BIOS LCDC LIDD Device Driver

Architecture/Design Document

Revision History

Document Version	Author(s)	Date	Comments
0.1	Madhvapathi Sriram	February 9, 2009	Created the document
0.2	Madhvapathi Sriram	February 23, 2009	Addressed review comments

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 LIDD 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 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

1.2 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.

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

1.3 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

1.4 Related Documents

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

1.5 Hardware

The LCDC LIDD 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:

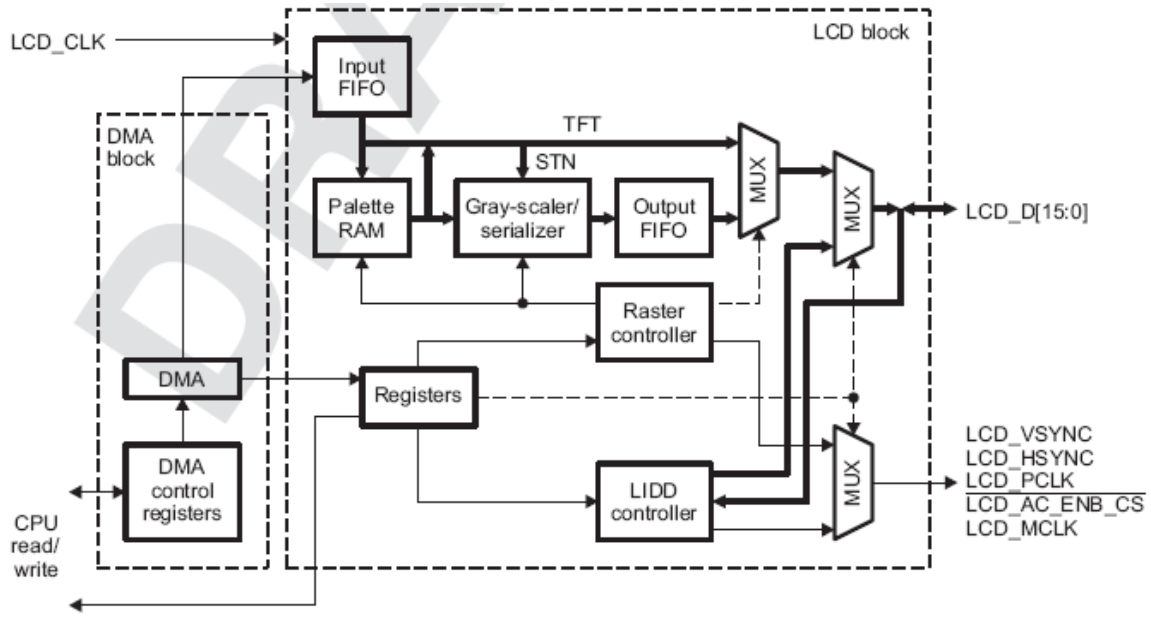

Figure 1 LCDC Block Diagram

1.6 Software

The LCDC LIDD 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.

