ADI_OV7X48
DEVICE DRIVER

DATE: FEBRUARY 15, 2006

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
2006/02/08	Updated document
2006/05/17	Added device driver usage examples.

Table 1 - Revision History

1. Overview

The driver uses the TWI and PPI device drivers to interface to the Omnivision OV7X48 VGA video input sensor. The PPI and TWI configuration is fully configurable via the driver controls. Internal registers of the omnivision sensor can be accessed using device access commands and specific return codes are sent in result of success or failure. The PPI is only opened when the dataflow is turned on in the OV7X48 driver and is closed when the dataflow is turned off (note: this will cause the buffers sent to the PPI to be removed from the PPI device, if you only want to pause the PPI the you can send a command via the OV7X48 driver to sent the PPI dataflow)

2. Files

The files listed below comprise the device driver API and source files.

2.1. Include Files

The driver sources include the following include files:

- <services/services.h> This file contains all definitions, function prototypes etc. for all the System Services.
- <drivers/adi_dev.h> This file contains all definitions, function prototypes etc. for the Device Manager and general device driver information.
- <drivers/ppi/adi_ppi.h> This file contains all definitions, function prototypes etc. specific to PPI device
- <drivers/twi/adi_twi.h> This file contains all definitions, function prototypes etc. specific to TWI device
- <drivers/deviceaccess/adi_device_access.h> This file contains all definitions, function prototypes etc. for TWI/SPI device access service
- <drivers/sensor/omnivision/ adi_OV7X48.h> This file contains all definitions, function prototypes etc. specific to Omnivision OV7X48 sensor

2.2. Source Files

The driver sources are contained in the following files, as located in the default installation directory:

- adi_OV7X48.c

3. Lower Level Drivers

The OV7X48 driver uses the TWI and PPI. The TWI is used with the device access services. Both drivers can be given extra configuration options. If no control table is setup for the PPI it will be configured automatically by the driver by reading the configuration of the OV7X48 sensor via the TWI.

3.1. TWI

The TWI device driver is used by the OV7X48 driver to read and write to the configuration registers located on the OV7X48 hardware. The TWI device can be configured for use by control commands in the OV7X48 driver (refer section 5.4.3)

3.2. PPI

The PPI device driver is used by the OV7X48 to read in the image data from the sensor. This has to be configured correctly to read the correct image data. This can be done automatically by the driver by reading the configuration of the OV7X48 sensor, or manually by the user if a specific functionality that is not supported by the auto method of configuration.

4. Resources Required

Device drivers typically consume some amount of system resources. This section describes the resources required by the device driver.

Unless explicitly noted in the sections below, this device driver uses the System Services to access and control any required hardware. The information in this section may be helpful in determining the resources this driver requires, such as the number of interrupt handlers or number of DMA channels etc., from the System Services.

Because dynamic memory allocations are not used in the Device Drivers or System Services, all memory used by the Device Drivers and System Services must be supplied by the application. The Device Drivers and System Services supply macros that can be used by the application to size the amount of base memory and/or the amount of incremental memory required to support the needed functionality. Memory for the Device Manager and System Services is provided in the initialization functions (adi_xxx_Init()).

Wherever possible, this device driver uses the System Services to perform the necessary low-level hardware access and control.

The OV7X48 driver uses one PPI port and DMA control and one TWI port, this can be either a hardware TWI if the Blackfin device being used has a hardware port, or pseudo TWI if no TWI hadrware exists. In this case the TWI uses one timer and 2 general purpose flags.

4.1. Interrupts

The OV7X48 does not use any interrupts directly, please see PPI and TWI documentation for resources required by these drivers.

4.2. DMA

The OV7X48 does not use any DMA directly, however check the PPI documentation for DMA resources required by this driver.

4.3. Timers

The OV7X48 does not use any timers directly, however check the PPI and TWI documentation for timer resources required by these drivers.

4.4. Real-Time Clock

This driver does not require the real-time clock.

4.5. Programmable Flags

This driver does not use any programmable flags directly, please check TWI documentation for reqsources required by this driver.

4.6. Pins

This driver does not use any extenal pins.

5. Supported Features of the Device Driver

This section describes what features are supported by the device driver.

5.1. Directionality

The driver supports the dataflow directions listed in the table below.

ADI_DEV_DIRECTION	Description
ADI_DEV_DIRECTION_INBOUND	Supports the reception of data in through the device.

Table 2 - Supported Dataflow Directions

5.2. Dataflow Methods

The driver supports the dataflow methods listed in the table below.

ADI_DEV_MODE	Description
ADI_DEV_MODE_CIRCULAR	Supports the circular buffer method
ADI_DEV_MODE_CHAINED	Supports the chained buffer method
ADI_DEV_MODE_CHAINED_LOOPBACK	Supports the chained buffer with loopback method

Table 3 - Supported Dataflow Methods

5.3. Buffer Types

The driver supports the buffer types listed in the table below.

- ADI_DEV_CIRCULAR_BUFFER
  - Circular buffer
  - pAdditionalInfo – ignored
- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored
- ADI_DEV_2D_BUFFER
  - Two-dimensional buffer
  - pAdditionalInfo – ignored

5.4. Command IDs

This section enumerates the commands that are supported by the driver. The commands are divided into three sections. The first section describes commands that are supported directly by the Device Manager. The next section describes common commands that the driver supports. The remaining section describes driver specific commands.

Commands are sent to the device driver via the adi_dev_Control() function. The adi_dev_Control() function accepts three arguments:

- DeviceHandle – This parameter is a ADI_DEV_DEVICE_HANDLE type that uniquely identifies the device driver. This handle is provided to the client in the adi_dev_Open() function call.
- CommandID – This parameter is a u32 data type that specifies the command ID.
- Value – This parameter is a void * whose value is context sensitive to the specific command ID.

