ANALOG DEVICES

PLATFORM TOOLS GROUP

ADI_MT9V022 DEVICE DRIVER

DATE: JANUARY 24, 2006

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
2006/01/24	Document created
2006/05/17	Added example code showing use of driver and updated register field table.

Table 1 - Revision History

1. Overview

The driver uses the TWI and PPI device drivers to interface to the Micron MT9V022 VGA video input sensor. The PPI and TWI configuration is fully configurable via the driver controls. Internal registers of the micron 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 MT9V022 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 MT9V022 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/micron/adi_mt9v022.h> This file contains all definitions, function prototypes etc. specific to Micron MT9V022 sensor

2.2. Source Files

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

- adi_mt9v022.c

3. Lower Level Drivers

The MT9V022 driver uses the TWI and PPI. 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 MT9V022 sensor via the TWI.

3.1. TWI

The TWI device driver is used by the MT9V022 driver to read and write to the configuration registers located on the MT9V022 hardware. The TWI device can be configured for use by control commands in the MT9V022 driver (refer section 5.4.3)

3.2. PPI

The PPI device driver is used by the MT9V022 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 MT9V022 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 MT9V022 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 hardware exists. In this case the TWI uses one timer and 2 general purpose flags.

4.1. Interrupts

The MT9V022 does not use any interrupts directly, please see PPI and TWI documentation for resources required by these drivers.

4.2. DMA

The MT9V022 does not use any DMA directly; however check the PPI documentation for DMA resources required by this driver.

4.3. Timers

The MT9V022 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 resources required by this driver.

4.6. Pins

This driver does not use any external 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_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_CMD_SET_STREAMING
  - Enables/disables the streaming mode of the driver.
  - Value – TRUE/FALSE
- ADI_DEV_GET_2D_SUPPORT
  - Determines if the driver can support 2D buffers
  - Value – u32 * (location where TRUE/FALSE is stored)
- ADI_DEV_SET_ERROR_REPORTING
  - Enables/Disables error reporting from the device driver
  - 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_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_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_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_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_GET_MAX_INBOUND_SIZE
  - Returns the maximum number of data bytes for an inbound buffer
  - Value – u32 * (location where the size is stored)

- ADI_DEV_CMD_GET_MAX_OUTBOUND_SIZE
  - Returns the maximum number of data bytes for an outbound buffer
  - Value – u32 * (location where the size is stored)
- ADI_DEV_CMD_FREQUENCY_CHANGE_PROLOG
  - Notifies device driver immediately prior to a CCLK/SCLK frequency change
  - Value – ADI_DEV_FREQUENCIES * (new frequencies)
- ADI_DEV_CMD_FREQUENCY_CHANGE_EPILOG
  - Notifies device driver immediately following a CCLK/SCLK frequency change
  - Value – ADI_DEV_FREQUENCIES * (new frequencies)
- 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_MT9V022_CMD_SET_TWI
  - Set the TWI device number to use
  - Value – u32 (device number)
- ADI_MT9V022_CMD_SET_TWIADDR
  - Set the device TWI address
  - Value – u32 (device address)
- ADI_MT9V022_CMD_SET_TWICONFIG
  - Set the extra configuration options for TWI device
  - Value - ADI_DEV_CMD_VALUE_PAIR * (extra TWI configuration controls)
- ADI_MT9V022_CMD_SET_PPI
  - Set the PPI device number to use
  - Value – u32 (device number)

- ADI_MT9V022_CMD_SET_PPICONFIG
  - Manually configure PPI with command table
  - Value - ADI_DEV_CMD_VALUE_PAIR * (manual PPI configuration controls)
- ADI_MT9V022_CMD_SET_PPICMD
  - Send commands directly to PPI (Note: Only works when MT9V022 dataflow is TRUE)
  - Value - ADI_DEV_CMD_VALUE_PAIR * (directly control PPI device)
- ADI_MT9V022_CMD_GET_HWID
  - Get the revision number of the MT9V022 (Note: TWI needs to be setup before calling this)
  - Value – u32 * (Revision of chip being used, this driver was tested on Rev 0x1313)
- ADI_MT9V022_CMD_GET_HEIGHT
  - Get the height of the MT9V022 window (Note: TWI needs to be setup before calling this)
  - Value – u32 * (Height of window)
- ADI_MT9V022_CMD_GET_WIDTH
  - Get the width of the MT9V022 window(Note: TWI needs to be setup before calling this)
  - Value – u32 * (Width of window)