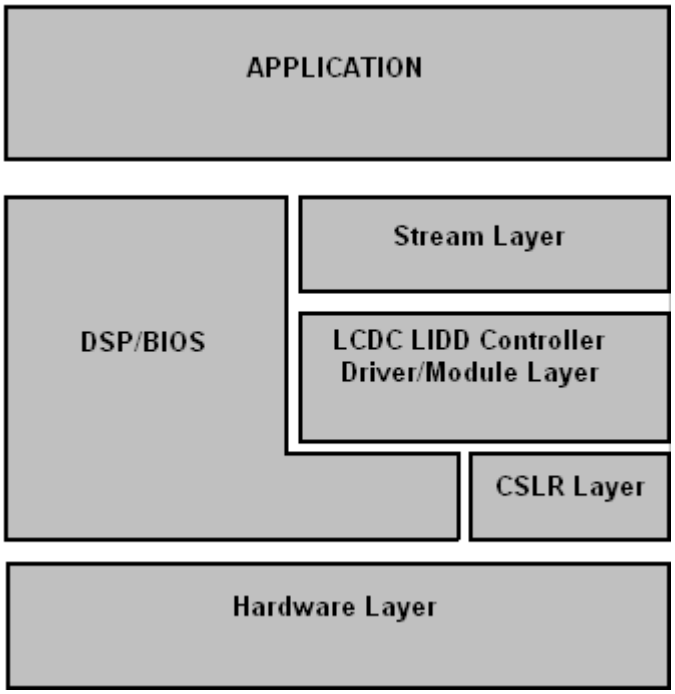

Figure 2 System Architecture

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 LIDD 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 synchronous driver (to the stream layer).

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 LIDD driver module is object of IDriver class. One may need to refer the IDriver documentation to access the LCDC LIDD 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 LIDD driver is intended to an XDC module and this module would inherit the IDriver interfaces. Thus the LCDC LIDD 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.

LIDD module specifications file (Lidd.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 etc

CONSTANTS: error ids, ioctls, minimum maximum values for 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 configuration 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 (Lidd.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.

LIDD module script file (Lidd.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 LIDD module/driver. The Lidd.xs file contains two parts

1. Handling the module use references

   When the module use is called in the application cfg for the LIDD module, the module use function in the Lidd.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 LIDD instance

   When the LIDD 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 LIDD IDriver module implements the following interfaces

S.No	IOM Interfaces	Description
1	Lidd_Module_startup()	Configure hardware and initialization needed before opening a channel. Effectively makes the LIDD module ready for use. This function is called at least once for sure when the LIDD IDriver module is used. Hence these tasks are done here for all the instances that can be created.
2	Lidd_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	Lidd_Instance_finalize()	Reset hardware and driver state and all de-initialization goes here. Effectively removes the usage of LIDD instance.
4	Lidd_open ()	Creates a communication channel in specified mode to communicate data between the application and the LIDD module instance.
5	Lidd_close ()	Frees a channel and all its associated resources.
6	Lidd_control ()	Implements the IOCTLS for LIDD IDriver module. All control operations go through this interface. Please refer to the Apeendix section for list of IOCTLs implemented
7	Lidd_submit ()	Submit an I/O packet to a channel for processing. Used for data transfer operations. Internally handles different mode of operation.

1.7.2 CSLR Interface

The CSL register interface (CSLR) provides register level implementations. CSLR is used by the LIDD IDriver module to configure LIDD 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 LIDD 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 (LIDD 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 (LIDD 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 LIDD channel handles, device configuration settings, hardware configuration etc. This is represented by the Instance_State in the LIDD 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 – LIDD 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 LIDD controller part of it.

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

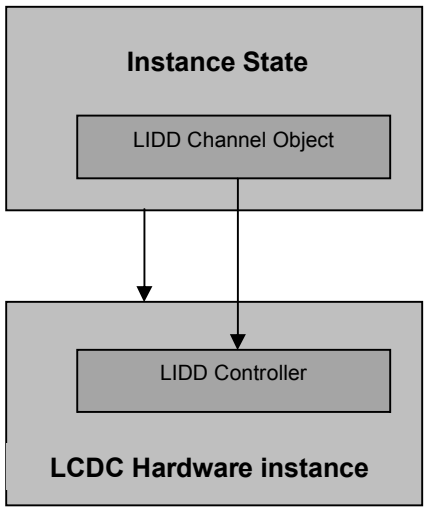

Figure 3 Instance Mapping

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

LIDD 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 LIDD 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 – LIDD module and the Raster module respectively.

- Only character type displays are supported for now. The display support is limited to 4 line displays

2 LIDD Driver Software Architecture

This section details the data structures used in the LIDD 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 LIDD IDriver module. The channel object is held inside the Instance State of the module. (This instance state is translated to the Lidd_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 LIDD driver.