The sections below enumerate the command IDs that are supported by the driver and the meaning of the Value parameter for each command ID.

5.4.1. Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. As such, all device drivers support these commands.

- ADI_DEV_CMD_TABLE
  - Table of command pairs being passed to the driver
  - Value – ADI_DEV_CMD_VALUE_PAIR *
- ADI_DEV_CMD_END
  - Signifies the end of a command pair table
  - Value – ignored
- ADI_DEV_CMD_PAIR
  - Single command pair being passed
  - Value – ADI_DEV_CMD_PAIR *
- ADI_DEV_CMD_SET_SYNCHRONOUS
  - Enables/disables synchronous mode for the driver
  - Value – TRUE/FALSE

5.4.2. Common Commands

The command IDs described in this section are common to many device drivers. The list below enumerates all common command IDs that are supported by this device driver.

- ADI_DEV_CMD_GET_2D_SUPPORT
  - Determines if the driver can support 2D buffers
  - Value – u32 * (location where TRUE/FALSE is stored)
- ADI_DEV_CMD_SET_DATAFLOW_METHOD
  - Specifies the dataflow method the device is to use. The list of dataflow types supported by the device driver is specified in section 5.2.
  - Value – ADI_DEV_MODE enumeration
- ADI_DEV_CMD_SET_STREAMING
  - Enables/disables the streaming mode of the driver.
  - Value – TRUE/FALSE
- ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID
  - Returns the DMA channel ID value for the device driver's inbound DMA channel
  - Value – u32 * (location where the channel ID is stored)
- ADI_DEV_CMD_GET_OUTBOUND_DMA_CHANNEL_ID
  - Returns the DMA channel ID value for the device driver's outbound DMA channel
  - Value – u32 * (location where the channel ID is stored)
- ADI_DEV_CMD_SET_INBOUND_DMA_CHANNEL_ID
  - Sets the DMA channel ID value for the device driver's inbound DMA channel
  - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- ADI_DEV_CMD_SET_OUTBOUND_DMA_CHANNEL_ID
  - Sets the DMA channel ID value for the device driver's outbound DMA channel
  - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID
  - Returns the PMAP ID for the device driver's inbound DMA channel
  - Value – u32 * (location where the PMAP value is stored)
- ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID
  - Returns the PMAP ID for the device driver's outbound DMA channel
  - Value – u32 * (location where the PMAP value is stored)
- ADI_DEV_CMD_SET_DATAFLOW
  - Enables/disables dataflow through the device
  - Value – TRUE/FALSE
- ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT
  - Determines if the device driver is supported by peripheral DMA
  - Value – u32 * (location where TRUE or FALSE is stored)

- ADI_DEV_CMD_REGISTER_READ
  - Reads a single device register
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_READ
  - Reads a specific field location in a single device register
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_TABLE_READ
  - Reads a table of selective device registers
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ
  - Reads a table of selective device register fields
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_BLOCK_READ
  - Reads a block of consecutive device registers
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)
- ADI_DEV_CMD_REGISTER_WRITE
  - Writes to a single device register
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_WRITE
  - Writes to a specific field location in a single device register
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_TABLE_WRITE
  - Writes to a table of selective device registers
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE
  - Writes to a table of selective device register fields
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_BLOCK_WRITE
  - Writes to a block of consecutive device registers
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)

5.4.3. Device Driver Specific Commands

The command IDs listed below are supported and processed by the device driver. These command IDs are unique to this device driver.

- ADI_OV7X48_CMD_SET_TWI
  - Set the TWI device number to use
  - Value – u32 (device number)
- ADI_OV7X48_CMD_SET_TWIADDR
  - Set the device TWI address
  - Value – u32 (device address)
- ADI_OV7X48_CMD_SET_TWICONFIG
  - Set the extra configuration options for TWI device
  - Value - ADI_DEV_CMD_VALUE_PAIR * (extra TWI configuration controls)
- ADI_OV7X48_CMD_SET_PPI
  - Set the PPI device number to use
  - Value – u32 (device number)
- ADI_OV7X48_CMD_SET_PPICONFIG
  - Manually configure PPI with command table
  - Value - ADI_DEV_CMD_VALUE_PAIR * (manual PPI configuration controls)
- ADI_OV7X48_CMD_SET_PPICMD
  - Send commands directly to PPI (Note: Only works when OV7X48 dataflow is TRUE)
  - Value - ADI_DEV_CMD_VALUE_PAIR * (directly control PPI device)

Note: TWI needs to be configured for OV7X48 before calling the following commands.

