



PXDAC4800
Operator's Manual
Revision 1.71

Vitretek LLC
900 N. State Street
Lockport, IL 60441 USA
Tel: (800) 567 - 4243
Fax: (800) 780 - 8411

<http://www.signatec.com>

June 29th, 2022

Table of Contents

1 BEFORE USING THE PXDAC4800!!	6
1.1 Package Contents	6
1.2 Unpacking and Handling	6
1.3 Checking for Damage	6
1.4 Warranty	6
1.5 System Requirements	6
1.6 Software	7
1.7 Physical Layout	7
2 FUNCTIONAL DESCRIPTION	8
2.1 Overview	8
2.2 Programmable Hardware Settings	9
2.2.1 Active Output Channels Selection	9
2.2.2 Playback Clock Selection	10
2.2.2.1 Internal Clocks	10
2.2.2.2 External Clock	10
2.2.2.3 Clock Division	11
2.2.3 Trigger Selection	11
2.2.3.1 Software Generated Trigger	12
2.2.3.2 Trigger Mode	12
2.2.4 Active Memory Region	12
2.2.4.1 Byte Offset	12
2.2.4.2 Byte Count	13
2.2.4.3 Configuring the Active Memory Region	13
2.2.5 Operating Mode	14
2.2.5.1 Standby Mode	14
2.2.5.2 Load RAM Mode	14
2.2.5.3 Streaming Playback Mode	14
2.2.5.4 RAM Playback Mode	15
2.3 PXDAC4800 Playback Data Format	15
2.3.1 Sample Format	15
2.3.2 Multichannel Data Format	15
2.3.3 PXDAC4800 Playback Data Files (*.rd16, *.rd8)	16
2.3.4 Signatec Recorded Data Context Files (*.srdc)	16
2.4 Maximum DAC clock frequency	16
2.4.1 Clock frequency with 8-bit Data Format	17
2.4.2 Clock frequency with 14-bit Data Format	17
3 PXDAC4800 SOFTWARE DEVELOPMENT REFERENCE	17
3.1 Contents	17
3.2 Installing Software	17
3.3 Developing PXDAC4800 Software	18
3.3.1 Setting up the Build Environment – Windows Platform	18
3.3.2 Library Functions That Use Character Strings	18
3.3.3 Active Channel Masks (How Playback Channels are Selected)	19
3.4 PXDAC4800 Library Functions	20
3.4.1 Library Utility Functions	20
3.4.1.1 CalculateCycleCountsXD48	20
3.4.1.2 DumpLibErrorXD48	21
3.4.1.3 GetErrorMessageXD48	22
3.4.1.4 GetErrorTextXD48	23
3.4.1.5 GetFPGAStatusXD48	24
3.4.1.6 GetMaxByteCountForActiveChanMaskXD48	25
3.4.1.7 GetOutputVoltageRangeVoltsXD48	26
3.4.1.8 SampleSizeFromTypeXD48	27
3.4.1.9 SetUserDataXD48	27

