BIOSPSP LCDC Raster Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Madhvapathi Sriram	February 9, 2009	Created the document

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments

Post Office Box 655303

Dallas, Texas 75265

Table of Contents

List Of Figures

1 System Context

The purpose of this document is to explain the device driver design for LCDC Raster peripheral using DSP/BIOS operating system running on DSP OMAPL138

Note: The usage of structure names and field names used throughout this design document is only for indicative purpose. These names shall not necessarily be matched with the names used in source code.

1.1 Terms and Abbreviations

Term	Description
API	Application Programmer's Interface
CSL	TI Chip Support Library – primitive h/w abstraction
IP	Intellectual Property
ISR	Interrupt Service Routine
OS	Operating System

1.2 Disclaimer

This is a design document for the UART driver for the DSP/BIOS operating system. Although the current design document explain the UART driver in the context of the BIOS 6.x driver implementation, the driver design still holds good for the BIOS 5.x driver implementation as the BIOS 5.x driver is a direct port of BIOS 6.x driver. The BIOS 5.x drivers conform to the IOM driver model whereas the BIOS 6.x drivers confirm to the IDriver model. The subsequent section explains how this document can be used to understand and modify the IOM drivers found in this product. Please note that all the flowcharts, structures and functions described here in this document are equally applicable to the UART driver 5.x

1.3 IOM driver Vs IDriver

The following are the main difference between the BIOS 5.x and BIOS 6.x driver. Please refer to the reference documents for more details in the IOM driver model.

1. All the references to the stream module should be treated as references to a module that provides data streaming. In BIOS 5.x the equivalent modules are SIO and GIO.
2. This document refers to the IDriver model supported by the BIOS 6.x.All the references to the IDriver should be assumed to be equivalent to the IOM driver model.

3. The BIOS 6.x driver uses a module specifications file (*.xdc) for the declaration of the enumerations, structures and various constants required by the driver. The equivalent of this xdc file is the header file XXX.h and the XXXLocal.h.

   Note: The XXXLocal.h file contains all the declaration specified in the "internal" section of the corresponding xdc file.

4. In BIOS 6.x creation of static driver instances follow a different flow and cause functions in module script files to run during build time. In BIOS 5.x creation of driver (for both static and dynamic instances) result in the execution of the mdBindDev function at runtime. Therefore any references to module script files (*.xs files) can be ignored for IOM drivers.

5. The XXX_Module_startup function referenced in this document can be ignored for IOM drivers.

6. IOM drivers have an XXX_init function which needs to be called by the application once per driver. This XXX_init function initializes the driver data structures. This application needs to call this function in the application initialization functions which are usually supplied in the tci file.

7. The functionality and behavior of the functions is the same for both driver models. The mapping of IDriver functions to IOM driver functions are as follows:

IDriver	IOM driver
XXX_Instance_init	mdBindDev
XXX_Instance_finalize	mdUnbindDev
XXX_open	mdCreateChan
XXX_close	mdDeleteChan
XXX_control	mdControlChan
XXX_submit	mdSubmitChan

8. All the references to module wide config parameters in the IDriver model map to macro definitions and preprocessor directives (#define and #ifdef etc) for the IOM drivers. e.g.

   a. XXX_edmaEnable in IDriver maps to –D XXX_EDMA_ENABLE for IOM driver.

   b. XXX_FIFO_SIZE in IDriver maps to #define FIFO_SIZE in IOM driver header file

9. In BIOS 6.x a cfg file is used for configuring the BIOS and driver options whereas in the BIOS 5.x the “tcf” and “tci” files are used for configuring the options.

10. In BIOS 5.x driver support for multiple devices is implemented as follows. A chip specific compiler define is required by the driver source files (-DCHIP_OMAPL138). Based on the include a chip specific header file (e.g soc_OMAPL138) which contains chip specific defines is used by the driver. In order to support a new chip, a new soc_XXX header file is required and driver sources files need to be changed in places where the chip specific define is used

1.4 Related Documents

1.	TBD	DSP/BIOS Driver Developer’s Guide
2.	SPRUFM0	LCDC Raster User Guide

1.5 Hardware

The LCDC Raster device driver design is in the context of DSP/BIOS running on DSP OMAPL138 /C64P core.

The LCDC module core used here has the following bocks:

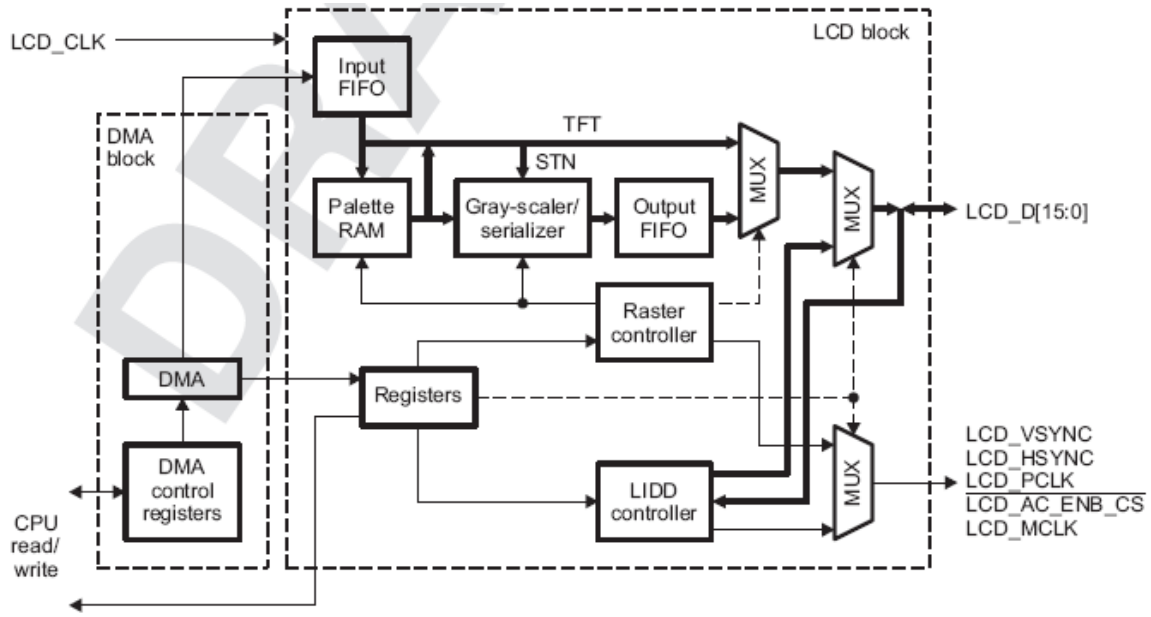

Figure 1 LCDC Block Diagram

1.6 Software

The LCDC Raster driver discussed here is intended to run in DSP/BIOS™ V6.10 on the OMAPL138 DSP.

