ANALOG DEVICES

ADI_PPI DEVICE DRIVER

DATE: FEBRUARY 20, 2006

Table of Contents

Document Revision History

Date	Description of Changes
2006-1-2	Initial version
2007-11-14	Modifed TIMERS section. ADI_PPI_FS_TMR struct has new member enable_delay.

1.Overview

This document describes use of the Parallel Peripheral Interface (PPI) device driver.

The PPI on the Blackfin processor is a half-duplex bi-directional data port. It accommodates up to 16 bits of data, and connects directly to such peripherals as parallel analog to digital (A/D) converters, digital to analog (D/A) converters, video encoders and video decoders.

The PPI device driver is not interrupt driven, but uses the Direct Memory Access (DMA) services, as explained later in this document.

The PPI device driver has been tested on the ADSP-BF533, ADSP-BF561 EZ-Kit Lite development boards, and on the ADSP-BF537 EZ-Kit Lite development board with A/V extender card.

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 definitions and function prototypes for the System Services.
<drivers/adi_dev.h> This file contains definitions and function prototypes for the Device Manager and general device driver information.
<drivers/ppi/adi_ppi.h> This file contains command codes, event codes and return codes, specific to the PPI device driver.

2.2.Source Files

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

<Blackfin/lib/src/drivers/ppi/adi_ppi.c> This file contains all the source code for the PPI device driver. All source code is written in 'C'. There are no assembly level functions in this driver.

3.Lower Level Drivers

The PPI device driver does not use any lower level device drivers.

4.Resources Required

Device drivers typically consume some amount of system resources, such as memory. This section describes the resource requirements of the PPI device driver.

Unless explicitly noted in the sections below, the PPI 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_ppi_Init()).

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

4.1.Interrupts

For each PPI driver that is opened, only one interrupt is used: the error interrupt.

Unless overridden with the appropriate “SetIVG” commands, the error interrupt for the PPI device driver uses the default, power-up, Interrupt Vector Group (IVG) mapping for the specific processor.

The PPI device driver hooks or unhooks the error interrupt handler when the client calls the ‘adi_dev_Control()’ function, with the command: ADI_DEV_CMD_SET_ERROR_REPORTING. If the command is accompanied by an argument of TRUE, the error interrupt is enabled, and the error interrupt handler is hooked into the IVG chain. If the command is accompanied by an argument of FALSE, the error interrupt is disabled, and the interrupt handler is unhooked from the IVG chain. When the client closes the driver by calling ‘adi_dev_close()’, the error interrupt, if enabled and hooked, is automatically disabled and unhooked.

4.2.DMA

This section will explain how to use the PPI device driver in conjunction with the Direct Memory Access (DMA) services, to pass data to and from peripheral devices.

One DMA channel should be allocated for each PPI driver that is opened. If the processor has two PPI ports, and they are used simultaneously, perhaps one for input and the other for output, then two DMA channels should be allocated and initialized. In a multi-core environment, each core might use a separate PPI device and DMA channel, allowing one core to process the input and the other core to process the output.

4.3.Timers

The PPI device driver uses the Timer Control service to access the timer resources for frame sync functionality.

When using frame sync 1 or frame sync 2, the client calls the function ‘adi_dev_Control()’ with the command ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_1 or ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_2, along with a pointer to a ADI_PPI_FS_TMR configuration structure which contains the following fields:

pulse - pulse level, 0 indicates low pulse and 1 indicates high pulse.

emu_run - flag specifying whether the timer will continue running when the processor is stopped, during emulation. A value of 1 indicates that the counter continues to run, 0 indicates it does not.
period - the pulse period (number of PPI_CLK cycles between the start of each pulse)
width - pulse width (number of PPI_CLK cycles within the active level of each pulse)
enable_delay – the number of PPI CLKs that a sync signal must be delayed before it can be enabled. If there are no delay requirements this field must be set to zero.