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 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_MT9V022_RESULT_ALREADYSTOPPED
    - Trying to stop the driver (Dataflow=FALSE) when the driver is not running
- ADI_MT9V022_RESULT_PPI_CLOSED
    - PPI needs to be open to perform the action.

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.

- ADIMT9V022EntryPoint

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.

The MT9V022 driver uses the default configuration values for the PPI and TWI. The TWI is configured with

```
ADI_DEV_CMD_VALUE_PAIR TWIConfig[]={
        {ADI_DEV_CMD_SET_DATAFLOW_METHOD,(void *)ADI_DEV_MODE_SEQ_CHAINED},
        {ADI_DEV_CMD_SET_DATAFLOW,(void *)TRUE},
        {ADI_DEV_CMD_END,(void *)NULL},
};
```

automatically. Any additional commands can be added by using ADI_MT9V022_CMD_SET_TWICONFIG to submit an ADI_DEV_CMD_VALUE_PAIR table.

Item	Default Value	Possible Values	Command ID
TWI device	0	0,1	ADI_MT9V022_CMD_SET_TWI
TWI address	ADI_MT9V022_TWIADDR4	ADI_MT9V022_TWIADDR1 ADI_MT9V022_TWIADDR2 ADI_MT9V022_TWIADDR3 ADI_MT9V022_TWIADDR4	ADI_MT9V022_CMD_SET_TWIADDR
PPI device	0	0,1	ADI_MT9V022_CMD_SET_PPI

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.

Item	Possible Values	Command ID
TWI config	ADI_DEV_CMD_VALUE_PAIR array	ADI_MT9V022_CMD_SET_TWICONFIG
PPI config	ADI_DEV_CMD_VALUE_PAIR array	ADI_MT9V022_CMD_SET_PPICONFIG

Table 5 – Additional Required Settings

7. Hardware Considerations

The MT9V022 device on the MI-350 is configured to use ADI_MT9V022_TWI_ADDR4. If the client intends to use pseudo TWI to access MT9V022 registers, specific port pins should be set in Blackfin to generate TWI SCL and SDA.