1.6.1 Operating Environment and dependencies

Details about the tools and DSP/BIOS versions that the driver is compatible with, can be found in the system Release Notes.

1.6.2 System Architechture

The block diagram below shows the overall system architecture.

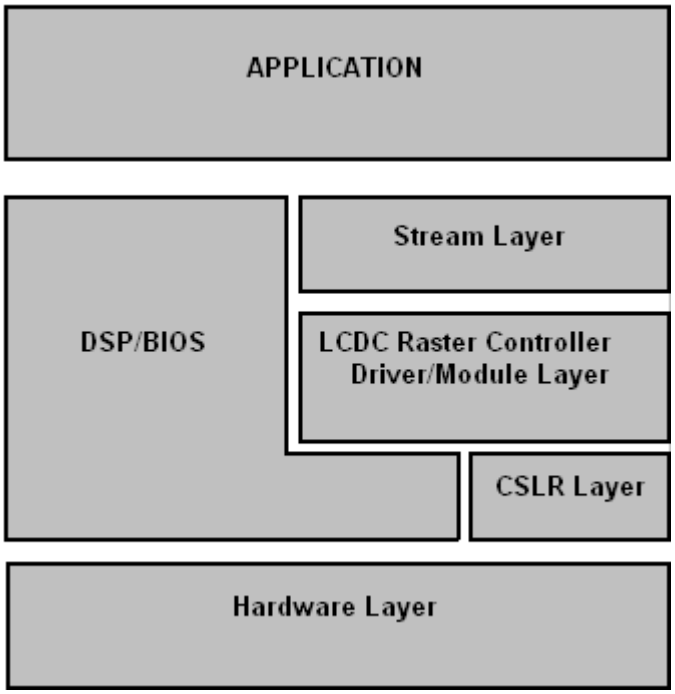

Figure 2 System Architecture

Driver module which this document discusses lies below the steam layer, which is a class driver layer provided by DSP BIOS™ (please note that the stream layer is designed to be OS independent with appropriate abstractions).The LCDC Raster Controller driver would use the rCSL (register overlay) to access the Hardware and would use the DSP BIOS™ APIs for OS services.

Also as the IDriver is supposed to be a asynchronous driver (to the stream layer), we plan not to use semaphores in IO path.

The Application would use the Stream APIs to make use of driver routines.

Figure 2 shows the overall device driver architecture. For more information about the IDriver model, see the DSP/BIOS™ documentation. The rest of the document elaborates on the architecture of the Device driver by TI.

1.7 Component Interfaces

In the following subsections, the interfaces implemented by each of the sub-component are specified. The LCDC Raster driver module is object of IDriver class. One may need to refer the IDriver documentation to access the LCDC Raster controller driver in raw mode or could refer the stream APIs to access the driver through stream abstraction. The structures and configuration parameters used would be documented in CDOC format as part of this driver development.

1.7.1 IDriver Interface

The IDriver constitutes the Device Driver Manifest to Stream (and hence to application). This LCDC Raster driver is intended to an XDC module and this module would inherit the IDriver interfaces. Thus the LCDC Raster driver module becomes an object of IDriver class. Please note that the terms “Module” and “Driver or IDriver” would be used in this document interchangeably.

As per XDC specification, a module should feature a *.xdc file, *.xs file and souce file as a minimum.

Raster module specifications file (Raster.xdc)

The XDC file defines the following in its public section: the data structures, enumerations, constants, IOCTLS, error codes and module wide configuration variables that shall be exposed for the user.

These definitions would include

ENUMS: Operation modes, driver states, output interface/format etc.

STRUCTURES: Event statistics, Channel/Device parameters, Frame Buffers etc

CONSTANTS: error ids, ioctls, minimum maximum values cor various settings etc.

Also this files specifies the list of configurable items which could be configured/specified by the application during instantiation (instance parameters)

In its private section it would contain the data structures, enums, constants and module wide confiugration variables. The Instance object (the driver object) and channel object contain all the information related to that particular IO channel. This information might be irrelevant to the user. The instance object is the container for all driver variables, channel objects etc. In essence, it contains the present state of the instance being used by the application

The XDC framework translates this into the driver header file (Raster.h) and this header file shall be included by the applications, for referring to any of the driver data structures/components. Hence, XDC file contains everything that should be exposed to the application and also accessed by the driver.

Please note that by nature of the specification of the xdc file, all the variables (independent or part of structure) need to be initialized in xdc file itself.

Raster module script file (Raster.xs)

The script file is the place where static instantiations and references to module usage are handled. This script file is invoked when the application compiles and refers to the Raster module/driver. The Raster.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg for the Raster module, the module use function in the Raster.xs file is used to initialize the hardware instance specific details like base addresses, interrupt numbers, frequency etc. This data is stored for further use during instantiation. This gives the flexibility to design, to handle multiple SOC with single c-code base (as long as the IP does not deviate).

2. Handling static instantiation of the Raster instance

   When the Raster instance is instantiated statically in the application cfg file, (please note that dynamic instantiation is also possible from a C file) the instance static init function is called. If a particular instance is configured with set of instance parameters (from CFG file) they are used here to configure the instance state. The instance state is populated based on the instance number, like the default state of the driver, channel objects etc.