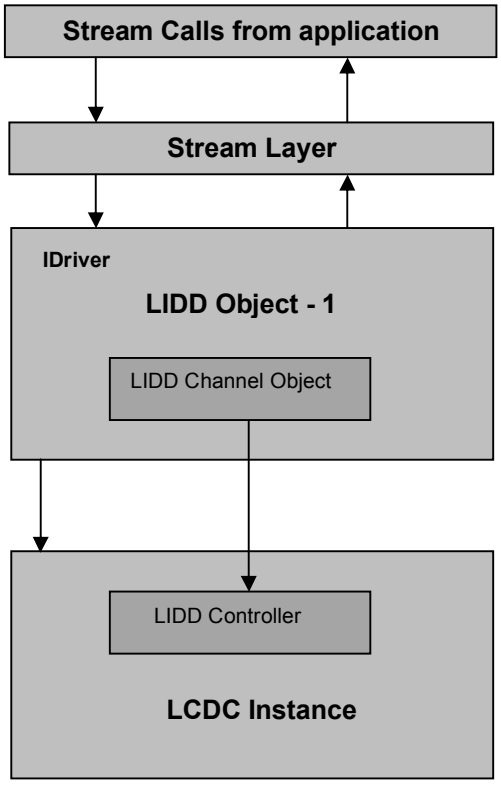

Figure 4 LIDD Driver Static View

2.1.2 Data Structures

The IDriver employs the Instance State (Lidd_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, LIDD configuration etc.

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

2.1.2.1 The Instance Object (Lidd_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 *(Lidd_Object)*	Description
1	*chCount*	Preserves the number of channels opened on this instance
1	*instNum*	Preserves instance number of LCDC LIDD hardware
3	*devType*	Preserves the device type. Currently only device type LCDC is supported
4	*state*	Preserves the current state of the instace/driver
5	*devConf*	Preserves the device configuration as passed by the application during instantiation
6	*ChannelObj*	The logical channel object that refers to the LIDD controller. This is described below.
7	*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 LIDD Controller driver supports a display device – the character LCD. Hence, there can be only output channels/Streams created for this driver.

S.No	Structure Elements (Lidd_ChanObject)	Description
1	*devInst*	The instance to which this channel belongs to.
2	*state*	The state of this channel
3	*chanConf*	The LIDD configuration passed as part of channel parameters described later below
4	*appCb and cbArg*	The call back function and the call back argument registered by the stream layer
5	*currPos*	The current position of the cursor
6	currSetting	The current display settings (cursor/display)
7	entryMode	The current mode settings for display/cursor
8	currShift	The current shift settings for cursor/display
9	onOffSetting	The current on/off settings for display/cursor

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 Lidd_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 (Lidd_DeviceConf)

S.No	Structure Elements	Description
1	displayType	Display type that is interfaced to the controller
2	clkFreqHz	This is used to configure the output clock frequency of the LCDC controller
3	hwiNum	The hardware interrupt number assigned to the event group of the LCDC interrupt event.
4	funcSet	The functionality setting of the display connected
5	addressArray	The array containing the address of first character of each line. This should be specified here, since the display type used is application specific

2.1.2.5 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 Lidd_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 LIDD 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 LIDD controller type the channel configuration is supplied as a DisplayConf structure as described later below.

2.1.2.6 The LIDD Display Configuration Parameters (Lidd_DisplayConf)

S.No	Structure Elements	Description
1	cs0Timing	This contains the strobe timings for read/write operations on the CS0 address space
2	cs1Timing	This contains the strobe timings for read/write operations on the CS1 address space

2.2 Dynamic View

2.2.1 The Execution Threads

The LIDD IDriver module involves following execution threads:

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

Interrupt context: The LIDD module does not contain any interrupt context.

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

2.2.2 IO using LIDD driver

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

The LIDD module receives this information via Lidd_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 LIDD driver can only be in write mode since it is a display driver.

2.2.3 Functional Decomposition

The LIDD 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.

