ANALOG DEVICES

ADI_ADV717X DEVICE DRIVER

DATE: NOVEMBER 25, 2005

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
2005/11/25	Initial release (supports only ADV7179)
2006/01/18	Supports ADV717x family Compatibility to the updated PPI driver
2006/05/15	Updated to new device access interface Added register access examples

Table 1 – Revision History

1. Overview

The driver allows the client to control ADV7170 / ADV7171 / ADV7174 / ADV7179 video encoders. The encoder's sub-address registers are accessed via TWI port and the video dataflow is through PPI port. The application program can configure the internal registers of ADV717x using device access commands and specific return codes are sent in result of success or failure.

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/deviceaccess/adi_device_access.h> This file contains all definitions, function prototypes etc. for TWI/SPI device access service
- <drivers/encoder/adi_adv717x.h> This file contains all definitions, function prototypes etc. specific to ADV717x device

2.2. Source Files

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

- adi_adv717x.c
- adi_adv7170.c
- adi_adv7171.c
- adi_adv7174.c
- adi_adv7179.c

Application must not include the adi_adv717x.c file in directly to the project but rather one, or more, of the files with the complete device number such as adi_adv7171.c.

3. Lower Level Drivers

ADV717x driver is layered on TWI and PPI drivers

3.1. TWI Device Driver

ADV717x can be operated in various modes by configuring its internal registers and it is done via TWI port.

3.2. PPI Device Driver

PPI is used for video data transfer to the encoder. ADV717x device driver sets PPI device 0 to be used for its video dataflow.

Application can directly communicate with the PPI device allocated for ADV717x video dataflow by calling adi_dev_Control( ) function with PDDHandle specific to ADV717x driver, command specific to the PPI driver and value specific to the command.

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 ADV717x device driver is build upon interrupt driven TWI driver and DMA operated PPI driver.

4.1. Interrupts

No specific interrupts or interrupt handlers are used by this driver.

4.2. DMA

The driver doesn't support DMA directly, but uses a DMA driven PPI for its video dataflow. ADV717x supports only outbound dataflow and memory should be allocated for one DMA channel.

4.3. Timers

The driver uses following timers to generate PPI Frame Sync signals (FS1 & FS2) when operated in non-standard video mode.

Processor	PPI Device Number	Timers
BF533	0	Timer 0 for FS1, Timer 1 for FS2
BF537	0	Timer 0 for FS1, Timer 1 for FS2
BF561	0	Timer 8 for FS1, Timer 9 for FS2
BF561	1	Timer 10 for FS1, Timer 11 for FS2

Table 2 – PPI Frame sync timers

If the client intends to use a pseudo TWI to control ADV717x, a TWI configuration table must be passed with timer settings that will be used for pseudo TWI operation.

4.4. Real-Time Clock

RTC service is not used by this driver

4.5. Programmable Flags

No programmable flags are directly used by this driver. If the client intends to use a pseudo TWI to control ADV717x, a TWI configuration table must be passed with flag settings that will be used for pseudo TWI operation.

4.6. Pins

Connect corresponding PPI device port pins of Blackfin processor to video data input port pins (P7 to P0) of ADV717x.
Connect corresponding TWI device port pins of Blackfin processor to TWI port pins of ADV717x.

Please refer to corresponding device reference manuals for further information.

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_OUTBOUND	Supports the transmission of data out through the device.

Table 3 – 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 4 – 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 – optional
- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored
- ADI_DEV_2D_BUFFER
  - Two-dimensional buffer
  - pAdditionalInfo – optional

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_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_DATAFLOW
  - Enables/disables dataflow through the device
  - Value – TRUE/FALSE
- ADI_DEV_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. The driver also supports commands specific to PPI driver. Please refer to PPI driver documentation for further information.