The Raster IDriver module implements the following interfaces

S.No	IOM Interfaces	Description
1	Raster_Module_startup()	Register interrupts, configure hardware and initialization needed before opening a channel. Effectively makes the Raster module ready for use. This function is called at least once for sure when the Raster IDriver module is used. Hence these tasks are done here for all the instances that can be created.
2	Raster_Instance_init()	Handle dynamic calls to module instantiation. This shall be a duplication of the tasks done in

		instance static init in the module script file.
3	Raster_Instance_finalize()	Unregister interrupt, reset hardware and driver state and all de-initialization goes here. Effectively removes the usage of Raster instance.
4	Raster_open ()	Creates a communication channel in specified mode to communicate data between the application and the Raster module instance.
5	Raster_close ()	Frees a channel and all its associated resources.
6	Raster_control ()	Implements the IOCTLS for Raster IDriver module. All control operations go through this interface.
7	Raster_submit ()	Submit an I/O packet to a channel for processing. Used for data transfer operations. Internally handles different mode of operation and asynchronous mode of communication.

1.7.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the Raster IDriver module to configure Raster Controller registers. CSLR is implemented as a header file that has CSLR macros and register overlay structure.

1.8 Design Philosophy

This device driver is written in conformance to the DSP/BIOS IDriver model and handles communication to and from the Raster hardware.

1.8.1 The Module and Instance Concept

The IDriver model, conforming to the XDC framework, provides the concept of the *module and instance* for the realization of the device and its communication path as a part of the driver implementation.

The module word usage (Raster module) refers to the driver as one entity. Any detail, configuration parameter or setting which shall apply across the driver shall configure the module behavior and thus shall be a module variable. However, there can also be module wide constants. For example, setting to capture event statistics is a module wide variable and can be set by the application.

This instance word usage (Raster instance 1) refers to every instantiation of module due to a static or dynamic create call. Each instance shall represent an instance of device directly by holding info like the Raster channel handles, device configuration settings, hardware configuration etc. This is represented by the Instance_State in the Raster module configuration file.

Note that the LCDC hardware instance has two parts. One is the raster controller and the other the LIDD controller. Since, these two modes of LCDC cannot co-exist, the LCDC driver is split into two separate parts – Raster Module/Driver and the LIDD module/Driver. This brings in simplicity in design and usage of the driver module. By the virtue of this design the Instance is a direct mapping to the LCD Controller and the channel is a direct mapping to the LCDC Raster controller part of it.

Also, the Raster Controller module is a driver for displaying images onto a Raster LCD. By virtue of this there is only an output stream (channel) possible on this driver.

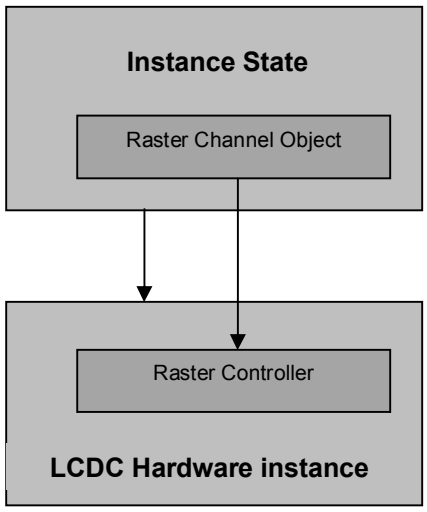

Figure 3 Instance Mapping

Also, every module shall only support as many number of instantiations as the number of LCDC hardware instances on the SoC

1.8.2 Design Constraints

Raster IDriver module imposes the following constraint(s).

- The LCDC controller is provided with an independent DMA unit. This is to facilitate faster and exclusive DMA transfers of streaming data to the hardware. Hence, the LCDC driver does not depend on the any other DMA peripheral driver or transfers. Also, the driver shall support only the DMA mode of operation

- The LCD controller supports two types of peripherals. One is the Raster type and the other is the LIDD type. However, since the two cannot be used simultaneously, the design approach is two have two independent driver modules for these mode of operation – Raster module and the LIDD module respectively.

2 Raster Driver Software Architecture

This section details the data structures used in the Raster IDriver module and the interface it presents to the Stream layer. A diagrammatic representation of the IDriver module functions is presented and then the usage scenario is discussed in some more detail.

Following this, we'll discuss the dynamic view of the driver where the driver operational scenarios are presented.

2.1 Static View

2.1.1 Functional Decomposition

The driver is designed keeping a device, also called instance, and channel concept in mind.

This driver uses an internal data structure, called channel, to maintain its state during execution. This channel is created whenever the application calls a Stream create call to the Raster IDriver module. The channel object is held inside the Instance State of the module. (This instance state is translated to the Raster_Object structure by the XDC frame work). The data structures used to maintain the state are explained in greater detail in the following *Data Structures* sub-section. The following figure shows the static view of Raster driver.

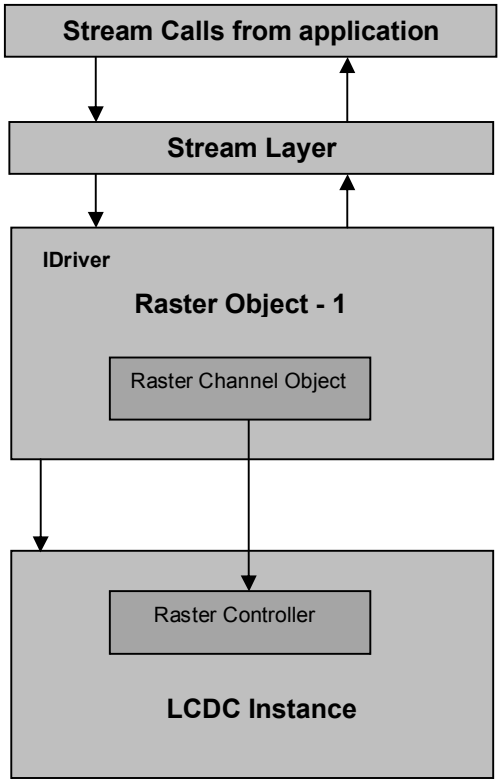

Figure 4 Raster Driver Static View

2.1.2 Data Structures

The IDriver employs the Instance State (Raster_Object) and Channel Object structures to maintain state of the instance and channel respectively.

In addition, the driver has two other structures defined – the device parameters and channel parameters. The device parameters structure is used to pass on data to initialize the driver during module start up or initialization. The channel parameters structure is used to specify required characteristics while creating a channel. For current implementation channel parameters contain the controller type, Raster configuration etc.