When three frame syncs are required, the client calls the function 'adi_pdd_Control()' with the argument ADI_PPI_CMD_SET_TRIPLE_FRAME_SYNC causing the 'TripleFrameSyncFlag' flag to be set.

4.4.Real-Time Clock

The PPI device driver does not use any real-time clock services.

4.5.Programmable Flags

4.6.Pins

The PPI has a dedicated clock pin, three frame sync pins, and 8 dedicated data pins. On processors where pin multiplexing is used, eight of the programmable flag (PF) pins can be reconfigured to provide eight additional data pins for the PPI, for 16 bit data width. The clock pin (PPIx_CLK) does not source a clock signal. It only receives external clock input.

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.
ADI_DEV_ DIRECTION_OUTBOUND	Supports the transmission of data out through the device.

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 loop back method

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
- One-dimensional buffer
- pAdditionalInfo – optional

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 an 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 5.1.
- 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_SET_ERROR_REPORTING
- Enables/Disables error reporting from the device driver
- Value – TRUE/FALSE

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_8BIT_BLOCK_READ
- Read a block of 8-bit registers from a device
- Value – ADI_DEV_8BIT_BLOCK * (register specifics)

ADI_DEV_CMD_8BIT_BLOCK_WRITE
- Write a block of 8-bit registers to a device
- Value – ADI_DEV_8BIT_BLOCK * (register specifics)

ADI_DEV_CMD_8BIT_BLOCK_READ_FIELD
- Read specific fields in a block of 8-bit registers from a device
- Value – ADI_DEV_8BIT_BLOCK_FIELD * (register and field specifics)

ADI_DEV_CMD_8BIT_BLOCK_WRITE_FIELD
- Write specific fields in a block of 8-bit registers to a device
- Value – ADI_DEV_8BIT_BLOCK_FIELD * (register and field specifics)

ADI_DEV_CMD_8BIT_SELECTIVE_READ
- Read a selective set of 8-bit registers from a device

- Value – ADI_DEV_8BIT_SELECTIVE * (register specifics)

ADI_DEV_CMD_8BIT_SELECTIVE_WRITE
- Write a selective set of 8-bit registers to a device
- Value – ADI_DEV_8BIT_SELECTIVE * (register specifics)

ADI_DEV_CMD_8BIT_SELECTIVE_READ_FIELD
- Read specific fields from a selective set of 8-bit registers from a device
- Value – ADI_DEV_8BIT_SELECTIVE_FIELD * (register specifics)

ADI_DEV_CMD_8BIT_SELECTIVE_WRITE_FIELD
- Write specific fields from a selective set of 8-bit registers to a device
- Value – ADI_DEV_8BIT_SELECTIVE_FIELD * (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_PPI_CMD_SET_CONTROL_REG
- Sets the PPI Control Register.
- Value – ADI_PPI_CONTROL_REG

ADI_PPI_CMD_SET_DELAY_COUNT_REG
- Sets the delay count.
- Value – u16

ADI_PPI_CMD_SET_TRANSFER_COUNT_REG
- Sets the transfer count.
- Value – u16

ADI_PPI_CMD_SET_LINES_PER_FRAME_REG
- Sets the lines per frame.
- Value – u16

ADI_PPI_CMD_SET_FS_INVERT
- inverts frame sync polarity by setting 1-bit field within control register structure
- Value = TRUE/FALSE, TRUE = invert, FALSE = do not invert.

ADI_PPI_CMD_SET_CLK_INVERT
- inverts frame sync and PPI clock polarity, by setting 1-bit field within control register structure
- Value = TRUE/FALSE, TRUE = invert, FALSE = do not invert.