7.1. MT9V022 registers

Register	Address	Default	Description
ADI_MT9V022_CHIPVERSION	0x00	0x1313	Chip Version
ADI_MT9V022_COLUMNSTART	0x01	0x0001	Column start
ADI_MT9V022_ROWSTART	0x02	0x0004	Row start
ADI_MT9V022_WINDOWHEIGHT	0x03	0x01E0	Window height
ADI_MT9V022_WINDOWWIDTH	0x04	0x02F0	Window width
ADI_MT9V022_HBLANK	0x05	0x005E	Horizontal blanking
ADI_MT9V022_VBLANK	0x06	0x002D	Vertical blanking
ADI_MT9V022_CHIPCONTROL	0x07	0x0388	Chip control
ADI_MT9V022_SHUTTER1	0x08	0x01BB	Shutter width 1
ADI_MT9V022_SHUTTER2	0x09	0x01D9	Shutter width 2
ADI_MT9V022_SHUTTERCTRL	0x0A	0x0164	Shutter width control
ADI_MT9V022_SHUTTERTOTAL	0x0B	0x01E0	Total shutter width
ADI_MT9V022_RESET	0x0C	0x0000	Reset
ADI_MT9V022_READMODE	0x0D	0x0300	Read mode
ADI_MT9V022_MONITORMODE	0x0E	0x0000	Monitor mode
ADI_MT9V022_PIXELMODE	0x0F	0x0011	Pixel mode
ADI_MT9V022_LEDOUTCTRL	0x1B	0x0000	LED_OUT control
ADI_MT9V022_ADCMODECTRL	0x1C	0x0002	ADC mode control
ADI_MT9V022_VREFADCCTRL	0x2C	0x0004	Vref ADC control
ADI_MT9V022_V1-4	0x31-0x34	0x1D,0x1B,0x15,0x04	
ADI_MT9V022_ANALOGGAIN	0x35	0x0010	Analog gain
ADI_MT9V022_ANALOGGAINMAX	0x36	0x0040	Maximum analog gain
ADI_MT9V022_DARKAVERAGE	0x42	0x0000	Darkness average
ADI_MT9V022_DARKAVGTHRESHOLD	0x46	0x231D	Darkness average threshold
ADI_MT9V022_BLCALIBCTRL	0x47	0x8080	Black level calibration control
ADI_MT9V022_BLCALIBVALUE	0x48	0x0000	Black level calibration value
ADI_MT9V022_BLCALIBSTEP	0x4C	0x0002	Black level calibration step size
ADI_MT9V022_ROWNOISECORRCTRL1	0x70	0x0034	Row noise correction control 1
ADI_MT9V022_ROWNOISECONSTANT	0x72	0x002A	Row noise constant
ADI_MT9V022_ROWNOISECORRCTRL2	0x73	0x02F7	Row noise correction control 2
ADI_MT9V022_PIXELCLK	0x74	0x0000	Pixel clock (FV and LV)
ADI_MT9V022_TESTPATTERN	0x7F	0x0000	Digital test pattern
ADI_MT9V022_TILEX0Y0 - TILEX4Y4	0x80-0x98	0x00F4	Tile weight/gain
ADI_MT9V022_TILECOORDX0 – X5	0x99-0x9E		Tile co-ordinate X 0/5
ADI_MT9V022_TILECOORDY0 – Y5	0x9F-0xA4		Tile co-ordinate Y 0/5
ADI_MT9V022_AECAGCDESIREDBIN	0xA5	0x003A	AEC/AGC desired bin
ADI_MT9V022_AECUPDATEFREQ	0xA6	0x0002	AEC update frequency

Register	Address	Default	Description
ADI_MT9V022_AECLPF	0xA8	0x0000	AEC low pass filter
ADI_MT9V022_AGCUPDATEFREQ	0xA9	0x0002	AGC update frequency
ADI_MT9V022_AGCLPF	0xAB	0x0002	AGC low pass filter
ADI_MT9V022_AECAGCENABLE	0xAF	0x0003	AEC/AGC enable
ADI_MT9V022_AECAGCPIXELCOUNT	0xB0	0xABE0	AEC/AGC pixel count
ADI_MT9V022_LVDSMASTERCTRL	0xB1	0x0002	LVDS master control
ADI_MT9V022_LVDSSHIFTCLKCTRL	0xB2	0x0010	LVDS shift clock control
ADI_MT9V022_LVDSDATACTRL	0xB3	0x0010	LVDS data control
ADI_MT9V022_LVDSLATENCY	0xB4	0x0000	LVDS latency control
ADI_MT9V022_LVDSINTERNALSYNC	0xB5	0x0000	LVDS internal sync
ADI_MT9V022_LVDSPAYLOADCTRL	0xB6	0x0000	LVDS payload control
ADI_MT9V022_STEREOERRORCTRL	0xB7	0x0000	Stereoscope error control
ADI_MT9V022_STEREOERRORFLAG	0xB8	0x0000	Stereoscope error flag
ADI_MT9V022_LVDSDATAOUTPUT	0xB9	0x0000	LVDS data output
ADI_MT9V022_AGCGAINOUTPUT	0xBA	0x0000	AGC gain output
ADI_MT9V022_AECGAINOUTPUT	0xBB	0x0000	AEC gain output
ADI_MT9V022_AECAGCCURRENTBIN	0xBC	0x0000	AEC/AGC current bin
ADI_MT9V022_SHUTTERMAX	0xBD	0x01E0	Maximum shutter width
ADI_MT9V022_AECAGCDIFFBIN	0xBE	0x0014	AEC/AGC bin diff threshold
ADI_MT9V022_FIELDBLANK	0xBF	0x0016	Field blank
ADI_MT9V022_CAPTURECTRL	0xC0	0x000A	Monitor mode capture control
ADI_MT9V022_TEMPERATURE	0xC1	0x0000	Temperature
ADI_MT9V022_ANALOGCTRL	0xC2	0x1840	Analog control
ADI_MT9V022_NTSCCTRL	0xC3	0x3840	NTSC FV & LV control
ADI_MT9V022_NTSCHBLANKCTRL	0xC4	0x4416	NTSC horizontal blank control
ADI_MT9V022_NTSCVBLANKCTRL	0xC5	0x4421	NTSC vertical blanking control
ADI_MT9V022_REGISTERLOCK	0xFE	0xBEEF	Register lock