The following sections provide major data structures maintained by IDriver module and the instance.

2.1.2.1 The Instance Object (Raster_Object)

The instance state comprises of all data structures and variables that logically represent the actual instance on the hardware. It preserves the communication channels, parameters for the instance etc. The handle to this is sent out to the application for access when the module instantiation is done via create statically (application CFG file) or dynamically (C file during run time). The parameters that are to be passed for this call are described in the section Device Parameters.

S.No	Structure Elements *(Raster_Object)*	Description
1	*chCount*	Preserves the number of channels opened on this instance
1	*instNum*	Preserves instance number of LCDC Raster hardware
3	*type*	Preserves the device type. Currently only device type LCDC is supported
4	*state*	Preserves the current state of the instance/driver
5	*devConf*	Preserves the device configuration as passed by the application during instantiation
6	*eventStat*	Preserves the statistics of the Raster/LCDC events that have occurred. However, for event statistics collection the module needs to be configured with captureEventStatistics enabled
7	*ChannelObj*	The logical channel object that refers to the raster controller. This is described below.
8	*instHwInfo*	This preserves the hardware details like event number, base address etc, obtained from soc.xs

2.1.2.2 The Channel Object

The interaction between the application and the device is through the instance object and the channel object. While the instance object represents the actual hardware instance, the channel object represents the logical connection of the application with the driver (and hence the device) for that particular IO direction. It is the channel which represents the characteristic/types of connection the application establishes with the driver/device and hence determines the data transfer capabilities the user gets to do to/from the device. For example, the channel could be input/output channel. This capability provided to the user/application, per channel, is determined by the capabilities of the underlying device. Per instance we have two channels, each for transmit and receive.

Note: The Raster Controller driver supports a display device – the raster LCD. Hence, there can be only output channels/Streams created for this driver.

S.No	Structure Elements (Raster_ChanObject)	Description
1	*dmaThrsld*	Used to calculate the ceiling address for DMA tansfer
2	*devInst*	The instance to which this channel belongs to.
3	*type*	The type of controller for which the channel is opened. (Only Raster type is supported)
4	*state*	The state of this channel
5	*chanConf*	The raster configuration passed as part of channel parameters described later below
6	*appCb and cbArg*	The call back function and the call back argument registered by the stream layer
7	*activeQ and pendingQ*	These queues are used to store the IO packets submitted by the application and later displayed by the driver. These two queues are necessary for asynchronous operation of the IDriver
8	*submitCount*	The number of packets currently pended in the queue. This is

		incremented every time a new packet is issued to the driver and decremented everytime the driver gives back the packet to the upper layer (Stream). Hence, this represents the number of packets yet to be processed by the driver
9	enabled	The current state of the channel. Since, the channel is a representation of the raster hardware, this Boolean is used to preserve the state of the raster. This is true if the raster is enabled. Note that once a packet is issued to the driver, the driver shall enable the raster to send it out on the hardware.
10	fbSize	This is the frame buffer size(in bytes) for the current configuration. For example if the raster is configured for 16 bits-per-pixel mode, for a resolution of 320 pixels by 240 lines, then the fbSize shall be 320x240x2 bytes = 153600 bytes), in addition to the palette size.
11	instHandle	Back pointer to the instance handle on which the channel is created
12	heapHandle	The raster driver supports the facility of allocating dynamic frame buffers on application's behalf as described in the coming sections. The driver has to invoke allocation methods of the Memory module, which optionally require a handle to the heap from which the memory has to be allocated. This is the case when user wants a separated heap for the frame buffer allocations alone. Then, the user is required to create a heap instance and pass a handle to this heap via channel parameters described below. If the user intends to use the default heap instance then this can be NULL.

2.1.2.3 The Device Parameters (also known as Instance parameters)

During module instantiation a set of parameters are required which shall be used to configure the hardware and the driver for that operation mode. This is passed via create call for the module instance. Please note that there are two ways to create an instance (static-using CFG file and dynamic using application c file) and each of these methods have different way of passing devparams. For further information of these methods please refer xdc documentation. These parameters are preserved in the params structure and are explained below:

S.No	Structure Elements Raster_Params	Description
1	instNum	This passes the instance number of the controller on which instantiation is being done.
2	devConf	The device configuration structure as described below

2.1.2.4 Device Configuration Structure (Raster_DeviceConf)

S.No	Structure Elements	Description
1	clkFreqHz	This is used to configure the output clock frequency of the raster controller
2	opMode	Operation mode of driver. Only DMA mode is supported. The raster controller is provided with an independent DMA engine and does not depend on other DMA peripheral/drivers
3	hwiNum	The hardware interrupt number assigned to the event group of the LCDC interrupt event.
4	dma	The DMA configuration structure as described below

2.1.2.5 DMA Configuration Structure (Raster_DmaConfig)

S.No	Structure Elements	Description
1	fbMode	This stores the DMA buffer mode. The raster DMA FIFO supports double frame buffer mode and single frame buffer mode. Please refer to the LCDC Peripheral User Guide for more details
2	burstSize	The FIFO level at which the DMA transfers are triggered
3	bigEndian	This is Boolean set to true if the processor is big endian architecture
4	eofInt	This Boolean is set to true if the End-of-frame interrupts for frame0 and frame1 are to be enabled. However, without enabling these interrupts the EOF0/1 status won't raise interrupts.

2.1.2.6 The Channel Parameters

Every channel opened to the device may need some setting by the user. These parameters may be passed through to the driver module by Raster_ChanParams. The channel parameters are passed as part of the Stream parameters structure during the stream create call. The channel parameters structure is described below.

S.No	Structure Elements	Description
1	Controller	This specifies the type of controller the channel is created for. Only raster type controller is supported
2	chanConf	This contains the channel configuration parameters for the controller type. This is a void pointer in order to be compatible with future changes if any. However, for raster controller type the channel configuration is

		supplied as a RasterConf structure as described later below.
3	heapHandle	The raster driver supports the facility of allocating dynamic frame buffers on application's behalf as described in the coming sections. The driver has to invoke allocation methods of the Memory module, which optionally require a handle to the heap from which the memory has to be allocated. This is the case when user wants a separated heap for the frame buffer allocations alone. Then, the user is required to create a heap instance and pass a handle to this heap via this variable. If the user intends to use the default heap instance then this can be NULL.