3.4.1.10 ValidateConfigEepromXD48	28
3.4.1.11 IsDcXD48	29
3.4.2 PXDAC4800 Device Enumeration and Connection Management	29
3.4.2.1 ConnectToDeviceXD48	29
3.4.2.2 ConnectToVirtualDeviceXD48	30
3.4.2.3 DuplicateHandleXD48	31
3.4.2.4 DisconnectFromDeviceXD48	32
3.4.2.5 GetDeviceCountXD48	32
3.4.2.6 IsDeviceRemoteXD48	33
3.4.2.7 IsDeviceVirtualXD48	33
3.4.2.8 IsHandleValidXD48	34
3.4.3 PXDAC4800 Device State and Configuration	34
3.4.3.1 GetBoardNameXD48	34
3.4.3.2 GetBoardRevisionXD48	35
3.4.3.3 GetItemVersionXD48	36
3.4.3.4 GetOrdinalNumberXD48	37
3.4.3.5 GetPlaybackClockRateXD48	38
3.4.3.6 GetPlaybackDataRateXD48	38
3.4.3.7 GetSampleRamSizeXD48	39
3.4.3.8 GetSerialNumberXD48	40
3.4.3.9 GetVersionTextXD48	40
3.4.3.10 ReadConfigEepromXD48	41
3.4.3.11 WriteConfigEepromXD48	42
3.4.4 PXDAC4800 Device Status	43
3.4.4.1 GetDacOutputFifoFullFlagXD48	43
3.4.4.2 GetDriverStatsXD48	44
3.4.4.3 GetSamplesCompleteInterruptCountXD48	44
3.4.4.4 InIdleModeXD48	45
3.4.4.5 InPlaybackModeXD48	46
3.4.5 PXDAC4800 Hardware Settings	46
3.4.5.1 IssueSoftwareTriggerXD48	46
3.4.5.2 SetActiveChannelMaskXD48	47
3.4.5.3 SetClockDividerVXD48	48
3.4.5.4 SetDacInterpolationEnableXD48	49
3.4.5.5 SetDacSampleFormatXD48	50
3.4.5.6 SetDacSampleSizeXD48	50
3.4.5.7 StartDacAutoCalibrationXD48	51
3.4.5.8 SetExternalPlaybackClockRateXD48	52
3.4.5.9 SetExternalReferenceClockEnableXD48	53
3.4.5.10 SetExternalTriggerDirXD48	53
3.4.5.11 SetExternalTriggerEnableXD48	54
3.4.5.12 SetDigitalIoCfgXD48	55
3.4.5.13 SetDigitalIoModeXD48	56
3.4.5.14 SetOutputVoltageChNXD48	57
3.4.5.15 SetPlaybackClockSourceXD48	58
3.4.5.16 SetTriggerModeXD48	59
3.4.5.17 _SetActiveMemoryRegionXD48	60
3.4.5.18 _SetOperatingModeXD48	62
3.4.5.19 SetFiltersCHXD48	63
3.4.5.20 SetCustomDacValueEnableXD48	64
3.4.5.21 SetCustomDacDefaultValueXD48	65
3.4.6 Device Register State Functions	65
3.4.6.1 CopyHardwareSettingsXD48	66
3.4.6.2 SetPowerupDefaultsXD48	66
3.4.7 Memory/DMA Buffer Allocation Routines (Advanced users ONLY)	68
3.4.7.1 AllocateDmaBufferXD48	68
3.4.7.2 EnsureUtilityDmaBufferXD48	69
3.4.7.3 FreeDmaBufferXD48	70
3.4.7.4 FreeMemoryXD48	70
3.4.7.5 FreeUtilityDmaBufferXD48	71
3.4.7.6 GetUtilityDmaBufferXD48	72
3.4.8 Data Playback Routines	72
3.4.8.1 BeginRamPlaybackXD48	72
3.4.8.2 BeginStreamingPlaybackXD48	74

3.4.8.3 EndRamPlaybackXD48	75
3.4.8.4 EndStreamingPlaybackXD48	76
3.4.8.5 IsPlaybackInProgressXD48	76
3.4.9 Data Transfer Routines	77
3.4.9.1 LoadFileIntoRamXD48	77
3.4.9.2 LoadRamBufXD48	78
3.4.9.3 NotifyAllStreamDataUploadedXD48	80
3.4.9.4 IsTransferInProgressXD48	80
3.4.9.5 WaitForTransferCompleteXD48	81
3.4.10 Data Manipulation Routines	82
3.4.10.1 DeInterleaveDataXbitYChanXD48	82
3.4.10.2 InterleaveDataXbitYChanXD48	84
3.4.11 PXDAC4800 Streaming Playback Session Routines	85
3.4.11.1 SessionStreamCreateXD48	85
3.4.11.2 SessionStreamCreateParmsXD48	86
3.4.11.3 _SessionStreamCreateStdXD48	88
3.4.11.4 SessionStreamDeleteXD48	90
3.4.11.5 SessionStreamDeleteNoStructXD48	90
3.4.11.6 SessionStreamEndXD48	91
3.4.11.7 SessionStreamEndNoStructXD48	92
3.4.11.8 SessionStreamProgressXD48	92
3.4.11.9 SessionStreamGetProgressXD48	93
3.4.11.10 SessionStreamSnapshotXD48	95
3.4.11.11 _SetStreamingLengthXD48	96
3.5 PXDAC4800 Library Data Types	97
3.5.1 Data Type: HXD48	97
3.5.2 Data Type: HXD48STREAM	97
3.5.3 Structure: XD48S_CYCLE_CALC_CTX	97
3.5.4 Structure: XD48S_DRIVER_STATS	99
3.5.5 Structure: XD48S_RECORDED_DATA_INFO	100
3.5.6 Structure: XD48S_STREAM_SES_CREATE	102
3.5.7 Structure: XD48S_STREAM_SES_PROG	104
4 APPENDIX A – PXDAC4800 LIBRARY ERROR CODES	106
5 APPENDIX B – REVISION HISTORY	110