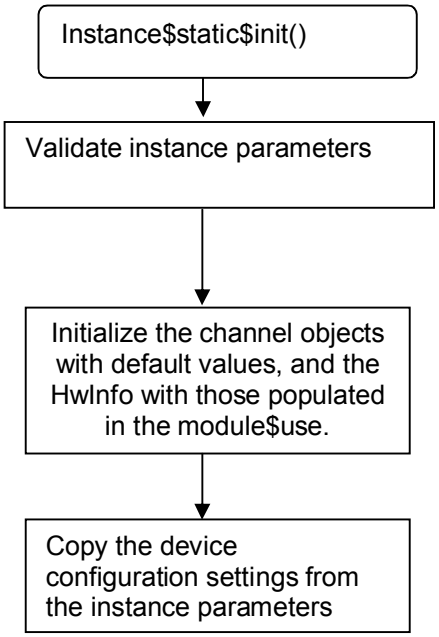

Figure 5 instance$static$init() flow diagram

2.2.3.3 Lidd_Module_startup() of C file of driver Module

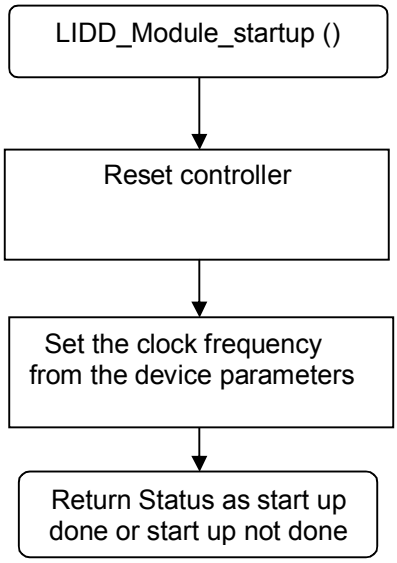

Figure 6 Lidd_Module_startup() flow diagram

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 LIDD module, for each instance:

1. Reset the controller instance
    a. Reset the device – clear all interrupt status
2. Configure clock settings as per device parameters

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 (Lidd_Module_startup).

2.2.3.4 Lidd_Instance_init()

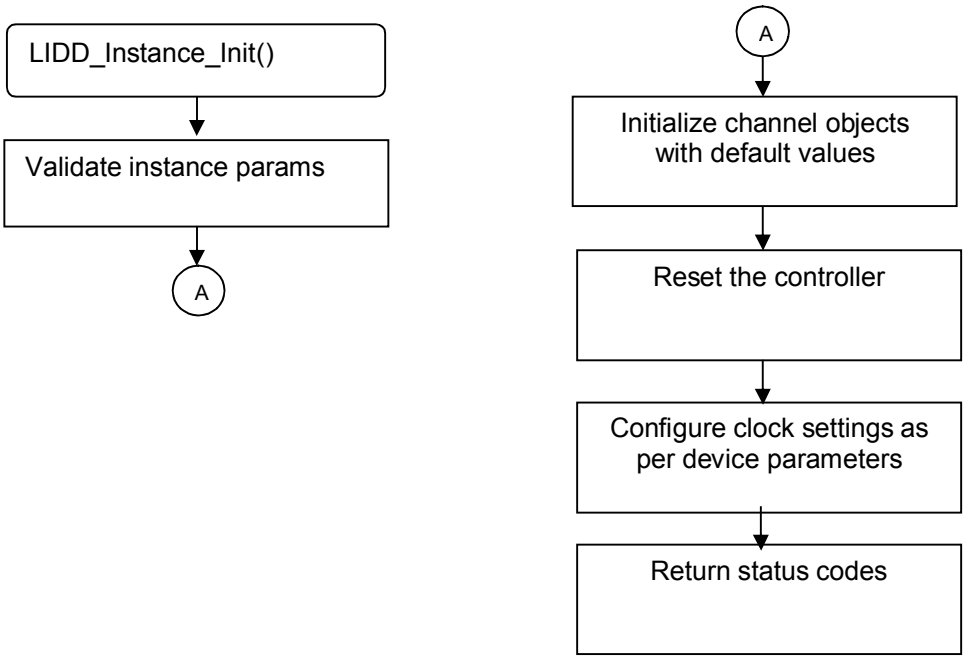

Figure 7 Lidd_Instance_Init() flow diagram

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 Lidd_Instance_finalize()