2.1.2.7 The Raster Configuration Parameters

S.No	Structure Elements	Description
1	outputFormat	The output format of the raster controller. This depends on the type of raster display connected to the controller
2	intface	This describes the connection interface like 4pin/8pin etc
3	panel	This describes the LCD panel type connected to the controller
4	display	This describes the display is color or monochrome display
5	bitsPP	This describes the bits per pixel of the image to be displayed
6	fbContent	This describes if the frame buffer data being supplied to the driver is palette only or data only or both data
7	dataOrder	This describes if the data is to be transferred with MSB first or the LSB first
8	nibbleMode	This describes if the data is to be transmitted in nibble mode or not

9	subPanel	This contains the subpanel settings are part of the subpanel structure
10	timing2	This describes the sync/clock/enable signal timing requirements as part of Timing 2 register requirements
11	fifoDmaDelay	The delay in terms of pixel clocks until which the input FIFO DMA requests should be disabled
12	intMask	The mask for which the interrupts should be enabled. The interrupt mask should correspond to the Raster_Intr enum
13	hFP, hBP, hSPW, vFP, vBP,vSPW	These describe the horizontal and vertical sync signal requirements of the display
14	pPL, lPP	These describe the pixels-per-line and the lines-per-panel of the display

2.2 Dynamic View

2.2.1 The Execution Threads

The Raster IDriver module involves following execution threads:

Application thread: Creation of channel, Control of channel, deletion of channel and processing of Raster data will be under application thread.

Interrupt context: Status and error handling are handled in the interrupt. Also, the updating of frame buffer address happens in the interrupt context.

Edma call back context: There is not EDMA callback context. The Raster module does not depend in the EDMA driver.

2.2.2 IO using Raster driver

In Raster, the application can perform IO operation using Stream_issue/reclaim() calls (corresponding IDriver function is Raster_Submit()). The handle to the channel, frame buffer for display data, size of data and timeout for transfer should be provided.

The Raster module receives this information via Raster_submit. Here some sanity checks on the driver shall be done like valid buffer pointers, mode of operation etc and actual write operation on the hardware shall be performed.

The channel/stream on the raster driver can only be in write mode since it is a display driver.

2.2.3 Functional Decomposition

The Raster driver, seen in the XDC framework, has two methods of instantiation, or instance creation – Static and Dynamic. By static instantiation we mean, the invocation of the create call for the module in the configuration file of the application. This is called so because, the creation of the instance is at build time of the application. By dynamic instantiation we mean, the invocation of the create call for the module during runtime of the application.

The two types of instantiation of the module are handled in different ways in the module. The static instantiation of the module is handled in the module script file, and the dynamic instantiation is handled in the C file.

This design concept explained in the sections to follow.

2.2.3.1 module$use() of XS file

One important feature in the XDC framework design of this package is that we have designed to have a soc.xs capsule file, instead of a soc.h file, which will have the SoC specific information(for more than one SOC), like interrupt/event numbers, base addresses, CPU/module frequency values etc. This move is adopted to keep the driver C code free from compiler switches and everything of this sort in the form of either configuration variables for the module or loading the platform specific data from the soc.xs capsule. This loading of data is done in the module use function.

The module shall have an array of device instance configuration structures (deviceInstInfo), which shall contain, base address, event number, frequency and such instance specific details. The length of this is defined by the array member of this instance in the soc.xs file.

The device information is populated for all the instance numbers in this function since this information is needed to later prepare the instances in instance static init and the instance init functions.

2.2.3.2 instance$static$init of XS file

This function context is the where the instance statically created is initialized. Please note that the instance parameters provided by the applications (from the CFG file) would override the default value of those parameters from XDC file.

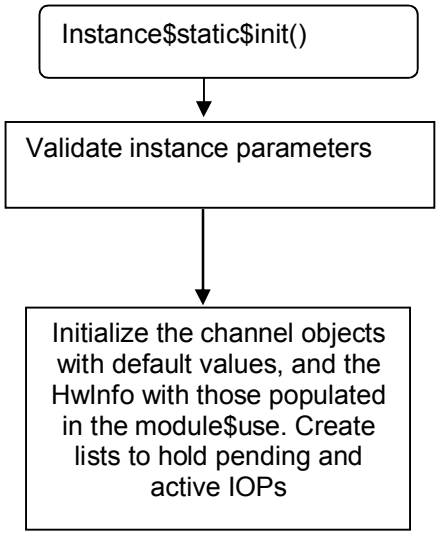

Figure 5 instance$static$init() flow diagram

2.2.3.3 Raster_Module_startup() of C file of driver Module

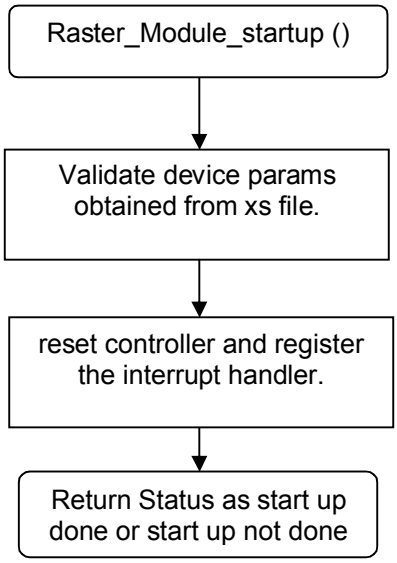

Figure 6 Raster_Module_startup() flow diagram

The module start up function is called as part of startup functions only once, during static instantiation, irrespective of the number of instances this module supports, during the module startup by the BIOS/XDC framework. This function is provided to make ready the module to start execution. In this function the data structures/resources needed for all instances are prepared. Here, initialization that cannot be performed in the module script file, as part of static (compile-time for ex., interrupt registration) initialization, should be performed.

This is the place where the C functions for initialization should be called

For the Raster module, for each instance:

1. Validate the device parameters for the instance
2. Reset the controller instance
    a. Reset the DMA engine
    b. Reset the device – clear all interrupt status
    c. Update DMA and interrupt settings as per device parameters
3. Register interrupt handler

In this function context the instance and the module variables shall not be available. In order to get these we shall use XDC APIs for getting these variables.