Table 6 – Device registers

7.2. MT9V022 register fields

Field	Position	Size	Description
CHIPCONTROL register			
ADI_MT9V022_CHIPCONTROLSCAN	0	3	Scan mode
ADI_MT9V022_CHIPCONTROLMASTERSLAVE	3	1	Sensor master/slave mode
ADI_MT9V022_CHIPCONTROLSNAPSHOT	4	1	Sensor snapshot mode
ADI_MT9V022_CHIPCONTROLSTEREO	5	1	Stereoscopy mode
ADI_MT9V022_CHIPCONTROLSTEREOMASTER	6	1	Stereosopic master/slave mode
ADI_MT9V022_CHIPCONTROLPARALLEL	7	1	Parallel output enable
ADI_MT9V022_CHIPCONTROLSEQUENTIAL	8	1	Simultaneous/sequential mode
ADI_MT9V022_CHIPCONTROLPIXELCORR	9	1	Defect pixel correction enable
SHUTTERCTRL register			
ADI_MT9V022_SHUTTERCTRLT2RATIO	0	4	T2 ratio

Field	Position	Size	Description
ADI_MT9V022_SHUTTERCTRLT3RATIO	4	4	T3 ratio
ADI_MT9V022_SHUTTERCTRLEXPOSURE	8	1	Exposure knee point auto adjust enable
ADI_MT9V022_SHUTTERCTRLSINGLEKNEE	9	1	Single knee enable
RESET register			
ADI_MT9V022_RESETSOFT	0	1	Soft reset
ADI_MT9V022_RESETAUTOBLOCK	1	1	Auto block soft reset
READMODE register			
ADI_MT9V022_READMODEROWBIN	0	2	Row bin
ADI_MT9V022_READMODECOLBIN	2	2	Column bin
ADI_MT9V022_READMODEROWFLIP	4	1	Row flip
ADI_MT9V022_READMODECOLFLIP	5	1	Column flip
ADI_MT9V022_READMODEROWDARK	6	1	Show dark rows
ADI_MT9V022_READMODECOLDARK	7	1	Show dark columns
PIXELMODE register			
ADI_MT9V022_PIXELMODECOLOUR	2	1	Colour/Mono
ADI_MT9V022_PIXELMODEDYNRANGE	6	1	High dynamic range
ADI_MT9V022_PIXELMODEEXTEXP	7	1	Enable exended exposure
LEDOUTCTRL register			
ADI_MT9V022_LEDOUTCTRLDISABLE	0	1	Disable LED_OUT
ADI_MT9V022_LEDOUTCTRLINVERT	1	1	Invert LED_OUT
BLCALIBCTRL register			
ADI_MT9V022_BLCALIBCTRLMANUAL	0	1	Manual override
ADI_MT9V022_LEDOUTCTRLINVERT	5	3	Frames to average over
ROWNOISECORRCTRL1 register			
ADI_MT9V022_ROWNOISECORRCTRL1DARKPIX	0	4	Number of dark pixels
ADI_MT9V022_ROWNOISECORRCTRL1NOISE	5	1	Enable noise correction
ADI_MT9V022_ROWNOISECORRCTRL1BLAVG	11	1	Use black level average
PIXELCLK register			
ADI_MT9V022_PIXELCLKINVERTLINE	0	1	Invert line valid
ADI_MT9V022_PIXELCLKINVERTFRAME	1	1	Invert frame valid
ADI_MT9V022_PIXELCLKXORLINE	2	1	Xor line valid
ADI_MT9V022_PIXELCLKCONTLINE	3	1	Continuous line valid
ADI_MT9V022_PIXELCLKINVERTPIXEL	4	1	Invert pixel clock
TESTPATTERN register			
ADI_MT9V022_TESTPATTERNTWIDATA	0	10	Two-wire serial interface test data
ADI_MT9V022_TESTPATTERNTWIENABLE	10	1	Use two-wire serial interface test data
ADI_MT9V022_TESTPATTERNGREY	11	2	Grey shade test pattern
ADI_MT9V022_TESTPATTERNENABLE	13	1	Test enable
ADI_MT9V022_TESTPATTERNTWIFLIP	14	1	Flip two-wire serial interface test data
TILEX?Y? register			
ADI_MT9V022_TILEGAIN	0	4	Tile gain
ADI_MT9V022_TILESAMPLE	4	4	Sample weight
LVDSMASTERCTRL register			
ADI_MT9V022_LVDSMASTERCTRLPLL	0	1	PLL bypass