ADI_PPI_CMD_SET_DATA_LENGTH
- Sets data length, by setting 3-bit field within control register structure.
- Value – u16e

ADI_PPI_CMD_SET_SKIP_EVEN_ODD
- skips even elements, by setting 1-bit field within control register structure
- Value u16, 1 = skip even, 0 = skip odd

ADI_PPI_CMD_SET_SKIP_ENABLE
- Controls skipping.
- Value = u16, 1 = enable skipping, 0 = disable skipping

ADI_PPI_CMD_SET_PACK_ENABLE
- Enables/Disables packing by setting a 1-bit field within control register structure
- Value = u16, 0 = disable packing, 1 = enable packing for input mode, unpacking for output mode

ADI_PPI_CMD_SET_ACTIVE_FIELD_SELECT
- Selects active fields.
- Value = type u16. In ITU-R 656 mode, when xfr_type = 00, 0 = field 1, 1 = field 1 and 2

In RX mode with external frame sync, when port_cfg = 11, 0 = external trigger, 1 = internal trigger

ADI_PPI_CMD_SET_PORT_CFG
- Sets port configuration by setting 2-bit field within ADI_PPI_CONTROL_REG structure
- In non-ITU-R 656 input modes (PORT_DIR=0, XFR_TYPE=11): 00=1external frame sync, 01 = 2 or 3 internal frame syncs, 10 = 2 or 3 external frame syncs, 11 = no frame syncs, triggered. In Output modes with frame syncs (PORT_DIR=0, XFR_TYPE=11): 00=1 frame sync, 01 = 2 or 3 frame syncs, 10 – reserved, 11 – Sync PPI_FS3 to assertion of PPI_FS2 rather than assertion of PPI_FS1.

ADI_PPI_CMD_SET_TRANSFER_TYPE
- Selects transfer type, by setting 2-bit field within ADI_PPI_CONTROL_REG structure
- Value = u16 bit field, 0 = ITU-R 656 active field only, no frame sync, 1 = ITU-R 656 entire field, , no frame sync 2 = ITU-R 656 vertical blanking only, , no frame sync, 3 = non-ITU-R 656 and port_cfg determines frame syncs

ADI_PPI_CMD_SET_PORT_DIRECTION
- Sets port direction by setting 1-bit field within ADI_PPI_CONTROL_REG structure
- Value = TRUE for outbound, FALSE for inbound, sets/clears value in config reg

ADI_PPI_CMD_SET_TRIPLE_FRAME_SYNC
- Sets triple frame sync flag
- Value = TRUE to set flag, FALSE to clear flag

ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_1
- Sets timer for frame sync 1 (PPI_FS1)
- Value = ADI_PPI_FS_TMR *

ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_2
- Sets timer for frame sync 2 (PPI_FS2)
- Value = ADI_PPI_FS_TMR *

5.5.Callback Events

This section enumerates the callback events that the PPI device driver generates. These events fall into two categories: events that are common to multiple device drivers, and driver specific events. The event ID's are divided into two sections, accordingly. The client should prepare it's 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 the PPI 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.

ADI_PPI_EVENT_ERROR_INTERRUPT
- The driver detected a PPI error
- Value – NULL

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 PPI 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 the PPI 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.

.

5.6.2.Device Driver Specific Return Codes

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

ADI_PPI_RESULT_TIMER_ERROR

- An error was detected when attempting to control and configure the timer for frame sync.

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 the PPI driver is listed below.

```
ADI_DEV_PDD_ENTRY_POINT ADIPPIEntryPoint = {
        adi_pdd_Open,
        adi_pdd_Close,
        adi_pdd_Read,
        adi_pdd_Write,
        adi_pdd_Control
};
```