- ADI_OV7X48_CMD_GET_HWID
  - Get the revision number of the OV7X48

  - Value – u32 * (Revision of chip being used, this driver was tested on Rev 0x7648)
- ADI_OV7X48_CMD_GET_MFRID
  - Get the revision number of the OV7X48
  - Value – u32 * (Address where manufacturers id is returned in)
- ADI_OV7X48_CMD_GET_HEIGHT
  - Get the height of the OV7X48 window
  - Value – u32 * (Height of window)
- ADI_OV7X48_CMD_GET_WIDTH
  - Get the width of the OV7X48 window
  - Value – u32 * (Width of window)
- ADI_OV7X48_CMD_SET_FRAMERATE
  - Set the framerate adjust value on OV7X48
  - Value – u32 (New 10bit framerate value)
- ADI_OV7X48_CMD_GET_FRAMERATE
  - Get the framerate adjust value from OV7X48
  - Value – u32 * (Address framerate read from OV7X48 is returned in)
- ADI_OV7X48_CMD_SET_HSYNCRDELAY
  - Set the horizontal sync rising edge delay on OV7X48
  - Value – u32 (New 10bit delay value, range 0-762 pixels)
- ADI_OV7X48_CMD_GET_HSYNCRDELAY
  - Get the horizontal sync rising edge delay value from OV7X48
  - Value – u32 * (Address delay read from OV7X48 is returned in)
- ADI_OV7X48_CMD_SET_HSYNCFDELAY
  - Set the horizontal sync rising edge delay on OV7X48
  - Value – u32 (New 10bit delay value, range 0-762 pixels)
- ADI_OV7X48_CMD_GET_HSYNCFDELAY
  - Get the horizontal sync rising edge delay value from OV7X48
  - Value – u32 * (Address delay read from OV7X48 is returned in)

5.5. Callback Events

This section enumerates the callback events the device driver is capable of generating. The events are divided into two sections. The first section describes events that are common to many device drivers. The next section describes driver specific event IDs. The client should prepare its callback function to process each event described in these two sections.

The callback function is of the type ADI_DCB_CALLBACK_FN. The callback function is passed three parameters. These parameters are:

- ClientHandle – This void * parameter is the value that is passed to the device driver as a parameter in the adi_dev_Open() function.
- EventID – This is a u32 data type that specifies the event ID.
- Value – This parameter is a void * whose value is context sensitive to the specific event ID.

The sections below enumerate the event IDs that the device driver can generate and the meaning of the Value parameter for each event ID.

5.5.1. Common Events

The events described in this section are common to many device drivers. The list below enumerates all common event IDs that are supported by this device driver.

- ADI_DEV_EVENT_BUFFER_PROCESSED
  - Notifies callback function that a chained or sequential I/O buffer has been processed by the device driver. This event is also used to notify that an entire circular buffer has been processed if the driver was directed to generate a callback upon completion of an entire circular buffer.

  - Value – For chained or sequential I/O dataflow methods, this value is the CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read(), adi_dev_Write() or adi_dev_SequentialIO() function. For the circular dataflow method, this value is the address of the buffer provided in the adi_dev_Read() or adi_dev_Write() function.
- ADI_DEV_EVENT_SUB_BUFFER_PROCESSED
  - Notifies callback function that a sub-buffer within a circular buffer has been processed by the device driver.
  - Value – The address of the buffer provided in the adi_dev_Read() or adi_dev_Write() function.
- ADI_DEV_EVENT_DMA_ERROR_INTERRUPT
  - Notifies the callback function that a DMA error occurred.
  - Value – Null.

5.5.2. Device Driver Specific Events

The events listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

This device driver does not have any specific events.

5.6. Return Codes

All API functions of the device driver return status indicating either successful completion of the function or an indication that an error has occurred. This section enumerates the return codes that the device driver is capable of returning to the client. A return value of ADI_DEV_RESULT_SUCCESS indicates success, while any other value indicates an error or some other informative result. The value ADI_DEV_RESULT_SUCCESS is always equal to the value zero. All other return codes are a non-zero value.

The return codes are divided into two sections. The first section describes return codes that are common to many device drivers. The next section describes driver specific return codes. The client should prepare to process each of the return codes described in these sections.

Typically, the application should check the return code for ADI_DEV_RESULT_SUCCESS, taking appropriate corrective action if ADI_DEV_RESULT_SUCCESS is not returned. For example:

```
if (adi_dev_Xxxx(…) == ADI_DEV_RESULT_SUCCESS) {
      // normal processing
} else {
      // error processing
}
```

5.6.1. Common Return Codes

The return codes described in this section are common to many device drivers. The list below enumerates all common return codes that are supported by this device driver.

- ADI_DEV_RESULT_SUCCESS
  - The function executed successfully.
- ADI_DEV_RESULT_NOT_SUPPORTED
  - The function is not supported by the driver.
- ADI_DEV_RESULT_DEVICE_IN_USE
  - The requested device is already in use.
- ADI_DEV_RESULT_NO_MEMORY
  - There is insufficient memory available.
- ADI_DEV_RESULT_BAD_DEVICE_NUMBER
  - The device number is invalid.
- ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED

  - The device cannot be opened in the direction specified.
- ADI_DEV_RESULT_BAD_DEVICE_HANDLE
  - The handle to the device driver is invalid.
- ADI_DEV_RESULT_BAD_MANAGER_HANDLE
  - The handle to the Device Manager is invalid.
- ADI_DEV_RESULT_BAD_PDD_HANDLE
  - The handle to the physical driver is invalid.
- ADI_DEV_RESULT_INVALID_SEQUENCE
  - The action requested is not within a valid sequence.
- ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE
  - The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE
  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- ADI_DEV_RESULT_DATAFLOW_UNDEFINED
  - The dataflow method has not yet been declared.
- ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE
  - The dataflow method is incompatible with the action requested.

- ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE
  - The device does not support the buffer type provided.
- ADI_DEV_RESULT_CANT_HOOK_INTERRUPT
  - The Interrupt Manager failed to hook an interrupt handler.
- ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT
  - The Interrupt Manager failed to unhook an interrupt handler.
- ADI_DEV_RESULT_NON_TERMINATED_LIST
  - The chain of buffers provided is not NULL terminated.
- ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED
  - No callback function was supplied when it was required.
- ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE
  - Requires the device be opened for either inbound or outbound traffic only.
- ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE
  - Requires the device be opened for bidirectional traffic only.