IMPORTANT NOTICE
ON
HARDWARE COMPATIBILITY

The PXDAC4800 is a PCI Express x8 (PCIe) local bus compliant device. As such the PXDAC4800 contains the configuration space register organization as defined by the PCI Express local bus specification 2.1. Among the functions of the configuration registers is the storage of unique identification values for the PXDAC4800 as well as storage of base address size requirements for PXDAC4800 operation.

The host computer that the PXDAC4800 is installed in is responsible for reading and writing to/from the PCIe configuration registers to enable proper operation. This functionality is referred to as 'Plug and Play' (PnP). As such, the host computer PnP BIOS must be capable of automatically identifying a PCI compliant device, determining the system resources required by the device, and assigning the necessary resources to the device. Failure of the host computer to execute any of these operations will prohibit the use of the PXDAC4800 in such a system.

It has been determined that systems that implement PnP BIOS, and contain only fully compliant PnP boards and drivers, operate properly. However, systems that do not have a PnP BIOS installed, or contain hardware or software drivers that are not PnP compatible, may not successfully execute PnP initialization. This can render the PXDAC4800 inoperable. It is beyond the ability of Signatec hardware or software to force a non-PnP system to operate a PXDAC4800 board.

1 Before Using the PXDAC4800!!

1.1 Package Contents

The PXDAC4800 package shipped to you will normally contain (as a minimum) the following:

- PXDAC4800 circuit board assembly
- A CD or DVD containing the following:
 - PXDAC4800 Operator's Manual (this document)
 - PXDAC4800 Software Installation Disk

1.2 Unpacking and Handling

The PXDAC4800, and any other electronic circuit board assembly included in its shipment, is shipped in an anti-static bag. These circuit boards are extremely sensitive to static electricity that can damage sensitive electronic parts. The human body can build up a damaging amount of electrical charge, especially in dry weather and in carpeted rooms. To avoid damaging the boards, rid yourself of any charge buildup by touching some large metal object which ideally is at earth potential. Also, only touch the circuit boards along the edges (excluding the PCIe connector), carefully avoiding contact with the electronic circuitry.

1.3 Checking for Damage

Carefully inspect the circuit board assemblies for any sign of physical damage during shipment. Any such damage must be reported within 15 days from the date of actual shipment in order to be covered by warranty. To report such damage, call Signatec, explain the nature of the damage, and request a RMA number for returning the merchandise.

1.4 Warranty

All Signatec manufactured products carry a full 1-year warranty. During the warranty period, Signatec will repair or replace any defective product at no cost to the customer. This warranty does not cover physical damage not reported within 15 days of the time of shipment by Signatec. Call Signatec for a RMA number before returning any merchandise.

1.5 System Requirements

Signatec recommends systems utilizing Intel based “workstation/server” class chipsets and CPUs for best performance with Signatec products with components such as:

- Windows 7 64-bit Operating System. A 64-bit operating system is necessary to utilize greater than 4+GB of system RAM.
- Intel Xeon 5400/5500/5600 Series CPUs
- Intel 5000P/5000X/5400/5500/5520 Series based Chipsets
- 8+ GB of Total System RAM
- Motherboards utilizing dedicated (non-shared bus) PCI-X 64-bit slots or PCI-Express (PCIe) x8/x16 slots that match desired Signatec product model board's interface.
- Note that high-end graphics cards are not necessary for Signatec board operation.
- For high-speed data storage and retrieval, high-performance RAID cards with a PCI-Express based interface that can connect multiple high-performance SATA/SAS hard drives in a RAID level configuration.