Commands to access ADV717x SCF registers

- ADI_ADV717x_CMD_SET_SCF_REG
  - Sets Sub-carrier Frequency value. SCF registers. SCF registers should be configured by this command as they are not recommended to access separately.
  - Value – u32
- ADI_ADV717x_CMD_GET_SCF_REG
  - Gets the present Sub-carrier Frequency value. SCF registers. SCF registers should be read by this command as they are not recommended to access separately.
  - Value – u32 *

Commands specific to ADV717x non-standard video mode operation

- ADI_ADV717x_CMD_SET_TIMER_FRAME_SYNC_1
  - Configures the corresponding timer to generate Frame Sync 1 signal
  - Value – ADI_PPI_FS_TMR * (register specifics)
- ADI_ADV717x_CMD_SET_TIMER_FRAME_SYNC_2
  - Configures the corresponding timer to generate Frame Sync 2 signal
  - Value – ADI_PPI_FS_TMR * (register specifics)
- ADI_ADV717x_CMD_SET_FRAME_SYNC_COUNT
  - Sets number of frame syncs to be generated in non-standard video mode
  - Value – ADI_ADV717x_FS_COUNT (register specifics)
- ADI_ADV717x_CMD_SET_FRAME_LINES_COUNT
  - Sets frame line count for non-standard video mode
  - Value – u32

PPI related commands

- ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER
  - Sets PPI device number to be used for ADV717x video dataflow
  - Value – u32
- ADI_ADV717x_CMD_SET_PPI_STATUS
  - Sets PPI device status to be used for ADV717x video dataflow (Opens/Closes PPI)
  - Value – ADI_ADV717x_SET_PPI_STATUS (register specifics)

TWI related commands

- ADI_ADV717x_CMD_SET_TWI_DEVICE_NUMBER
  - Sets TWI device number to be used to access ADV717x internal registers.
  - Value – u32

- ADI_ADV717x_CMD_SET_TWI_CONFIG_TABLE
    - Clients TWI Configuration table
    - Value – ADI_DEV_CMD_VALUE_PAIR *
- ADI_ADV717x_CMD_SET_TWI_DEVICE_ADDRESS
    - Sets TWI address specific to ADV717x
    - Value – u32

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 driver doesn't have any unique 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_ADV717x_RESULT_CMD_NOT_SUPPORTED
    - Command supplied by the client is not supported by ADV717x device driver
- ADI_ADV717x_RESULT_TIMING_NOT_CONFIGURED
    - Results when client attempts to enable video dataflow in non-standard video modes without configuring Frame Sync generation timers
- ADI_ADV717x_RESULT_FRAME_ERROR
    - Results when client provides a wrong Frame Sync or Frame Line count (for non-standard video)
- ADI_ADV717x_RESULT_DISABLE_DATAFLOW
    - Results when client attempts to change ADV717x operating mode with video dataflow still on
- ADI_ADV717x _RESULT_BAD_PPI_DEVICE
    - Results when the client provides a wrong PPI device number
- ADI_ADV717x_RESULT_OPERATING_MODE_MISMATCH
    - Results when client tries to set Frame sync / Frame Line count with ADV717x in wrong mode
- ADI_ADV717x_RESULT_PPI_STATUS_INVALID
    - Results when client tries operate PPI in an invalid state

6. 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.

- ADIADV7170EntryPoint
- ADIADV7171EntryPoint
- ADIADV7174EntryPoint
- ADIADV7179EntryPoint

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.

Item	Default Value	Possible Values	Command ID
Video Format	NTSC	NTSC, PAL	Configuring ADV717x registers (refer page 26 for examples)
Video Mode	Standard (ITU-R 656)	ITU-R 656, user specific	Configuring TMR0 of ADV717x (refer page 26 for examples)
TWI Address	0x6A for AD7V170 & ADV7174 0x2A for ADV7171 & ADV7179	Device Specific	ADI_ADV717x_CMD_SET_TWI_DEVICE_ADDRESS
TWI Device Number	0	N	ADI_ADV717x_CMD_SET_TWI_DEVICE_NUMBER

Table 5 – 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.