Hence the initialization for static instantiation consists of two parts, the script file (instance$static$init) and the C initialization (Raster_Module_startup).

2.2.3.4 Raster_Instance_init()

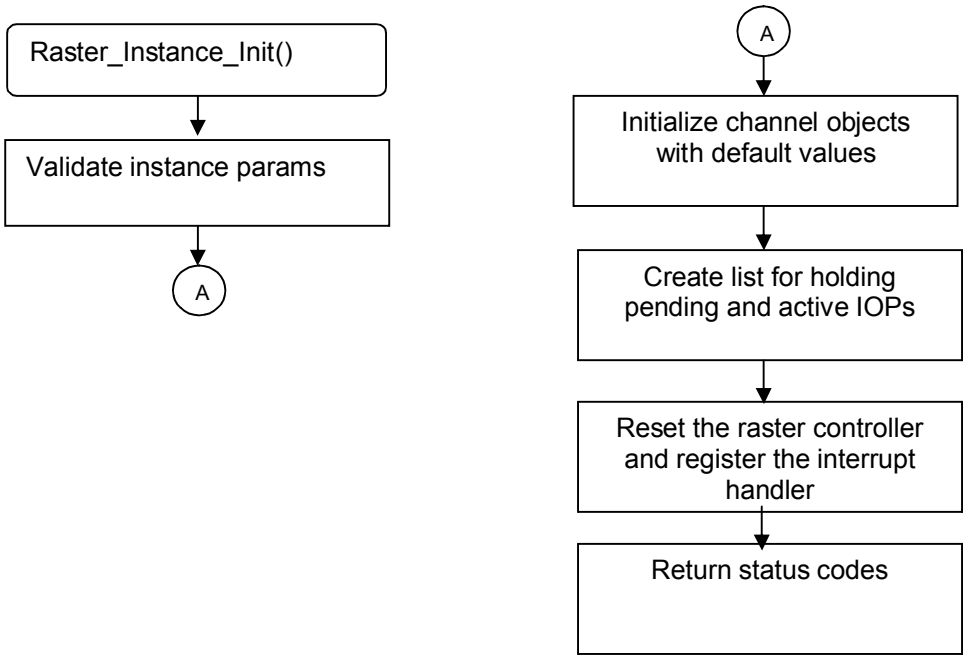

Figure 7 Raster_Instance_Init() flow diagram

The instance init function is called when the module is dynamically instantiated. This is the only context available for initialization per instance, when doing dynamic (application C file during run time) instantiation. Hence, this function should be including all the initialization done in the instance static init in the module script file and the module startup function. The return value of this function represents the extent to which the instance (and hence its resources) were initialized. For example, we could use return value of 0 for complete (successful) initialization done, return value of 1 for failure at the stage of a resource allocation and so on. This return value is preserved by the RTSC/XDC framework and passed to the Instance_finalize function, which does a clean up of the driver during instance removal accordingly.

2.2.3.5 Raster_Instance_finalize()

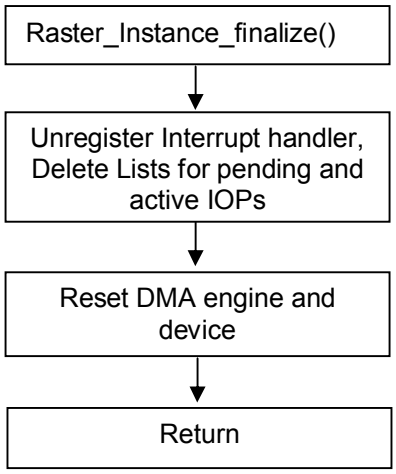

Figure 8 Raster_Instance_finalize () flow diagram

The Raster_Instance_init and Raster_Instance_finalize functions are called when by the XDC/BIOS framework.

The instance init function does a start up initialization for the driver. This function is called when the module is instantiated dynamically by the Raster_create call by the application. At this module instantiation, the interrupt registration and all other tasks (including any allocation of resource needed later by the driver) should be done here which form a pre-requisite before the actual functioning of the driver (viz channel creation and then data transfers).

The instance finalize function does a final clean up before the driver could be relinquished of any use. Here, all the resources which were allocated during instance initialization shall be unallocated, interrupt handlers shall be unregistered. After this the instance no more is valid and needs to be reinitialized. Please note that the input parameter for this function is the initialization status returned from the instance)init function. This helps in de-allocation of resources only that were actually allocated during instance_init.

2.2.3.6 Raster_open()

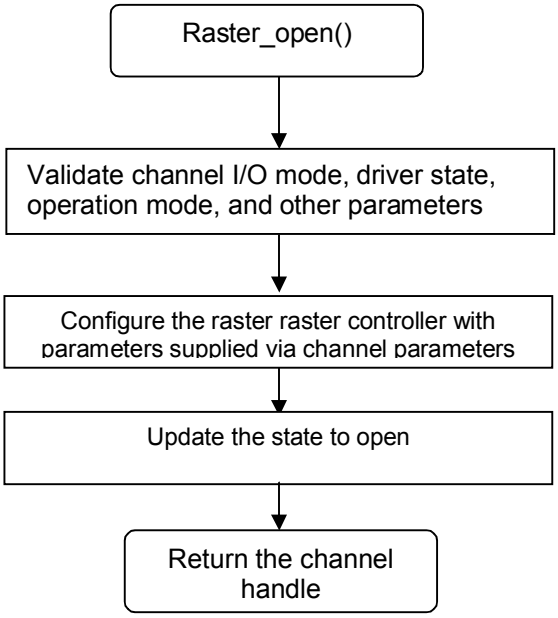

Figure 9 Raster_open () flow diagram

2.2.3.7 Raster_close()

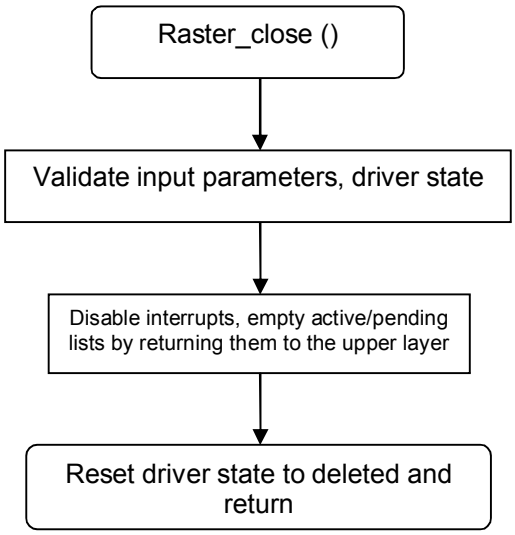

Figure 10 Raster_close() flow diagram

It is important that the interrupts are disabled, which will stop updating the frame buffer pointer updates in the interrupt handler. Also, after this, the packets currently the driver holds should be returned back to the upper layer.