Please note that Signatec products are capable of operating with lower end “desktop” class systems as well; they just may not be able to utilize the maximum data transfer performance rates that Signatec products are capable of reaching as these lower end class systems typically use less performance based chipsets and interconnections limiting the bandwidth of the interfaces.

The PXDAC4800 requires the following “minimum” hardware configuration:

- The availability of 1 open PCI Express (PCIe) x8 slot on the system motherboard for PXDAC4800 board installation.
 - Note: Some motherboards have “mechanical” PCIe x8 or x16 slots that actually operate “electrically” at slower PCIe x4 rates. The PXDAC4800 is capable of being operated in such slots, but its maximum transfer rate will be reduced by the slower PCIe x4 connection.
- A total of 2GB system RAM.
- Supported 32-bit or 64-bit operating systems:
 - Windows XP / Windows Vista / Windows 7
 - Windows Server 2003 & 2003 R2 / Windows Server 2008 & 2008 R2

Note: DOS, Windows 3.x/95/98/ME, Windows NT 3.x/4.x, Windows Server 2000 are **NOT** supported by the PXDAC4800 software.

The Linux platform is not supported at the time of this writing.

1.6 Software

The PXDAC4800 is supplied with all software required to use PXDAC4800 devices on local or remote systems. The software will install the following items:

- Kernel mode driver – This is small piece of kernel-mode software that interfaces the hardware on behalf of the operating system and requests from the user-mode library.
- User mode library – All user-mode device access goes through this library. This is the only interface to the PXDAC4800 device driver, which in turn, is the only interface to the underlying Signatec hardware. Other platforms (e.g. .NET, LabVIEW, Matlab, etc) all interface through this library. This library exports a number of functions that may be used to interact with PXDAC4800 devices. These functions are documented in the [PXDAC4800 Software Development Reference](#) section of this manual.
- PXDAC4800 Playback Application – This is a Windows application that allows the user to interact with one or more PXDAC4800 devices and can work with both local (devices physically present in local system) and remote (devices located on a remote system connected via a standard TCP/IP network). The application can be used to experiment with hardware settings, view and play back data. The PXDAC4800 Playback Application Manual document details the behavior of the PXDAC4800 Playback Application.
- Source code for various example applications that show how to use the PXDAC4800 library to control PXDAC4800 devices.

1.7 Physical Layout

The PXDAC4800 is shown in Figure 1. Signals are input via the SMA connectors on the bracket as shown. The upper connector is for an external clock, the second connector is for digital I/O, the third connector is a trigger input, and connectors 4 through 7 are for the analog output signals.

At the top of the board in the far left corner is a JTAG header that is available for Xilinx FPGA user programmability. Other top connectors are for internal use only.

The PXDAC4800 employs a PCIe x8 bus via the bottom card edge connector. This allows the PXDAC4800 to be plugged into all existing PCIe slots that have an 8-lane (x8) or higher mechanical connection even if the electrical connection is for fewer lanes.