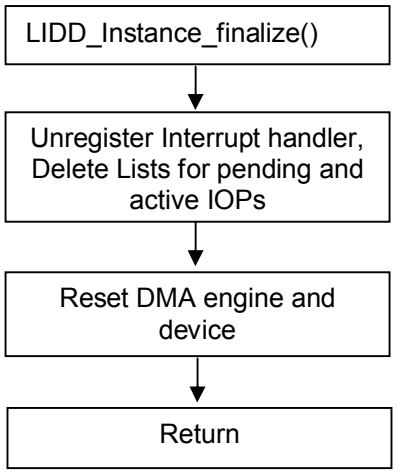

Figure 8 Lidd_Instance_finalize () flow diagram

The Lidd_Instance_init and Lidd_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 Lidd_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 Lidd_open()

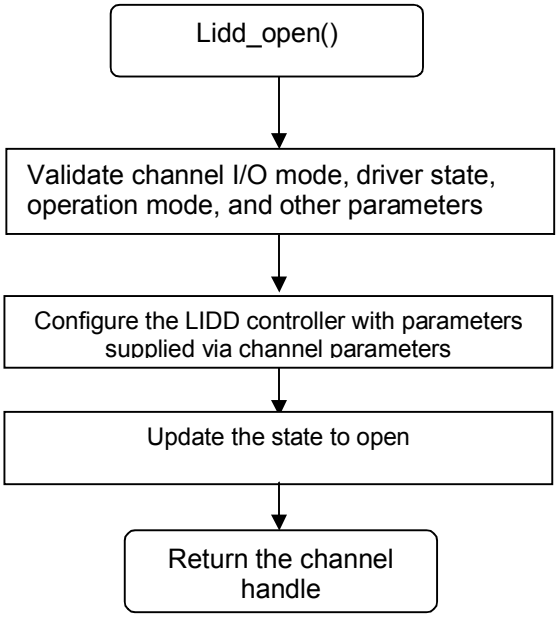

Figure 9 Lidd_open () flow diagram

2.2.3.7 Lidd_close()

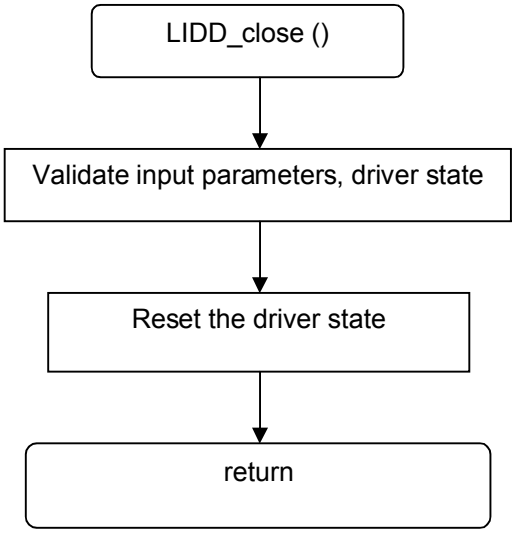

Figure 10 Lidd_close() flow diagram

2.2.3.8 Lidd_control()

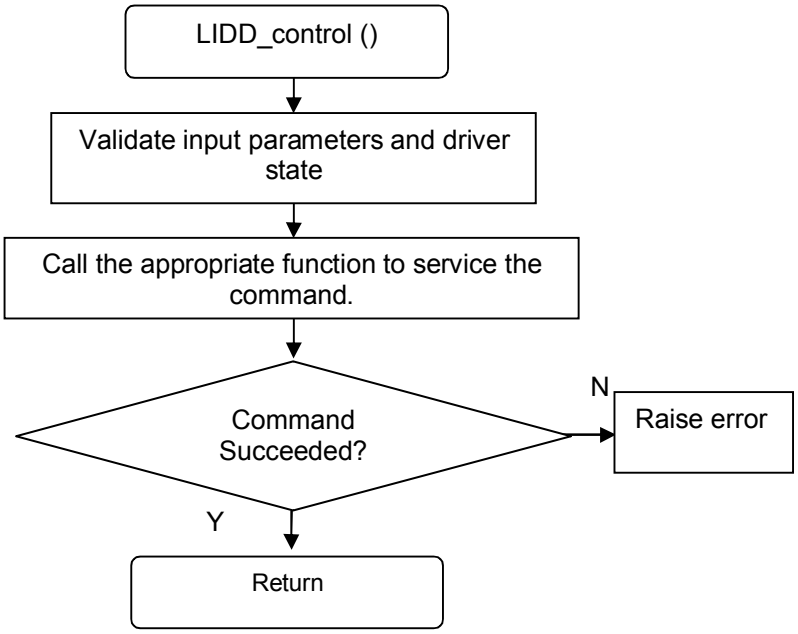

Figure 11 Lidd_control () flow diagram

The LIDD driver module implements IOCTLs for most of the operations required to be done on the character display module. Apart from this it also implements a generic IOCTL call for writing a command code to display module. This could be used by the application for specific operations that it needs and not implemented by the IOCTLs supplied.

2.2.3.9 Lidd_submit()

The LIDD driver being a display driver is used in the context of displaying data onto a character display devices. There is no other context in this module, than the application task context, which is submitting the display request. Hence, the driver is a synchronous driver.

As part of the submit call the driver needs a set of information like, buffer that contains the data to be displayed, length of the data to be displayed and the chip select address space onto which the operation needs to be performed. These are collected under one structure – Lidd_DataParam.

The submit call takes care of line wrap, that is, it automatically sets the cursor position to next line in case the current line display exceeds number of character s per line. The line wrap is done based on the parameters – number of lines, number of characters per line and address of the first character in each line. Once the position of the cursor reaches the number of characters per line, the next line s starting address is set in the address register of the panel. This occurs takes effect of line wrapping and the user need not worry about the number of charaters that are being sent on each line.

3 APPENDIX A – IOCTL commands

Command	Arguments	Description