2.2.3.8 Raster_control()

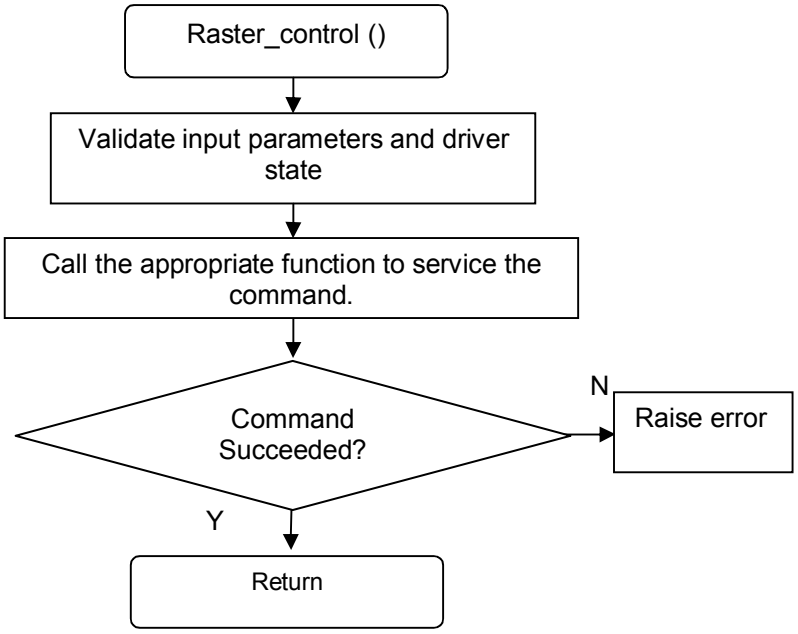

Figure 11 Raster_control () flow diagram

The raster driver, apart from implementing the routine device control IOCTLs, implements two distinct IOCTLs for allocation and freeing of the frame buffers.
In some cases it may be desired that the application may not be willing to use statically created frame buffers for display. It may do so at runtime. The memory allocations for such cases may be requested, to be done by the driver, from the application. This is facilitated by Raster_IOCTL_ALLOC_FB IOCTL call. However, the usage of this IOCTL requires some work to be done by the application prior to this call – creating of a heap handle. The dynamic memory allocations, Memory_alloc(…) etc, happen from a heap. This heap instance may be one exclusively created for this purpose by the application and handle to this is passed via channel parameters. This heap shall be used to allocate the buffers. This creation of heap instance is, however, optional. The application may choose to use the default heap instance of the system (Memory module). In such a case the heaphandle is NULL. During, such a scenario, the memory module uses the NULL handle which allocates the memory from the default heap instance. Please note that the size of the heap (default/exclusive) however has to be modified in the .cfg file of the application to suit the memory requirements of the application. Refer to Appendix A for complete list of IOCTLs

2.2.3.9 Raster_submit()

2.2.3.9.1 The issue/reclaim concept

The raster driver being a display driver is used in the context of streaming display data onto a display device. The stream data is a continuous stream of display buffers and in many cases real time in nature and hence the display calls (API to display data) cannot be blocking in nature. Also, the application and the driver should evolve some method for buffer exchange – the exchange of used/displayed buffer for a new one.

From the application perspective it supplies the buffer with new data and the raster driver should notify the application with completion by giving back the buffer to it in this exchange. The stream frame work provides a very good model of issue and reclaims to facilitate this mechanism. The application provides new data by issue (Stream_issue calls) and the waits to reclaim packet (completed) by calling Stream_reclaim call.

From the driver perspective, once the application issues a buffer, the raster is fit to be enabled because there is some data to be displayed. It queues up the incoming packet and then enables the raster. This buffer is returned to the application by calling the application callback with this packet. It is only after this callback assertion, which is done in the interrupt handler that the application's reclaim call returns.

So, it should be noted that, the reclaim blocks until there is some packet available for reclaim. This may be detrimental to the overall display performance.

In order that the application's reclaim call always returns, the application should prime the driver and then start the reclaim and issue. However, the number of packets primed should be traded-off with the latency requirements. If too few packets are primed the application might find a delay in the output rate of the driver and hence see glitch (a still image because of a frame being re-displayed repeatedly until new frame buffer is submitted to driver). By this way the driver and the application always have some packets to exchange always – except when the application does not have anything else to display. In this case if the application stops the issues, then the driver holds back on giving the last buffer to the application, since this is what is currently being displayed and if it returns there is no image to be displayed on the raster while the raster is enabled. Also if the driver gives back the buffer while it continues to display the image with this buffer address the DMA addresses programmed by this buffer addresses may be altered by the application without its knowledge (or a new buffer for display) and the display may get corrupted. The raster keeps displaying this last frame. If under such condition the reclaim is called again, it blocks. Hence, reclaim should not be called in the last packet. In essence the number of issues on a stream should be greater than reclaims. The last packet can be only returned when the raster is disabled. In this case all packets pended into the driver are returned back to the application.

2.2.3.9.2 I/O packet management

As seen above, packets pending to be processed (displayed) and packets being processed (displayed), should be simultaneously managed in the driver. This is done by maintaining two queues – one for packets pending (submitted to the driver but yet to be displayed) and one for packets being displayed. These queues are called pending and active queues respectively. The number of packets under driver control at anytime (submit count) is also maintained. This count is the difference of number of packets submitted to the driver by the application and the number of packets returned by the driver to the application.