Field	Position	Size	Description
ADI_MT9V022_LVDSMASTERCTRLPOWERDOWN	1	1	LVDS powerdown
ADI_MT9V022_LVDSMASTERCTRLPLLTEST	2	1	PLL test mode
ADI_MT9V022_LVDSMASTERCTRLTEST	3	1	LVDS test mode
LVDSSHIFTCLKCTRL register			
ADI_MT9V022_LVDSSHIFTCLKCTRLDELAY	0	3	Shift clock delay element select
ADI_MT9V022_LVDSSHIFTCLKCTRLPOWERDOWN	4	1	Shift clock (driver) power-down
LVDSDATACTRL register			
ADI_MT9V022_LVDSDATACTRLDELAY	0	3	Data delay element select
ADI_MT9V022_LVDSDATACTRLPOWERDOWN	4	1	Data (receiver) power-down
STEREOERRORCTRL register			
ADI_MT9V022_STEREOERRORCTRLERRORDETECT	0	1	Enable stereo error detect
ADI_MT9V022_STEREOERRORCTRLERRORSTICKY	1	1	Enable stick stereo error flag
ADI_MT9V022_STEREOERRORCTRLERRORCLEAR	2	1	Clear stereo error flag
ANALOGCTRL register			
ADI_MT9V022_ANALOGCTRLANTIECLIPSE	7	1	Anti-eclipse enable
ADI_MT9V022_ANALOGCTRLREFVOLTAGE	11	3	V_rst_lim voltage level
NTSCCTRL register			
ADI_MT9V022_NTSCCTRLEXTENDFRAME	0	1	Extend frame valid
ADI_MT9V022_NTSCCTRLREPLACEFVLV	1	1	Replace FV/LV with Ped/Sync
NTSCHBLANKCTRL register			
ADI_MT9V022_NTSCHBLANKCTRLPORCH	0	8	Front porch width
ADI_MT9V022_NTSCHBLANKCTRLSYNC	8	8	Sync width
NTSCVBLANKCTRL register			
ADI_MT9V022_NTSCVBLANKCTRLEQUALIZE	0	8	Equalizing pulse width
ADI_MT9V022_NTSCVBLANKCTRLSERRATION	8	8	Vertical serration width

Table 7 – Device register fields

8. Appendix

8.1. Using MT9V022 Device Driver in Applications

This section explains how to use MT9V022 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 MT9V022 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 for devices included in the application

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

a. MT9V022 (driver) initialization

Step 1: Open MT9V022 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 MT9V022 video data flow
Example:
// Set MT9V022 to use PPI 0 for video dataflow
adi_dev_Control (MT9V022DriverHandle, ADI_MT9V022_CMD_SET_PPI, (void *) 0);

b. MT9V022 (hardware) initialization

Step 5: Set MT9V022 TWI device address
Example:
// in this case, set TWI device address as ADI_MT9V022_TWIADDR2
adi_dev_Control(MT9V022DriverHandle, ADI_MT9V022_CMD_SET_TWI,
(void *) ADI_MT9V022_TWIADDR2);

Step 6: Configure MT9V022 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 MT9V022 video buffers

Step 9: Enable MT9V022 video dataflow

d. Terminating MT9V022 driver

Step12: Terminate MT9V022 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 MT9V022 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 MT9V022 registers

This section explains how to access the MT9V022 internal registers using driver specific commands and device access commands (refer *'deviceaccess'* documentation for more information).