Return codes specific to TWI/SPI Device access service

- ADI_DEV_RESULT_TWI_LOCKED
  - Indicates the present TWI device is locked in other operation
- ADI_DEV_RESULT_REQUIRES_TWI_CONFIG_TABLE
  - Client need to supply a configuration table for the TWI driver
- ADI_DEV_RESULT_CMD_NOT_SUPPORTED
  - Command not supported by the Device Access Service
- ADI_DEV_RESULT_INVALID_REG_ADDRESS
  - The client attempting to access an invalid register address
- ADI_DEV_RESULT_INVALID_REG_FIELD
  - The client attempting to access an invalid register field location
- ADI_DEV_RESULT_INVALID_REG_FIELD_DATA
  - The client attempting to write an invalid data to selected register field location
- ADI_DEV_RESULT_ATTEMPT_TO_WRITE_READONLY_REG
  - The client attempting to write to a read-only location
- ADI_DEV_RESULT_ATTEMPT_TO_ACCESS_RESERVE_AREA
  - The client attempting to access a reserved location
- ADI_DEV_RESULT_ACCESS_TYPE_NOT_SUPPORTED
  - Device Access Service does not support the access type provided by the driver

5.6.2. Device Driver Specific Return Codes

The return codes listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

- ADI_OV7X48_RESULT_ALREADYSTOPPED
  - ADI_DEV_CMD_DATAFLOW=FALSE command was sent, but the device was not running
- ADI_OV7X48_RESULT_INVALID_DEVICE
  - The device driver detected that there was no OV7X48 sensor plugged in.
- ADI_OV7X48_RESULT_PPI_CLOSED
  - ADI_OV7X48_CMD_SET_PPICMD command was executed while the PPI had not been started. The PPI starts when the OV7X48 device driver DATAFLOW has been enabled.

6. Opening and Configuring the Device Driver

This section describes the default configuration settings for the device driver and any additional configuration settings required from the client application.

6.1. Entry Point

When opening the device driver with the adi_dev_Open() function call, the client passes a parameter to the function that identifies the specific device driver that is being opened. This parameter is called the entry point. The entry point for this driver is listed below.

- ADIOV7X48EntryPoint

6.2. Default Settings

The table below describes the default configuration settings for the device driver. If the default values are inappropriate for the given system, the application should use the command IDs listed in the table to configure the device driver appropriately. Any configuration settings not listed in the table below are undefined.

Item	Default Value	Possible Values	Command ID
TWI device	0	Valid TWI device number	ADI_0V7X48_CMD_SET_TWI
TWI address	0x21	8-123	ADI_OV7X48_CMD_SET_TWIADDR
TWI config	NULL	See TWIDeviceDriverDocumentation for TWI configuration details	ADI_OV7X48_CMD_SET_TWICONFIG
PPI device	0	Valid PPI device number	ADI_OV7X48_CMD_SET_PPI
PPI config	NULL	See PPIDeviceDriverDocumentation for PPI configuration details	ADI_OV7X48_CMD_SET_PPICONFIG

Table 4 - Default Settings

6.3. Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the device driver requires the application to specify the additional configuration information listed in the table below.

There are specifically no additonal commands that are required before starting the driver, however the user must make not of the TWI and PPI device to be used and if different to the default these must be set before starting the driver. Check TWIDeviceDriverDocumentation and PPIDeviceDriverDocumentation for defaults for these two drivers.

Item	Possible Values	Command ID
Dataflow method	See section 5.2	ADI_DEV_CMD_SET_DATAFLOW_METHOD

Table 5 – Additional Required Settings

7. Hardware Considerations

If the client intends to use pseudo TWI to access OV7X48 registers, specific port pins should be set in Blackfin to generate TWI SCL and SDA.

7.1. OV7X48 registers

Register	Address	Mode	Default	Description
ADI_OV7X48_GAIN	0x00	RW	0x00	Gain control [0x00-0x3F]
ADI_OV7X48_BLUE	0x01	RW	0x80	Blue channel gain [0x00-0xFF]
ADI_OV7X48_RED	0x02	RW	0x80	Red channel gain [0x00-0xFF]
ADI_OV7X48_SAT	0x03	RW	0x84	Image format – colour saturation [0x00-0xFF]
ADI_OV7X48_HUE	0x04	RW	0x34	Image format – colour hue control [0x00-0xFF]
ADI_OV7X48_CWF	0x05	RW	0x3E	AWB – Red/Blue pre-amplifier gain setting
ADI_OV7X48_BRT	0x06	RW	0x80	ABC – Brightness setting [0x00-0xFF]
ADI_OV7X48_PID	0x0A	R	0x76	Product ID number
ADI_OV7X48_VER	0x0B	R	0x48	Product version number
ADI_OV7X48_AECH	0x10	RW	0x41	Exposure value
ADI_OV7X48_CLKRC	0x11	RW	0x00	Internal clock settings
ADI_OV7X48_COMA	0x12	RW	0x14	Common control A
ADI_OV7X48_COMB	0x13	RW	0xA3	Common control B
ADI_OV7X48_COMC	0x14	RW	0x04	Common control C
ADI_OV7X48_COMD	0x15	RW	0x00	Common control D
ADI_OV7X48_HSTART	0x17	RW	0x1A	Output format – Horizontal frame (HREF column) start
ADI_OV7X48_HSTOP	0x18	RW	0xBA	Output format – Horizontal frame (HREF column) stop
ADI_OV7X48_VSTART	0x19	RW	0x03	Output format – Vertical frame (Row) start
ADI_OV7X48_VSTOP	0x1A	RW	0xF3	Output format – Vertical frame (Row) stop
ADI_OV7X48_PSHIFT	0x1B	RW	0x00	Data format – Pixel delay select (0x00-no delay, 0xFF=256 pixel delay
ADI_OV7X48_MIDH	0x1C	R	0x7F	Manufacturer ID byte – High
ADI_OV7X48_MIDL	0x1D	R	0xA2	Manufacturer ID byte – Low
ADI_OV7X48_FACT	0x1F	RW	0x01	Output format – Format control
ADI_OV7X48_COME	0x20	RW	0xC0	Common control E
ADI_OV7X48_AEW	0x24	RW	0x10	AGC/AEC – Stable operating region - upper limit
ADI_OV7X48_AEB	0x25	RW	0x8A	AGC/AEC – Stable operating region – lower limit
ADI_OV7X48_COMF	0x26	RW	0xA2	Common control F
ADI_OV7X48_COMG	0x27	RW	0xE2	Common control G
ADI_OV7X48_COMH	0x28	RW	0x20	Common control H
ADI_OV7X48_COMI	0x29	RW	0x00	Common control I
ADI_OV7X48_FRARH	0x2A	RW	0x00	Output format – Frame rate adjust high
ADI_OV7X48_FRARL	0x2B	RW	0x00	Data format – Frame rate adjust seting LSB
ADI_OV7X48_COMJ	0x2D	RW	0x81	Common control J
ADI_OV7X48_SPBC	0x60	RW	0x06	Signal process control B
ADI_OV7X48_RMCO	0x6C	RW	0x11	Colour matrix – RGB crosstalk compensation – R channel
ADI_OV7X48_GMCO	0x6D	RW	0x01	Colour matrix – RGB crosstalk compensation – G channel
ADI_OV7X48_BMCO	0x6E	RW	0x06	Colour matrix – RGB crosstalk compensation – B channel
ADI_OV7X48_COMK	0x70	RW	0x01	Common control K

