reduced significantly.

  NOTE: Only ISP graph related FDMA transactions are implemented. Any requests for host CPU defined transactions will be refused by the driver in Sequencer- based mode as the support for this feature was not implemented.

- **The Sequencer-less mode**

  In this mode the host CPU will initiate fDMA transactions by directly using the fDMA HW.

  The Sequencer HW is powered down or its driver is not loaded. This is intended mostly for testing or debugging purposes. The host CPU is responsible for the full management of the

fDMA HW block: scheduling all transactions and handling all fDMA events (IRQs). Please see Figure 4 – red control paths.

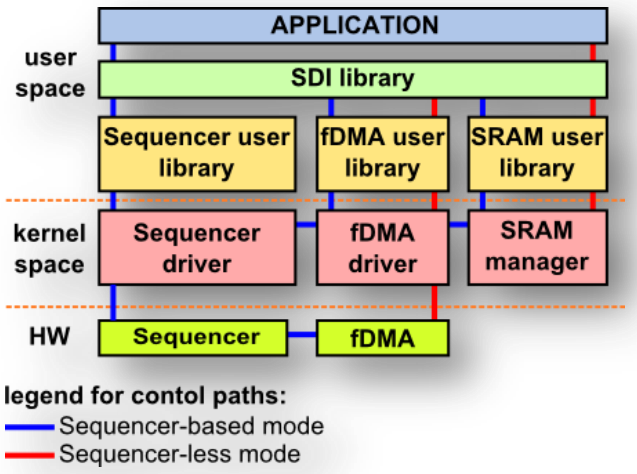

Figure 4: FastDMA SW layout & interactions

From the SW point of view it is expected, the fDMA driver will be used mainly by the Sensor Data Interface (SDI) library [3] (see Figure 4) to accomplish image data transfers between SRAM and DDR memory blocks. In such case the fDMA functionality is abstracted completely by the SDI library and user applications do not have to access the underlying SW layers directly. Therefore the fDMA driver design is aimed at accommodating the SDI needs as much as possible.

On the other hand there has to be a possibility for the user applications to directly utilize the fDMA user library services and operate the fDMA HW on their own.

3 Functional Description

The fDMA driver SW has two layers. The first layer operates in kernel space and accomplishes most of the driver's functionality. Internal behavior of the kernel space layer depends on the Sequencer HW availability and will be described in detail in section 3.2.

The second layer will be implemented as a user space library creating a thin interface for user application and facilitating communication between the applications and the kernel interface of the driver. The provided functionality is explained in section 3.3 and will be independent on current configuration of the kernel space part with regards to Sequencer availability.

3.1 FastDMA HW

The fDMA HW is designed to perform data transfers between SRAM and DDR. It is not capable to transfer data from SRAM to SRAM or DDR to DDR. Each transaction is described by a transaction descriptor stored in a transaction descriptor table (TDT) in the SRAM. The fDMA HW can also generate CRC values to accelerate functional safety algorithms (see section 3.4).

To get detailed information about all features, configuration and operation of the fDMA HW block please refer to [1].

3.2 FastDMA Driver – Kernel Space

The internal functionality of the fDMA kernel module and its user space interface make the fDMA HW features available for user applications.

3.2.1Data structures

The fDMA driver introduces the following data types and containers (relations depicted in Figure 1; please see [2] for full definitions):

- Structure `fdma_regs`:

  Mirrors content of the fDMA registers. For full listing and description of fDMA registers please see refer to [1].

- Structure `fdma_td`:

  Contains complete description of a data transaction between SRAM and DDR (see Figure 1).

- Structure `fdma_tdm`:

  Transaction descriptor metadata containing: index to the TDT, reserved or unused flag, owner process ID.

- Structure `fdma_tc`:

  Abstracts management of fDMA transfers for user space in a form of a transfer channel (TC - see Figure 1). Combines a copy of transaction descriptor metadata together with a copy of the corresponding transaction descriptor. Used for communication between user and kernel space.