Refer section 7.1 for list of MT9V022 device registers and section 7.2 for list of MT9V022 device registers fields

8.3.1. Read MT9V022 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_MT9V022_CHIPVERSION;

//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_MT9V022_CHIPCONTROL;
// Load the device register field location to be read
Read_Field.Address = ADI_MT9V022_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_MT9V022_CHIPVERSION,        0},
                                    {ADI_MT9V022_WINDOWHEIGHT,       0},
                                    {ADI_MT9V022_WINDOWWIDTH,        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_MT9V022_CHIPVERSION will be read to Read_Regs[0].Data,
// ADI_MT9V022_WINDOWHEIGHT to Read_Regs[1].Data
// and value of ADI_MT9V022_WINDOWWIDTH 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_MT9V022_CHIPCONTROL,      ADI_MT9V022_CHIPCONTROLSEQUENTIAL,  0},
            {ADI_MT9V022_PIXELMODE,        ADI_MT9V022_PIXELMODEDYNRANGE,      0},
            {ADI_MT9V022_CHIPCONTROL,      ADI_MT9V022_CHIPCONTROLSNAPSHOT,    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_MT9V022_CHIPCONTROLSEQUENTIAL will be read to Read_Fields[0].Data,
// ADI_MT9V022_PIXELMODEDYN to Read_Fields [1].Data
// and ADI_MT9V022_CHIPCONTROLSNAPSHOT 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_MT9V022_CHIPVERSION;
// 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_MT9V022_CHIPVERSION will be read to Block_Data [0],
// ADI_MT9V022_COLUMNSTART to Block_Data[1] and ADI_MT9V022_ROWSTART to Block_Data[2]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) & Read_Block);
```

8.3.2. Configure MT9V022 internal registers

1. Configure a single MT9V022 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_MT9V022_CHIPCONTROL;

//Load the configuration value to Cfg_Reg.Data location
Cfg_Reg.Data = 0x038A;
// 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_MT9V022_PIXELMODE;
// Load the device register field location to be configured
Cfg_Field.Address = ADI_MT9V022_PIXELMODEDYNRANGE;

// 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 Micron MT9V022 sensor to output 10bit greyscale video signal
u8 Scale = 1;        // output scale factor
ADI_DEV_ACCESS_REGISTER Cfg_Regs [ ] = {
                              {ADI_MT9V022_READMODE,        0x0330|Scale|(Scale<<2)    },
                              {ADI_MT9V022_CHIPCONTROL,     0x038A                     },
                              {ADI_MT9V022_COLUMNSTART,     16                         },
                              {ADI_MT9V022_ROWSTART,        6                          },
                              {ADI_MT9V022_WINDOWWIDTH,     720                        },
                              {ADI_MT9V022_WINDOWHEIGHT,    480                        },
/*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
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Regs [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_MT9V022_CHIPCONTROL,     ADI_MT9V022_CHIPCONTROLSEQUENTIAL,  0},
            {ADI_MT9V022_PIXELMODE,       ADI_MT9V022_PIXELMODEDYNRANGE,      1},
            {ADI_MT9V022_CHIPCONTROL,     ADI_MT9V022_CHIPCONTROLSNAPSHOT,    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 = 4;
// load the starting address of the register block to be configured
Cfg_Block.Address = ADI_MT9V022_COLUMNSTART;

// define a 'Count' sized array to hold register data read from the device
// Register Configuration values
u16 Block_Data [4] = { 16, 6, 720, 480 };

// 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);
```