Item	Possible Values	Command ID
TWI Configuration Table	Pointer to TWI configuration table of type ADI_DEV_CMD_VALUE_PAIR	ADI_ADV717x_CMD_SET_TWI_CONFIG_TABLE
PPI Device	0 (for BF533, BF537) 0, 1 (for BF561)	ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER
PPI Status	ADI_ADV717x_PPI_OPEN, ADI_ADV717x_PPI_CLOSE	ADI_ADV717x_CMD_SET_PPI_STATUS

Table 6 – Additional Required Settings

7. Hardware Considerations

ADV17x video encoders use a pin to set bit 1 of its TWI slave address. The TWI slave address of ADV717x can be set by issuing the command 'ADI_ADV717x_CMD_SET_TWI_DEVICE_ADDRESS'. If the client intends to use pseudo TWI to access ADV717x registers, specific port pins should be set in Blackfin to generate TWI SCL and SDA.

7.1. ADV717X registers

The following tables contains list of accessible registers & register fields in ADV717x and corresponding macro names defined in the driver header file. Please refer to the ADV717x device manual for a full description of registers and chip functionality.

Register	Address	Default	Description
ADV717x_MR0	0x00	0x00	Mode Register 0
ADV717x_MR1	0x01	0x58	Mode Register 1
ADV717x_MR2	0x02	0x00	Mode Register 2
ADV717x_MR3	0x03	0x00	Mode Register 3
ADV717x_MR4	0x04	0x10	Mode Register 4
ADV717x_TMR0	0x07	0x00	Timing Mode Register 0
ADV717x_TMR1	0x08	0x00	Timing Mode Register 1
ADV717x_SCFR0	0x09	0x16	Sub carrier Frequency Register 0
ADV717x_SCFR1	0x0A	0x7C	Sub carrier Frequency Register 1
ADV717x_SCFR2	0x0B	0xF0	Sub carrier Frequency Register 2
ADV717x_SCFR3	0x0C	0x21	Sub carrier Frequency Register 3
ADV717x_SCPR	0x0D	0x00	Sub Carrier Phase Register
ADV717x_CCED0	0x0E	0x00	Closed Captioning Extended Data Byte 0
ADV717x_CCED1	0x0F	0x00	Closed Captioning Extended Data Byte 1
ADV717x_CCD0	0x10	0x00	Closed Captioning Data Byte 0
ADV717x_CCD1	0x11	0x00	Closed Captioning Data Byte 1
ADV717x_PTCR0	0x12	0x00	NTSC Pedestal Control / PAL TTX Control Register 0
ADV717x_PTCR1	0x13	0x00	NTSC Pedestal Control / PAL TTX Control Register 1
ADV717x_PTCR2	0x14	0x00	NTSC Pedestal Control / PAL TTX Control Register 2
ADV717x_PTCR3	0x15	0x00	NTSC Pedestal Control / PAL TTX Control Register 3
ADV717x_CGMS_WSS0	0x16	0x00	CGMS_WSS Register 0
ADV717x_CGMS_WSS1	0x17	0x00	CGMS_WSS Register 1
ADV717x_CGMS_WSS2	0x18	0x00	CGMS_WSS Register 2
ADV717x_TTX_REQ	0x19	0x00	Teletext request control register
ADV717x_MVR01	0x1E	0x00	Macrovision register 1
ADV717x_MVR02	0x1F	0x00	Macrovision register 2
ADV717x_MVR03	0x20	0x00	Macrovision register 3
ADV717x_MVR04	0x21	0x00	Macrovision register 4
ADV717x_MVR05	0x22	0x00	Macrovision register 5

Register	Address	Default	Description
ADV717x_MVR06	0x23	0x00	Macrovision register 6
ADV717x_MVR07	0x24	0x00	Macrovision register 7
ADV717x_MVR08	0x25	0x00	Macrovision register 8
ADV717x_MVR09	0x26	0x00	Macrovision register 9
ADV717x_MVR10	0x27	0x00	Macrovision register 10
ADV717x_MVR11	0x28	0x00	Macrovision register 11
ADV717x_MVR12	0x29	0x00	Macrovision register 12
ADV717x_MVR13	0x2A	0x00	Macrovision register 13
ADV717x_MVR14	0x2B	0x00	Macrovision register 14
ADV717x_MVR15	0x2C	0x00	Macrovision register 15
ADV717x_MVR16	0x2D	0x00	Macrovision register 16
ADV717x_MVR17	0x2E	0x00	Macrovision register 17
ADV717x_MVR18	0x2F	0x00	Macrovision register 18