Register	Address	Mode	Default	Description
ADI_OV7X48_COML	0x71	RW	0x00	Common control L
ADI_OV7X48_HSDYR	0x72	RW	0x10	Data format – HSYNC rising edge delay LSB
ADI_OV7X48_HSDYF	0x73	RW	0x50	Data format – HSYNC falling edge delay LSB
ADI_OV7X48_COMM	0x74	RW	0x20	Common control M
ADI_OV7X48_COMN	0x75	RW	0x02	Common control N
ADI_OV7X48_COMO	0x76	RW	0x00	Common control O
ADI_OV7X48_AVGY	0x7E	RW	0x00	AEC – Digital Y/G channel average
ADI_OV7X48_AVGR	0x7F	RW	0x00	AEC – Digital R/V channel average
ADI_OV7X48_AVGB	0x80	RW	0x00	AEC – Digital B/U channel average

Table 6 – Device registers

7.2. OV7X48 register fields

Field	Position	Size	Description
GAIN register			
ADI_OV7X48_GAINVALUE	0	6	Gain setting
SAT register			
ADI_OV7X48_SATVALUE	4	4	Saturation value
HUE Register			
ADI_OV7X48_HUEVALUE	0	5	Hue value
ADI_OV7X48_HUEENABLE	5	1	Hue enable
CWF Register			
ADI_OV7X48_BLUEGAIN	0	4	Blue channel pre-amplifier gain setting
ADI_OV7X48_REDGAIN	4	4	Red channel pre-amplifer gain setting
CLKRC Register			
ADI_OV7X48_CLKPRESCALAR	0	6	Internal clock pre-scalar [0x00-0x3F]
ADI_OV7X48_SYNCPOLARITY	6	2	Data format – HSYNC/VSYNC polarity 00 HSYNC Neg VSYNC Pos 01 HSYNC Neg VSYNC Neg 10 HSYNC Pos VSYNC Pos 11 HSYNC Neg VSYNC Pos
COMA Register			
ADI_OV7X48_AWBENABLE	2	1	AWB enable
ADI_OV7X48_OUTPUTCHANNEL	3	1	Output format – Output channel select A
ADI_OV7X48_YUVFORMAT	4	1	YUV Format (When register COMD[0]=0) 0: Y U Y V Y U Y V 1: U Y V Y U Y V Y (When register COMD[0]=0) 0: Y V Y U Y V Y U 1: V Y U Y V Y U Y
ADI_OV7X48_MIRRORIMAGE	6	1	Output format – Mirror image enable
ADI_OV7X48_SCCBRESET	7	1	SCCB – Register reset 0: No change 1: Reset all registers to default values
COMB Register			