- Structure `fdma_tc_event`:

Structure capturing the state of FDMA registers at the time of some asynchronous event being signaled from the HW (transfer done or error).

3.2.2API

The fDMA driver communicates with user space applications through predefined LLDCMDs (see Table 3) directed to the driver through a special device file (fdma).

Error conditions that appear inside of the LLDCMDs are signaled to user space through the return value of the LLDCMD function. Errors are signaled also in case that there are no more resources available at the moment (all TDs are used, the fDMA transaction fifo is full).

Command	Description
`FDMA_TD_RESERVE`	Reserves one transaction descriptor from TDT in SRAM **Output:** If succeeded, the transfer channel structure is returned back to user space through command parameter. Otherwise LLDCMD call returns error.
`FDMA_TD_CONFIG`	Writes configuration of one transaction descriptor to TDT in SRAM. New configuration values are passed to the driver in a `fdma_tc` data structure. **Output:** Success if the specified transfer channel index has been valid. Error otherwise.
`FDMA_TD_ISP_CONFIG`	Writes configuration of one transaction descriptor to TDT in SRAM. New configuration values are passed to the driver in a `fdma_tc` data structure. ISP graph transactions only. **Output:** Success if the specified transfer channel index has been valid. Error otherwise.
`FDMA_TD_SCHEDULE` **NOTE:** Sequencer-less mode only	The driver instructs fDMA HW to schedule a transaction specified by an index of a transaction descriptor. **Output:** LLDCMD returns error if invalid transaction descriptor index has been specified or the queue is full. Otherwise returns success.
`FDMA_TD_CONFIG_SCHEDULE` **NOTE:** Sequencer-less mode only	Sets new transaction configuration and schedules it immediately. **Output:** LLDCMD returns error if invalid transaction descriptor index has been specified or the queue is full. Otherwise returns success.
`FDMA_TD_RELEASE`	The specified transaction descriptor can be reserved again from now on. **Output:** If an invalid transaction descriptor handle is specified error is returned. Otherwise, including the transaction descriptor has been released before, success is returned.

FDMA_SEQ_MODE_GET	Checks in what mode the fDMA kernel module has been initialized with regards to the Sequencer HW availability. **Output**: 1 for Sequencer-based mode, 0 for Sequencer-less mode.
FDMA_STATUS_GET	Loads current fDMA status information (copy of fDMA HW register set) to user application. **Output:** The driver fills in current fDMA status information into the user specified fdma_status data structure and success is returned.
FDMA_EVENT_GET	Returns details for latest fDMA event related to specified TD index. **Output:** The driver fills in current fDMA Event information into the user specified FDMA_TcEvent_t data structure and success is returned. Fails in case invalid TD ID provided.
FDMA_TC_GET	Updates user space copy of transaction descriptor in the specified transfer channel structure. **Output:** Members of transaction descriptor in the given transfer channel instance are updated and returned to user space.
FDMA_TD_USED	Read current number of reserved transaction descriptors. Intended mainly for debug purposes. **Output:** Number of reserved transaction descriptors.
FDMA_TDT_GET	Get the HW base address of the TDT in SRAM. **Output:** HW base address of the TDT.

Table 3: fDMA driver LLDCMDs

3.3 FastDMA Driver – User Space Part

The fDMA driver SW includes a thin user space library to abstract kernel space driver from user space (as defined in section 3.2). All data types and functions exported by the user space library are listed in Table 4 and Table 5. For more information please refer to [2].

To enable the use of fDMA each application has to call `FdmaOpen()` function. This function opens the driver special device file and stores its descriptor for future use. If succeeded a new transfer channel can be initialized by the call to `FdmaTcReserve()` function.

After that a transaction has to be set up by configuring appropriate values of previously initialized transfer channel structure members. To transfer the configuration to fDMA HW the `FdmaTcConfigure()` function has to be called. Now the transaction can be scheduled by `FdmaTcSchedule()` function.