Table 7 – ADV717x Registers

7.2. ADV717x register fields

Field	Position	Size	Description
Mode Register 0 (ADV717x_MR0)			
ADV717x_CHROMA_FILTER	5	3	Chroma Filter Select
ADV717x_LUMA_FILTER	2	3	Luma Filter Select
ADV717x_OUT_VIDEO	0	2	Output Video Standard Selection
Mode Register 1 (ADV717x_MR1)			
ADV717x_COLOR_BAR	7	1	Color Bar control
ADV717x_DAC_A	6	1	DAC A Control
ADV717x_DAC_B	5	1	DAC B Control
ADV717x_DAC_C	3	1	DAC C Control
ADV717x_CC_FIELD	1	2	Closed Captioning Field Selection
ADV717x_INTERLACE	0	1	Interlace Control
Mode Register 2 (ADV717x_MR2)			
ADV717x_LOW_POWER	6	1	Low Power Mode selection
ADV717x_BURST_CONTROL	5	1	Burst Control selection
ADV717x_CROM_CONTROL	4	1	Chrominance Control
ADV717x_ACTIVE_LINES	3	1	Active Video Line Duration
ADV717x_GENLOCK	1	2	Genlock Control
ADV717x_SQ_PIXEL	0	1	Square Pixel Control
Mode Register 3 (ADV717x_MR3)			
ADV717x_DEFAULT_COLOR	7	1	Low Power Mode selection
ADV717x_TTXREQ_MODE	6	1	Burst Control selection
ADV717x_TTX_ENABLE	5	1	Teletext Enable

Field	Position	Size	Description
ADV717x_CHROMA_OUT	4	1	Chroma Output Select
ADV717x_DAC_OUT	3	1	DAC Output (SCART / EUROSCART)
ADV717x_VBI_OPEN	2	1	Vertical Blanking Interval output select
Mode Register 4 (ADV717x_MR4)			
ADV717x_SLEEP_MODE	6	1	Sleep mode control
ADV717x_ACTIVE_VIDEO_FILTER	5	1	Active Video Control
ADV717x_PEDESTAL	4	1	Pedestal Control
ADV717x_VSYNC_3H	3	1	VSYNC line control
ADV717x_RGB_SYNC	2	1	Setup RGB outputs
ADV717x_RGB_YUV	1	1	RGB/YUV Control
ADV717x_OUTPUT_SELECT	0	1	Output select (Composite video or RGB/YPbPr mode)
Timing Mode Register 0 (ADV717x_TMR0)			
ADV717x_TIMING_REG_RST	7	1	Timing Register Reset
ADV717x_LUMA_DELAY	4	2	Luma Delay
ADV717x_BLANK_INPUT	3	1	Blank\ Input control
ADV717x_TIMING_MODE	1	2	Timing Mode Selection
ADV717x_MASTER_SLAVE	0	1	Master / Slave control
Timing Mode Register 1 (ADV717x_TMR1)			
ADV717x_HSYNC_ADJUST	6	2	HSYNC\ to Pixel Data Adjust
ADV717x_VSYNC_WIDTH	4	2	HSYNC\ to Field Raising Edge delay
ADV717x_HSYNC_VSYNC_DELAY	2	2	HSYNC\ to Field/VSYNC\ Delay
ADV717x_HSYNC_WIDTH	0	2	HSYNC\ Width
CGMS_WSS Register 0 (ADV717x_CGMS_WSS0)			
ADV717x_WIDESCREEN_SIGNAL	7	1	Wide screen signal control
ADV717x_CGMS_EVEN_FIELD	6	1	CGMS Even field control
ADV717x_CGMS_ODD_FIELD	5	1	CGMS Odd field control
ADV717x_CGMS_CRC_CHECK	4	1	CGMS CRC Check control
ADV717x_CGMS_REG0_DATA	0	4	CGMS_WSS register 0 Data bits (Data bits only for CGMS)
CGMS_WSS Register 1 (ADV717x_CGMS_WSS1)			
ADV717x_CGMS_REG1_DATA	6		CGMS_WSS Register 1 Data bits
ADV717x_CGMS_WSS_DATA	0		CGMS_WSS Data bits (shard by CGMS & WSS)
Teletext Request Control Register (ADV717x_TTX_REQ)			
ADV717x_TTXREQ_RAISING	4	4	Teletext request raising edge control
ADV717x_TTXREQ_FALLING	0	4	Teletext request falling edge control