Field	Position	Size	Description
ADI_OV7X48_AECENABLE	0	1	AEC Enable
ADI_OV7X48_AGCENABLE	1	1	AGC Enable
ADI_OV7X48_SCCBTRISTATE	2	1	SCCB – Tri-state enable Y[0:7]
ADI_OV7X48_ITUFORMAT	4	1	Data format – ITU-656 format enable
COMC Register			
ADI_OV7X48_HREFPOLARITY	3	1	Data format – HREF polarity 0: HREF positive 1: HREF negative
ADI_OV7X48_RESOLUTION	5	1	Output format – Resolution 0: VGA (640x480) 1: QVGA (320x240)
COMD Register			
ADI_OV7X48_BYTESWAP	0	1	Data format – UV sequence exchange
ADI_OV7X48_PCLKEDGE	6	1	Data format – Y[7:0] – PCLK reference edge 0: Y[7:0] data on PCLK falling edge 1: Y[7:0] data on PCLK rising edge
ADI_OV7X48_OFBD	7	1	Data format – Output flag bit disable 0: Frame = 254 data bits (00/FF=reserved flag bits) 1: Frame = 256 data bits
FACT Register			
ADI_OV7X48_RGBFORMAT	2	1	RGB:565/555 mode select 0: RGB:565 output format 1: RGB:555 output format
ADI_OV7X48_RGBFORMATENABLE	4	1	RGB: 565/555 enable control 0: Disable 1: Enable
COME Register			
ADI_OV7X48_COME_2XE	0	1	Y[7:0] 2x Iol/Ioh enable
ADI_OV7X48_EDGEENHANCE	4	1	Image quality – Edge enhancement enable
ADI_OV7X48_AECDIGAVERAGE	6	1	AEC – Digital averaging enable
COMF Register			
ADI_OV7X48_SWAPENABLE	2	1	Data format – Output data MSB/LSB swap enable
COMG Register			
ADI_OV7X48_OUTPUTRANGE	1	1	Data format – Output full range enable
ADI_OV7X48_RGBCCD	4	1	Colour matrix – RGB crosstalk compensation disable (used to increase colour filter's effiency)
COMH Register			
ADI_OV7X48_SCANSELECT	5	1	Output format – Scan select 0: Interlaced 1: Progressive
ADI_OV7X48_DEVICESELECT	6	1	Device select 0: OV7648 1: OV7148
ADI_OV7X48_RGBMODE	7	1	Output format – RGB output select 0: RGB 1: Raw RGB
COMI Register			
ADI_OV7X48_DEVICEVERSION	0	2	Device version
FRARH Register			
ADI_OV7X48_2PIXELDELAY	4	1	A/D – UV Channel 2 pixel delay enable
ADI_OV7X48_FRAMERATEMSB	5	2	Data format – Framerate adjust setting MSB FRA[9:0]=MSB+LSB=FRARH[6:5]+FRARL[7:0]

Field	Position	Size	Description
ADI_OV7X48_FRAMERATEEN	7	1	Data format – Frame rate adjust enable
COMJ Register			
ADI_OV7X48_BANDFILTER	2	1	AEC – Band filter enable
SPCB Register			
ADI_OV7X48_PREAMPMULT	7	1	AEC – 1.5x multiplier (pre-amplifier) enable
COMK Register			
ADI_OV7X48_COMK_2XE	6	1	Y[7:0] 2X Iol/Ioh enable
COML Register			
ADI_OV7X48_HSDYFMSB	0	2	Data format – HSYNC falling edge delay MSB
ADI_OV7X48_HSDYRMSB	2	2	Data format – HSYNC rising edge delay MSB
ADI_OV7X48_HSYNCREF	5	1	Data format – Output to HSYNC on HREF pin enable
ADI_OV7X48_PCLKREF	6	1	Data format – PCLK output gated by HREF enable
COMM Register			
ADI_OV7X48_MAXGAINSELECT	5	2	AGC – Maximum gainselect
COMN Register			
ADI_OV7X48_VERTICALFLIP	7	1	Output format – Vertical flip enable
COMO Register			
ADI_OV7X48_STANDBYMODE	5	1	Standby mode enabled

Table 7 – Device register fields

8. Appendix

8.1. Using OV7X48 Device Driver in Applications

This section explains how to use OV7X48 device driver in an application.

Device Manager Data memory allocation

This section explains device manager memory allocation requirements for applications using this driver. The application should allocate base memory + memory for one TWI device + memory for one PPI device + memory for OV7X48 device + memory for other devices used by the application

DMA Manager Data memory allocation

This section explains DMA manager memory allocation requirements for applications using this driver. The application should allocate base memory + memory for 1 DMA channel for PPI device + memory for DMA channels used by other devices in the application

Initialize Ez-Kit, Interrupt manager, Deferred Callback Manager, DMA Manager, Device Manager (all application dependent)

a. OV7X48 (driver) initialization

Step 1: Open OV7X48 Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set TWI device number

Step 3: Pass TWI Configuration table (refer section 8.2 for TWI configuration table examples)

Step 4: Set PPI device number to be used for OV7X48 video data flow
Example:
// Set OV7X48 to use PPI 0 for video dataflow
adi_dev_Control (OV7X48DriverHandle, ADI_OV7X48_CMD_SET_PPI, (void *) 0);

b. OV7X48 (hardware) initialization

Step 5: Set OV7X48 TWI device address
Example:
// set TWI device address as ADI_OV7X48_TWIADDR
adi_dev_Control(OV7X48DriverHandle, ADI_OV7X48_CMD_SET_TWI,
(void *) ADI_OV7X48_TWIADDR);

Step 6: Configure OV7X48 device to specific mode using device access commands
(refer section 8.3.2 for examples)

c. Video Dataflow configuration

Step 7: Set video dataflow method

Step 8: Load OV7X48 video buffers

Step 9: Enable OV7X48 video dataflow

d. Terminating OV7X48 driver

Step12: Terminate OV7X48 driver with adi_dev_Terminate( )

Terminate DMA Manager, Deferred Callback etc.., (application dependent)

8.2. TWI Configuration tables

This section contains TWI configuration table examples to access OV7X48 internal registers using BF533, BF537 and BF561 Ez-Kits

```
// Select TWI clock frequency & duty cycle (in this case its 100MHz & 50% Duty Cycle)
adi_twi_bit_rate          rate = { 100, 50 };
```