NOTE: Scheduling of fDMA transactions from host CPU in case of Sequencer-based mode is not supported.

The configuration and scheduling calls can repeat until all the data have been transferred. If the HW accelerated update was enabled in the transfer channel structure, the fDMA HW will automatically update transaction descriptor in SRAM after each transaction described by that particular TD has been finished. Therefore in such case the configuration step can be skipped at user side.

The user application is responsible for releasing the transfer channel if there are no data left to be transferred. This can be achieved by the call to `FdmaTcRelease()` function.

If the services of fDMA are no longer required by the application the `FdmaClose()` function should be called to close the fDMA special device file and allow the driver to be unloaded if required.

To fetch status information about the fDMA HW, latest transaction descriptors setting and event information the following functions can be used: `FdmaStatusGet()`,`FdmaTcGet()`, `FdmaEventGet()` as described in Table 5.

3.3.1Event handling

The fDMA user library provides a mechanism for registering user defined functions as handlers for asynchronous events (frame done and error signaling) generated by the fDMA HW block.

3.3.2API

The fDMA user library exports data types and functionality as summarized in the Table 4 and Table 5 respectively. Please see [2] for implementation details.

Data type	Description
`fdma_regs`	Full copy of fDMA HW register values.
`fdma_td`	Transaction descriptor structure as used for SRAM table.
`fdma_tdm`	Transaction descriptor metadata structure.
`fdma_tc`	Transfer channel structure. Contains `fdma_td` and `fdma_tdm` members.
`fdma_tc_event`	Event details describing structure.

Table 4: fDMA user library exported data types

Function	Description
`FdmaOpen`	Opens fDMA special device file which makes the fDMA functionality available to the user application.
`FdmaTcReserve`	Allocates one fDMA transfer channel structure instance and attempts to initialize it by reserving a free transaction descriptor. If successful the address of the new transfer channel instance is set to the user specified `fdma_tc` pointer.

FdmaTcConfig	Copies the specified transfer channel settings to appropriate structures managed in the driver. This effectively sets up future transaction parameters.
FdmaTcIspConfig	Copies the specified transfer channel settings to ISP transfer descriptor table in the driver. This effectively sets up future transaction parameters.
FdmaTcSchedule	Instructs fDMA HW to schedule specified data transaction.
FdmaTcRelease	Releases the specified transfer channel. This includes dealocating the structure and setting the pointer to NULL.
FdmaClose	Closes fDMA special device file which disables the fDMA functionality for this application.
FdmaStatusGet	Copies current fDMA HW register values to the user specified fdma_regs structure.
FdmaTcGet	Reads current content of the requested transaction descriptor and its metadata and updates the user specified fdma_tc structure.
FdmaEventGet	Fetches details about latest HW event related to specified transfer channel id.
FdmaSeqModeGet	Returns the Sequencer relation mode
FdmaTcUsed	Fetches the current number of reserved transfer channel
FdmaEventHandlerSet	Setup the handler when event of fDMA HW occur.
FdmaTdtAddrGet	Fetches the physical address of the transfer descriptor table in sram memory

Table 5: fDMA user library exported functions

3.4 FastDMA high level workflow

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

- through the SDI library by running the Sequencer graph that includes fDMA data transfers. This requires the Sequencer functionality to be available.
- directly through the fDMA driver user-space library.

Both of the approaches can be used at the same time.

To enable the use of the fDMA HW services the fDMA kernel module has to be loaded (insmod command). During the kernel module initialization, the OAL CMA allocator kernel module has to be available (to allocate SRAM space for TDT). Otherwise the fDMA module insertion will fail. At the same time the fDMA will check for the presence of the Sequencer driver module. If loaded, the

fDMA will be configured to Sequencer-based mode. Otherwise the Sequencer-less mode will be used.