Table 8 – ADV717x Register Fields

ADV717x Sub-carrier Frequency (SCF) Values

SCF value for ITU-656 NTSC Mode:	0x21F07C1E
SCF value for ITU-656 PAL Mode:	0x2A098ACA

8. Appendix

8.1. Using ADV717x Device Driver in Applications

This section explains how to use ADV717x 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 number of ADV717x device instances + 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. ADV717x (driver) initialization

Step 1: Open ADV717x Device driver with device specific entry point (refer section 6.1 for valid entry points)

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 ADV717x video data flow
Example:
// Set ADV717x to use PPI 0 for video dataflow
```
adi_dev_Control (ADV717xDriverHandle, ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER, (void *) 0);
```

b. ADV717x (hardware) initialization

Step 5: Set ADV717x TWI device address
Example:
// this case, set ADV7171 TWI device address
```
adi_dev_Control(ADV717xDriverHandle, ADI_ADV717x_CMD_SET_TWI_DEVICE_ADDRESS,
                                (void *) 0x2A);
```

Step 6: Configure ADV717x device sub-carrier frequency register (refer section 8.3.1 for examples)

Step 7: Configure ADV717x device to specific mode using device access commands
(refer section 8.3.3 for examples)

c. Video Dataflow configuration

Step 8: Open the above PPI device via ADV717x for video dataflow
Example:
// Open the PPI device for video dataflow

```
adi_dev_Control (ADV717xDriverHandle, ADI_ADV717x_CMD_SET_PPI_STATUS,
                                        (void *) ADI_ADV717x_PPI_OPEN);
```

Step 9: Set video dataflow method

Step10: Load ADV717x video buffers

Step11: Enable ADV717x video dataflow

d. Terminating ADV717x driver

Step12: Terminate ADV717x 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 ADV717x 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 ADV717x registers

This section explains how to access the ADV717x internal registers using driver specific commands and device access commands (refer *'deviceaccess'* documentation for more information).

Refer section 7.1 for list of ADV717x device registers and section 7.2 for list of ADV717x device registers fields

8.3.1. Access ADV717x sub-carrier frequency registers (SCFRs)

```
// location to hold the sub-carrier frequency (SCF) value
u32     SCFRval;

// to read ADV717x SCFRs
adi_dev_Control ( DriverHandle, ADI_ADV717x_CMD_GET_SCF_REG, (void *) &SCFRval );

// to configure ADV717x SCFRs
// load the SCFR value (refer page 20 for SCF values)
SCFRval = 0x21F07C1E;           // ADV717x in ITU-656 NTSC mode

// pass the value to ADV717x driver
adi_dev_Control ( DriverHandle, ADI_ADV717x_CMD_SET_SCF_REG, (void *) SCFRval );
```

8.3.2. Read ADV717x 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 = ADV717x_MR2;

    //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 = ADV717x_MR1;
    // Load the device register field location to be read
    Read_Field.Address = ADV717x_COLOR_BAR;