- Different scenarios of inserting the packets ("enqueue") in the pending and the active queue are discussed below.
    - Initial submit

      This scenario occurs when channel has no packets queued to the driver and the raster is thus still disabled as there is nothing to display.

      In this case, when there is a Stream issue call and thus the Raster_submit() is invoked, the raster immediately shall en-queue the packet. The decision if it is pending queue or active queue is to be taken. If the active queue does not contain any packets, that is, the driver has not programmed the controller yet to display anything, then the packet can be put directly in to the active queue and the raster can be enabled and the channel could be flagged as enabled.
    - Intermediate submit

      This scenario occurs when the channel already has some packet(s) queued to the driver. However, the number of packets in active queue could be is less than it can actually hold, that is, in single frame buffer mode there can be only one packet in the active queue (only one packet being processed for frame0) and in double frame buffer mode there can be two packets in the active queue (two packets being processed for frame 0 and 1). If driver is working in double frame buffer mode and if the active queue contains only one frame then the second packet can also be queued into the active queue.

    In all other scenarios the packet is queued into the pending queue.

    The raster driver being an asynchronous driver, Raster_submit(…) always returns DriverTypes_PENDING in case of successful submit and DriverTypes_Error in case of submit errors while raising an error.
- Different scenarios of removing the packets ("dequeue") from the pending and the active queue are discussed below. (Dequeue is always in the ISR context).
    - One packet in active queue

When there is one packet in the active queue and the pending queue is empty, that is now the submit count is 1, packet from the active queue is not returned back to the application, since there is only one packet with the driver.

This is necessary to keep displaying something on the raster display incase if no further queue-ing from the application. The frame 0 and frame 1 addresses are programmed with the same buffer incase of double frame buffer mode. The application call to reclaim shall block indefinitely in this case if a finite timeout for reclaim is not is not specified.

If this last packet is returned (and the raster is still enabled) the application could possibly modify the contents of the buffer. The DMA engine since is still running with this buffer address, the display image is invalid and gets corrupted.

- Two packets in active queue

  When the active queue contains two packets the driver gives the just processed packet from the active queue back to the application. It then de-queues the second packet from the active queue programs the frame buffer DMA addresses and puts it back in the active queue. This “en-queue back to the active queue” of the last packet is required because we need to have at least one packet in the driver while the raster is enabled and to keep the display showing something. Here after, the one packet scenario comes up until further issues from the application.

- More than two packets

  When the issue of packets is fast enough there is always a chance that the pending queue always contains couple of packets (2 or more). In this case the driver gives back the one that is just processed from the active queue and gives it back to the application. It then dequeues from the pending queue and then puts it in the active queue and programs the DMA frame buffer addresses.

2.2.3.9.3 Synchronization of queue operations

The queue operations are done either in the task context in the Raster_submit() path(enqueue) or in the ISR context (dequeue). The only option here to prevent race condition on these queue operations is to disable the interrupt in the task context while doing queue related operations.

2.2.3.10 rasterIntrHandler()

The raster interrupt handler handles the status of raster operation. Only those status notifications that are enabled to raise interrupts result in the interrupt handler being called. It also handles the error conditions. In case of FIFO underflow or Sync errors the raster is disabled and re-enabled to recover from the errors.

In case of frame done interrupts (EOF1 and/or EOF2) interrupts the DMA frame buffer addresses are programmed accordingly as discussed in the section above.

3 APPENDIX A – IOCTL commands

Command	***Arguments***	***Description***
Raster_IOCTL_GET_DEVICE_CONF	*Pointer to DeviceConf structure*	*To get the current device configuration*
Raster_IOCTL_GET_RASTER_CONF	*Pointer to RasterConf structure*	*To get the current raster configuration*
Raster_IOCTL_GET_RASTER_SUBPANEL_CONF	*Pointer to SubPanel structure*	*To get the current raster sub panel configuration*
Raster_IOCTL_SET_RASTER_SUBPANEL_EN	*Pointer to boolean variable*	*If boolean is true then enables subpanel, else disables subpanel*
Raster_IOCTL_SET_RASTER_SUBPANEL_POS	*Pointer to SubpanelPos enum variable*	*To configure the position of the raster subpanel*
Raster_IOCTL_SET_RASTER_SUBPANEL_LPPT	*Pointer to interger variable*	*To configure the number of lines to be refreshed in the subPanel*
Raster_IOCTL_SET_RASTER_SUBPANEL_DATA	*Pointer to interger variable*	*To configure the default pixel data outside the subPanel*
Raster_IOCTL_GET_DMA_CONF	*Pointer to DmaConfig structure*	*To get the current DMA configuration setting*
Raster_IOCTL_SET_DMA_FB_MODE	*Pointer to DmaFb enum variable*	*To set the frame buffer mode for the*
Raster_IOCTL_SET_DMA_BURST_SIZE	*Pointer to the DmaBurstSize enum*	*To set the DMA burst size*
Raster_IOCTL_SET_DMA_EOF_INT	*Pointer to Boolean variable*	*To enable/disable the end-of-frame interrupt*
Raster_IOCTL_ADD_RASTER_EVENT	*Pointer to Integer variable containing the interrupt mask*	*To enable a specific event interrupt enable*
Raster_IOCTL_REM_RASTER_EVENT	*Pointer to integet variable containing interrupt mask*	*To disable a specific event interrupt disable*
Raster_IOCTL_GET_EVENT_STAT	*Pointer to EvenStat structure*	*To get the current event statistics*
Raster_IOCTL_CLEAR_EVENT_STAT	*None*	*Clears the current event statistics*
Raster_IOCTL_RASTER_ENABLE	*None*	*To enable the raster controller*
Raster_IOCTL_RASTER_DISABLE	None	To disable the raster

		controller
Raster_IOCTL_GET_DEVICE_VERSION	Pointer to Interger variable	To get the current version of the controller
Raster_IOCTL_ALLOC_FB	Pointer to a frame buffer pointer	To allocate a frame buffer on application's behalf
Raster_IOCTL_FREE_FB	Pointer to a frame buffer	To de-allocate a frame buffer in application's behalf