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

3.4.1fDMA transfers in the Sequencer graphs

In case the fDMA transactions are executed as a part of some Sequencer graph the SDI library, which is required to manage the graphs, will hide the control of FDMA functionality. This includes reservation and configuration of transfer channels and allocation of required SRAM regions for the image data and potential CRC tables. Also the DDR buffers can be allocated automatically by the SDI although there will be a possibility for the user applications to manage the DDR buffers on their own.

If the transfer channel has been configured by the SDI and the graph has been executed, the fDMA data transfers are controlled directly by the Sequencer (transaction scheduling and DONE event handling). The exclusive Sequencer management of the graph related fDMA data transfers is interrupted when a whole image has been processed by the graph. At this point the SDI is notified by the Sequencer using a frame-done message.

3.4.2Host CPU initiated fDMA transactions

Apart from the Sequencer graphs the fDMA transaction can be managed directly by the Host CPU. An example of this is regular CRC validation of some data in DDR. Another example can be the use of h264 decoder from A53 alone.

Host CPU initiated fDMA transactions are supported and allowed only in Sequencer-less operational mode (ISP Sequencer functionality not enabled).

3.4.3CRC computation and validation

The fDMA HW is capable to compute CRC values (32 bits wide) for the data being transferred. The CRC values are generated on a transaction basis – one for each transaction. Each transaction can be configured to generate one CRC value using appropriate TD fields:

- `CRC_ENA`

  If set to 1 the CRC computation is enabled.

- `CRC_START_ADDR`

  Contains the start address of the CRC table for this particular TD. The table has to be located in SRAM. User application is responsible for allocation of required space

- `CRC_MODE`

  Used to choose between 4 different CRC operation modes:

  - 0: CRC value is calculated and written only to `FASTDMA_CALC_CRC_VAL` register.
  - 1: CRC value is calculated. If the transfer direction is SRAM->DDR (`DIR` field in TD is set to 0) the value is written to the CRC table specified by the `CRC_START_ADDR` TD field. In case of the DDR->SRAM direction (`DIR` field in

TD is set to 1), the computed value is validated against corresponding entry in the CRC table.

- 2: CRC value is calculated and always written to the CRC table in SRAM. The direction of the transfer has no effect.
- 3: CRC value is calculated and always validated against corresponding entry in the CRC table. The direction of the transfer has no effect.

- `CRC_POLY`

  Selects the polynomial type for the CRC value calculation.

  - 0: the Castagnoli (AUTOSAR) polynomial is used to calculate the CRC.
  - 1: the Koopman polynomial is used to calculate the CRC.

It is possible to use the CRC functionality without actually transferring the data if the WR_ENA bit in TD is set to 0. Please refer to [1] for complete set of CRC use-case scenarios.

NOTE: Scheduling of fDMA transactions from host CPU including CRC functionality is supported by the SW driver only in Sequencer-less mode.

4 High Level Design

4.1 System Decomposition

The fDMA 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 3. For more information about SDI and data preprocessing please refer to [3].

The preferred way to use the fDMA functionality in a user application is to use Sequencer graphs together with the SDI library services. The SDI library provides complete abstraction of the fDMA 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

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

- kernel
  - build-v234ce-gnu-linux-d – build folder for Linux kernel module
    - Makefile
  - include
    - fdma_func.h – declaration of FDMA driver functionality
    - fdma_lldcmd.h – declaration of LLDCMD codes
    - fdma_types.h – declaration of FDMA related data types
  - src
    - fdma_core.c – Linux module related functionality
    - fdma_func.c – definition of the FDMA driver functionality
    - fdma_lldcmd.c – definition of LLDCMD handling
- user
  - build-* – build folders for supported platforms (standalone and Linux),
    - Makefile
  - src
    - fdma_user.cpp – definition of user space level public API,
  - BUILD.mk – defines build details
- Public include
  - isp_fdma.h – declaration of user space level public API.

4.3 Module Usage

< This section contains module usage.>