    //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 [ ] = {
                                    {ADV717x_MR0,          0},
                                    {ADV717x_MR2,          0},
                                    {ADV717x_TMR1,         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 ADV717x_MR0 will be read to Read_Regs[0].Data, ADV717x_MR2 to Read_Regs[1].Data
// and value of ADV717x_TMR1 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 [ ] = {
                              {ADV717x_MR0,       ADV717x_OUT_VIDEO,         0},
                              {ADV717x_MR4,       ADV717x_OUTPUT_SELECT,     0},
                              {ADV717x_MR4,       ADV717x_RGB_YUV,           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 ADV717x_OUT_VIDEO will be read to Read_Fields[0].Data,
// ADV717x_OUTPUT_SELECT to Read_Fields [1].Data and ADV717x_RGB_YUV 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 = 4;
// load the starting address of the register block to be read
Read_Block.Address = ADV717x_MR0;
// define a 'Count' sized array to hold register data read from the device
u16 Block_Data[4] = { 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 ADV717x_MR0 will be read to Block_Data [0], ADV717x_MR1 to Block_Data[1],
// ADV717x_MR2 to Block_Data[2] and ADV717x_MR3 to Block_Data[3]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) & Read_Block);
```

8.3.3. Configure ADV717x internal registers

1. Configure a single ADV717x 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 = ADV717x_MR1;

//Load the configuration value to Cfg_Reg.Data location
Cfg_Reg.Data = 0x58;
// 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 = ADV717x_MR1;
// Load the device register field location to be configured
Cfg_Field.Address = ADV717x_COLOR_BAR;

// 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)
// ADV717x Register Configuration values for ITU-656 NTSC mode
ADI_DEV_ACCESS_REGISTER Cfg_Regs_NTSC [ ] = {
                                   {ADV717x_MR0,        0x00},
                                   {ADV717x_MR1,        0x58},
                                   {ADV717x_MR2,        0x00},
                                   {ADV717x_MR3,        0x00},
                                   {ADV717x_MR4,        0x10},
                                   {ADV717x_TMR0,       0x00},
                                   {ADV717x_TMR1,       0x00},
   /*MUST include delimiter */     {ADI_DEV_REGEND,     0    }    };      // Register access delimiter

// ADV717x Register Configuration values for ITU-656 PAL mode
ADI_DEV_ACCESS_REGISTER Cfg_Regs_PAL [ ] = {
                                   {ADV717x_MR0,        0x05},
                                   {ADV717x_MR1,        0x10},
                                   {ADV717x_MR2,        0x00},
                                   {ADV717x_MR3,        0x00},
                                   {ADV717x_MR4,        0x00},
                                   {ADV717x_TMR0,       0x08},
                                   {ADV717x_TMR1,       0x00},
   /*MUST include delimiter */     {ADI_DEV_REGEND,     0    }    };      // Register access delimiter
```

```
// Application calls adi_dev_Control( ) function with corresponding command and value
// Configure ADV717x in ITU-656 NTSC mode
// Registers listed in the table will be configured with corresponding table Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Regs_NTSC[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 [ ] = {
                              {ADV717x_MR0,      ADV717x_OUT_VIDEO,       0},
                              {ADV717x_MR4,      ADV717x_OUTPUT_SELECT,   1},
                              {ADV717x_MR4,      ADV717x_RGB_YUV,         1},
/*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 = 7;
// load the starting address of the register block to be configured
Cfg_Block.Address = ADV717x_MR0;

// define a 'Count' sized array to hold register data read from the device
// ADV717x Register Configuration values for ITU-656 NTSC mode
u16 Block_Cfg_NTSC [7]   = { 0x00, 0x58, 0x00, 0x00, 0x10, 0x00, 0x00 };
// ADV717x Register Configuration values for ITU-656 PAL mode
u16 Block_Cfg_PAL [7]    = { 0x05, 0x10, 0x00, 0x00, 0x00, 0x08, 0x00 };

// Configure ADV717x in ITU-656 NTSC mode
// load the start address of the above array to Cfg_Block data pointer
Cfg_Block.pData = & Block_Cfg_NTSC [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_Cfg_NTSC[ ] array
adi_dev_Control (DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) &Cfg_Block);
```