IOCTL_CLEAR_SCREEN	Pointer to ioctlCmdArg type variable.	To clear the display screen, connected on chipSelect specified by the ioctlCmdArg
IOCTL_CURSOR_HOME	Pointer to ioctlCmdArg type variable.	To set the cursor to home position, for the display connected on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_POSITION	Pointer to CursorPosition structure	To set the cursor to a particular position in the display
IOCTL_SET_DISPLAY_ON	Pointer to ioctlCmdArg type variable.	To turn the display on for the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_OFF	Pointer to ioctlCmdArg type variable.	To turn the display off for, the chipsel specified by the ioctlCmdArg
IOCTL_SET_BLINK_ON	Pointer to ioctlCmdArg type variable.	To turn the cursor blink on for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_BLINK_OFF	Pointer to ioctlCmdArg type variable.	To turn the cursor blink off for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_ON	Pointer to ioctlCmdArg type variable.	To show the cursor for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_CURSOR_OFF	Pointer to ioctlCmdArg type variable.	To not show the cursor for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_SHIFT_ON	Pointer to ioctlCmdArg type variable.	To turn the display shift on for display, on the chipsel specified by the ioctlCmdArg
IOCTL_SET_DISPLAY_SHIFT_OFF	Pointer to ioctlCmdArg type variable.	To turn the display shift off for display, on the chipsel specified by the ioctlCmdArg
IOCTL_CURSOR_MOVE_LEFT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the cursor left display, on the chipsel specified by the ioctlCmdArg
IOCTL_CURSOR_MOVE_RIGHT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the cursor right display, on the chipsel specified by the ioctlCmdArg
IOCTL_DISPLAY_MOVE_LEFT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the display left, on the chipsel specified by the ioctlCmdArg

IOCTL_DISPLAY_MOVE_RIGHT	Pointer to ioctlCmdArg type variable.variable containing the interrupt mask	To move the display right, on the chipsel specified by the ioctlCmdArg
IOCTL_COMMAND_REG_WRITE	Pointer to Integer type variable	A generic IOCTL to write a command word to the Character display