ADSP-BF533 EZ-KIT Lite & ADSP-BF561 EZ-KIT Lite

BF533 and BF561 do not have an inbuilt TWI peripheral. Analog Devices TWI device driver (adi_twi.c) can be configured in pseudo mode to mimic TWI operation with selected port pins and a timer. BF533 and BF561 Ez-Kits are designed to use PF0 and PF1 to generate TWI SCL and SDA signals respectively.

```
// BF533 TWI mimic pins and timer (PF0=SCL, PF1=SDA & General purpose Timer 0 used for pseudo TWI)
// BF561 TWI mimic pins and timer (PF0=SCL, PF1=SDA & General purpose Timer 2 used for pseudo TWI)\
#if defined (__ADSPBF533__)            // for BF533

adi_twi_pseudo_port     pseudo = { ADI_FLAG_PF0, ADI_FLAG_PF1, ADI_TMR_GP_TIMER_0,
                                   (ADI_INT_PERIPHERAL_ID) NULL };

#elif defined (__ADSPBF561__)          // for BF561

adi_twi_pseudo_port     pseudo = { ADI_FLAG_PF0, ADI_FLAG_PF1, ADI_TMR_GP_TIMER_2,
                                   (ADI_INT_PERIPHERAL_ID) NULL };

#endif

// Pseudo TWI configuration table
ADI_DEV_CMD_VALUE_PAIR  TWIConfig [ ] =  {
        { ADI_TWI_CMD_SET_PSEUDO,               (void *)(&pseudo)                   },
        { ADI_DEV_CMD_SET_DATAFLOW_METHOD,      (void *)ADI_DEV_MODE_SEQ_CHAINED    },
        { ADI_TWI_CMD_SET_FIFO,                 (void *)0x0000                      },
        { ADI_TWI_CMD_SET_RATE,                 (void *)(&rate)                     },
        { ADI_TWI_CMD_SET_LOSTARB,              (void *)1                           },
        { ADI_TWI_CMD_SET_ANAK,                 (void *)0                           },
        { ADI_TWI_CMD_SET_DNAK,                 (void *)0                           },
        { ADI_DEV_CMD_SET_DATAFLOW,             (void *)TRUE                        },
        { ADI_DEV_CMD_END,                      NULL                                }
        };
```

ADSP-BF537 EZ-KIT Lite

BF537 have an inbuilt TWI peripheral and the TWI device driver (adi_twi.c) can be configured to use hardware TWI

```
// Hardware TWI configuration table
ADI_DEV_CMD_VALUE_PAIR  TWIConfig [ ] = {
        { ADI_TWI_CMD_SET_HARDWARE,             (void *)ADI_INT_TWI                 },
        { ADI_DEV_CMD_SET_DATAFLOW_METHOD,      (void *)ADI_DEV_MODE_SEQ_CHAINED    },
        { ADI_TWI_CMD_SET_FIFO,                 (void *)0x0000                      },
        { ADI_TWI_CMD_SET_LOSTARB,              (void *)1                           },
        { ADI_TWI_CMD_SET_RATE,                 (void *)(&rate)                     },
        { ADI_TWI_CMD_SET_ANAK,                 (void *)0                           },
        { ADI_TWI_CMD_SET_DNAK,                 (void *)0                           },
        { ADI_DEV_CMD_SET_DATAFLOW,             (void *)TRUE                        },
        { ADI_DEV_CMD_END,                      NULL                                }
        };
```

8.3. Accessing OV7X48 registers

This section explains how to access the OV7X48 internal registers using driver specific commands and device access commands (refer *'deviceaccess'* documentation for more information).

Refer section 7.1 for list of OV7X48 device registers and section 7.2 for list of OV7X48 device registers fields

8.3.1. Read OV7X48 internal registers

1. Read a single register

```
// define the structure to access a single device register
ADI_DEV_ACCESS_REGISTER Read_Reg;

// Load the register address to be read
Read_Reg.Address = ADI_OV7X48_GAIN;

//clear the Data location
Read_Reg.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
// Register value will be read back to location - Read_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_READ, (void *) & Read_Reg);
```

2. Read a specific register field

```
// define the structure to access a specific device register field
ADI_DEV_ACCESS_REGISTER_FIELD Read_Field;

// Load the device register address to be accessed
Read_Field.Address = ADI_OV7X48_CHIPCONTROL;
// Load the device register field location to be read
Read_Field.Address = ADI_OV7X48_CHIPCONTROLSCAN;

//clear the Read_Field.Data location
Read_Field.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The register field value will be read back to location - Read_Field.Data
adi_dev_Control (DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_READ, (void *) & Read_Field);
```

3. Read table of registers

```
// define the structure to access table of device registers
ADI_DEV_ACCESS_REGISTER Read_Regs [ ] = {
                                    {ADI_OV7X48_SAT,     0},
                                    {ADI_OV7X48_BLUE,    0},
                                    ADI_OV7X48_RED,      0},
   /*MUST include delimiter */      {ADI_DEV_REGEND,     0}        // Register access delimiter
                                    };

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of registers listed above will be read to corresponding Data location in Read_Regs array
// i.e., value of ADI_OV7X48_SAT will be read to Read_Regs[0].Data,
// ADI_OV7X48_BLUE to Read_Regs[1].Data and value of ADI_OV7X48_RED to Read_Regs[2].Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *) &Read_Regs[0]);
```

4. Read table of register(s) fields

```
// define the structure to access table of device register(s) fields
ADI_DEV_ACCESS_REGISTER_FIELD Read_Fields [ ] = {
        {ADI_OV7X48_CLKRC,        ADI_OV7X48_SYNCPOLARITY,      0},
        {ADI_OV7X48_SAT,          ADI_OV7X48_SATVALUE,          0},
        {ADI_OV7X48_CLKRC,        ADI_OV7X48_CLKPRESCALAR,      0},
/*MUST include delimiter */ {ADI_DEV_REGEND,   0,      0}      };      // Register access delimiter

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of register fields listed above will be read to corresponding Data location in Read_Fields array
// i.e., value of ADI_OV7X48_SYNCPOLARITY will be read to Read_Fields[0].Data,
// ADI_OV7X48_SAT to Read_Fields [1].Data and ADI_OV7X48_CLKPRESCALAR to Read_Fields [2].Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *) & Read_Fields [0]);
```