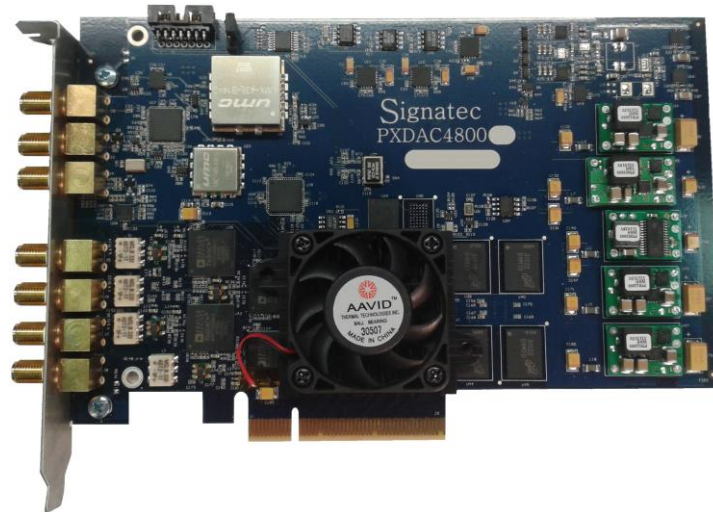


Figure 1: PXDAC4800 Physical Layout

2 Functional Description

2.1 Overview

The PXDAC4800 is a very high speed, four channel Digital to Analog Conversion (DAC) board which may be used as an Arbitrary Waveform Generator, a waveform playback device, or for generating multiple communication frequency bands. Each DAC can output data at a maximum real rate of 1200 Megasamples per second. This allows for up to 600 MHz of bandwidth for each output signal. Output waveforms may be “single shot” or “continuously looped” from the on-board 1 gigabyte memory.

The DACs can also be provided with a continuous data stream via the PCI Express bus. The 1400 Megabytes per second bus rate allows for providing up to 87 MHz of bandwidth for each of the four analog outputs. By using the supplied up-conversion capabilities, this frequency band can be placed anywhere in the dc to 600 MHz frequency space.

The DAC clock source may be selected as either the 1200 MHz or 900 MHz VCXO oscillator or from the external clock input. Any of the three clock sources can be divided by any integer from 1 to 32. The VCXO oscillators have extremely low jitter and can be synchronized to an external reference input or to the internal TCXO reference. The internal reference is accurate to better than 3ppm. Fixed frequency oscillators have been chosen instead of a wide-band frequency synthesizer based on providing an SNR benefit of approximately 10 dB when producing very high frequency output signals.

The PXDAC4800 is a nice complement to Signatec’s PX1500-4 waveform acquisition board and is ideal for playing back extremely long waveforms, such as those previously captured with a Signatec Waveform Recording system. This is particularly useful in testing a system’s response to previously recorded real world signals. In

such an operation the data is supplied from the high-speed data storage system via the PCI Express bus and the PXDAC4800's on-board RAM is utilized as a very large FIFO data buffer.

The PXDAC4800 has seven external signal connections: a clock input, a trigger input, 4 analog input signals, and a user-selectable digital IO signal. Figure 2 is a simplified functional representation for the board and will be referenced many times in the following descriptions of various board functions.