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
PPI control Reg	0	See individual fields	ADI_PPI_CMD_SET_CONTROL_REG
port enable	0	0=disabled, 1=enabled	ADI_DEV_CMD_SET_DATAFLOW
direction	0	0=input, 1=output	ADI_PPI_CMD_SET_PORT_DIRECTION
transfer type	0	0,1,2,3 (see commands)	ADI_PPI_CMD_SET_TRANSFER_TYPE
port config	0	0,1,2,3 (see commands)	ADI_PPI_CMD_SET_PORT_CFG
active field	0	0,1 (see commands)	ADI_PPI_CMD_SET_ACTIVE_FIELD_SELECT
packing	0	0=disabled, 1=enabled	ADI_PPI_CMD_SET_PACK_ENABLE
32-bit DMA	0	0=disabled, 1=enabled	ADI_PPI_CMD_SET_CONTROL_REG
transfer count	0	0-65535	ADI_PPI_CMD_SET_TRANSFER_COUNT_REG
delay count reg	0	0-65535	ADI_PPI_CMD_SET_DELAY_COUNT_REG
lines/frame	0	0-65535	ADI_PPI_CMD_SET_LINES_PER_FRAME_REG
triple frame sync	FALSE	TRUE/FALSE	ADI_PPI_CMD_SET_TRIPLE_FRAME_SYNC
timer open FS1	FALSE	TRUE/FALSE	ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_1
timer open FS2	FALSE	TRUE/FALSE	ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_2
data length	0	0-7 (8-bit to 16-bit width)	ADI_PPI_CMD_SET_DATA_LENGTH
skip even, odd	0	1=skip even, 0=skip odd	ADI_PPI_CMD_SET_SKIP_EVEN_ODD
enable skipping	0	1=enable, 0=disable	ADI_PPI_CMD_SET_SKIP_ENABLE
clock invert	0	1=inverted, 0=not	ADI_PPI_CMD_SET_CLK_INVERT
frame sync invert	0	1=inverted, 0=not	ADI_PPI_CMD_SET_FS_INVERT

6.3.Additional Required Configuration Settings

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

Item	Possible Values	Command ID
Control Reg	See individual fields	ADI_PPI_CMD_SET_CONTROL_REG
Dataflow	1=Enable; 0=Disable	ADI_DEV_CMD_SET_DATAFLOW
delay count	0-65535	ADI_PPI_CMD_SET_DELAY_COUNT_REG
transfer count	0-65535	ADI_PPI_CMD_SET_TRANSFER_COUNT_REG
lines per frame	0-65535	ADI_PPI_CMD_SET_LINES_PER_FRAME_REG
invert frame sync polarity	1=invert, 0=don't	ADI_PPI_CMD_SET_FS_INVERT
invert clock	1=invert, 0=don't	ADI_PPI_CMD_SET_CLK_INVERT
data length	0-7 (8-bit to 16-bit width)	ADI_PPI_CMD_SET_DATA_LENGTH
skip even, odd elements	1=skip even, 0=skip odd	ADI_PPI_CMD_SET_SKIP_EVEN_ODD
control skipping	1=enable, 0=disable	ADI_PPI_CMD_SET_SKIP_ENABLE
enable packing	0=disabled, 1=enabled	ADI_PPI_CMD_SET_PACK_ENABLE
select active fields	TRUE/FALSE	ADI_PPI_CMD_SET_ACTIVE_FIELD_SELECT
port configuration	0-3	ADI_PPI_CMD_SET_PORT_CFG
transfer type	0-3	ADI_PPI_CMD_SET_TRANSFER_TYPE
port direction	0=input, 1=output	ADI_PPI_CMD_SET_PORT_DIRECTION
triple frame syncs	TRUE/FALSE	ADI_PPI_CMD_SET_TRIPLE_FRAME_SYNC
timer for frame sync 1	TRUE/FALSE	ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_1
timer for frame sync 2	TRUE/FALSE	ADI_PPI_CMD_SET_TIMER_FRAME_SYNC_2

7.Hardware Considerations

Note that when using frame sync, the general purpose timer to use is pre-determined by the processor, and the number of frame syncs desired. The specific Timer ID that is used, can be seen in the "Device" structure definition, in 'adi_ppi.c'.

Some processors allow pin multiplexing of input and output data pins with programmable flag (PF) pins, so that the PPI can be configured for 8 bit data width, or borrow 8 PF pins, for 16 bit data width. The user must use caution to insure that the PPI does not use any PF pins used by any other general purpose I/O device, and vice-versa.