5. Read block of registers

```
// define the structure to access a block of registers
ADI_DEV_ACCESS_REGISTER_BLOCK Read_Block;

// load the number of registers to be read
Read_Block.Count = 3;
// load the starting address of the register block to be read
Read_Block.Address = ADI_OV7X48_GAIN;
// define a 'Count' sized array to hold register data read from the device
u16 Block_Data[3] = { 0 };
// load the start address of the above array to Read_Block data pointer
Read_Block.pData = & Block_Data [0];

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of the registers in the given block will be read to corresponding Block_Data[ ] array
// value of ADI_OV7X48_GAIN will be read to Block_Data [0],
// ADI_OV7X48_BLUEto Block_Data[1] and ADI_OV7X48_RED to Block_Data[2]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) & Read_Block);
```

8.3.2. Configure OV7X48 internal registers

1. Configure a single OV7X48 register

```
// define the structure to access a single device register
ADI_DEV_ACCESS_REGISTER Cfg_Reg;

// Load the register address to be configured
Cfg_Reg.Address = ADI_OV7X48_PSHIFT;

//Load the configuration value to Cfg_Reg.Data location
Cfg_Reg.Data = 0x02;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The device register will be configured with the value in Cfg_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_WRITE, (void *) & Cfg_Reg);
```

2. Configure a specific register field

```
// define the structure to access a specific device register field
ADI_DEV_ACCESS_REGISTER_FIELD Cfg_Field;

// Load the device register address to be accessed
Cfg_Field.Address = ADI_OV7X48_CLKRC;
// Load the device register field location to be configured
Cfg_Field.Address = ADI_OV7X48_CLKPRESCALAR;

// load the new field value
Cfg_Field.Data = 1;
// Application calls adi_dev_Control( ) function with corresponding command and value
// Selected register field will be configured with the value in Cfg_Field.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_WRITE, (void *) & Cfg_Field);
```

3. Configure table of registers

```
// define the structure to access table of device registers (register address, register configuration value)
// Configuration table to set OV7648 sensor to output NTSC active frame only image.
ADI_DEV_ACCESS_REGISTER Cfg_Regs_NTSC[ ] = {
                                        {ADI_OV7X48_CLKRC,      0x00},
                                        {ADI_OV7X48_COMA,       0x14},
                                        {ADI_OV7X48_COMB,       0xA3},
                                        {ADI_OV7X48_COMC,       0x04},
                                        {ADI_OV7X48_COMD,       0x00},
                                        {ADI_OV7X48_COMH,       0x00},
                                        {ADI_OV7X48_FACT,       0x01},
                                        {ADI_OV7X48_PSHIFT,     0x02},
    /*MUST include delimiter */         {ADI_DEV_REGEND,        0}      }; // Register access delimiter
    // Configuration table to set OV7648 sensor to output BGR video to stream to LCD panel.
    ADI_DEV_ACCESS_REGISTER Cfg_Regs_BGR[ ] = {
                                        {ADI_OV7X48_CLKRC,      0x00},
                                        {ADI_OV7X48_COMA,       0x1C},
                                        {ADI_OV7X48_COMB,       0xA3},
                                        {ADI_OV7X48_COMC,       0x04},
                                        {ADI_OV7X48_COMD,       0x00},
                                        {ADI_OV7X48_COMH,       0x20},
                                        {ADI_OV7X48_FACT,       0x11},
                                        {ADI_OV7X48_PSHIFT,     0x03},
    /*MUST include delimiter */         {ADI_DEV_REGEND,        0}      }; // Register access delimiter
```

```
// Application calls adi_dev_Control( ) function with corresponding command and value
// Registers listed in the table will be configured with corresponding table Data values (in this case,BGR)
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Regs_BGR [0]);
```

4. Configure a table of register(s) fields

```
// define the structure to access table of device register(s) fields
// register address, register field to configure, field configuration value
ADI_DEV_ACCESS_REGISTER_FIELD Cfg_Fields [ ] = {
        {ADI_OV7X48_CLKRC,      ADI_OV7X48_CLKPRESCALAR,     0},
        {ADI_OV7X48_COMB,       ADI_OV7X48_AECENABLE,        1},
        {ADI_OV7X48_CLKRC,      ADI_OV7X48_SYNCPOLARITY,     0},
/*MUST include delimiter */ {ADI_DEV_REGEND,   0,     0}     };     // Register access delimiter

// Application calls adi_dev_Control( ) function with corresponding command and value
// Register fields listed in the above table will be configured with corresponding Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Fields [0]);
```

5. Configure a block of registers

```
// define the structure to access a block of registers
ADI_DEV_ACCESS_REGISTER_BLOCK Cfg_Block;

// load the number of registers to be configured
Cfg_Block.Count = 5;
// load the starting address of the register block to be configured
Cfg_Block.Address = ADI_OV7X48_CLKRC;

// define a 'Count' sized array to hold register data read from the device
// Register Configuration values
u16 Block_Data [5] = { 0x00, 0x1C, 0xA3, 0x04, 0x00 };

// load the start address of the above array to Cfg_Block data pointer
Cfg_Block.pData = & Block_Data [0];

// Application calls adi_dev_Control( ) function with corresponding command and value
// Registers in the given block will be configured with corresponding values in Block_Data[ ] array
adi_dev_Control (DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) &Cfg_Block);
```