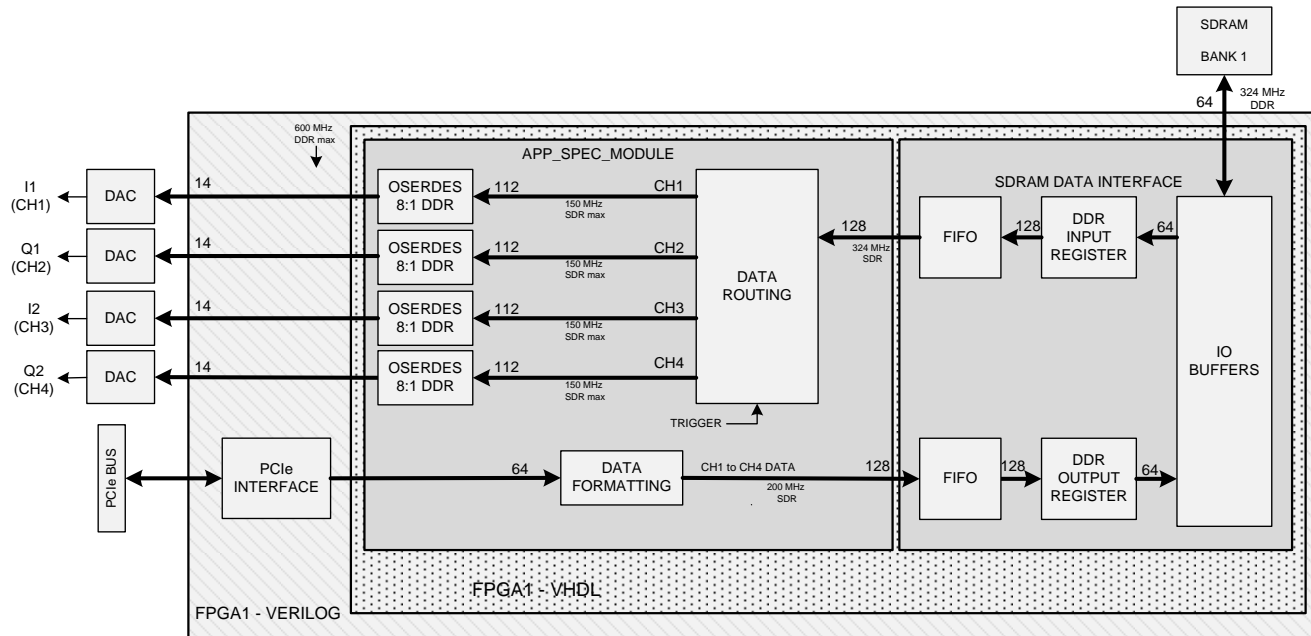


Figure 2: PXDAC4800 Data Flow

2.2 Programmable Hardware Settings

This section defines the various PXDAC4800 programmable hardware settings options. These settings include the various clocks and trigger settings.

2.2.1 Active Output Channels Selection

The Active Channel Mask selection defines which channels will be played back for subsequent playback operations. The PXDAC4800 can support 1, 2, or 4 input channels in various combinations:

Active Channels Selection	Channel Count	Notes
Four channel (Ch. 1, 2, 3, 4)	4	All four output channels are played back
Dual (Ch. 1, 2)	2	Channels 1 and 2
Dual (Ch. 3, 4)	2	Channels 3 and 4
Single (Ch. 1)	1	Channel 1
Single (Ch. 2)	1	Channel 2
Single (Ch. 3)	1	Channel 3
Single (Ch. 4)	1	Channel 4

The Active Channel Mask selection may only be changed while the board is in the Standby operating mode.

The [SetActiveChannelMaskXD48](#) function can be used to select the Active Channels selection.

2.2.2 Playback Clock Selection

The playback clock is the clock that is used to drive the onboard digital to analog converters (DACs). With each rising edge of the playback clock, the DAC will output one sample of playback data.

The PXDAC4800 can be configured to use one of several playback clock sources:

- One of the internal oscillators. The PXDAC4800 has two oscillators that may be used as the playback clock source, a 1200MHz oscillator and a 900MHz oscillator.
- An externally provided clock from the board's external clock input

The figure below shows the functionality of the playback clock circuitry.

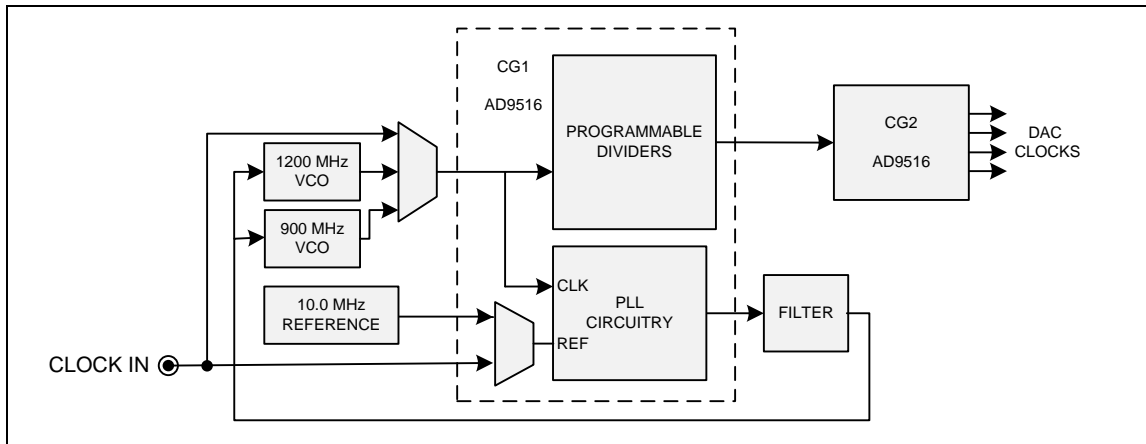


Figure 3: Playback Clock Selection

The Playback Clock Source, as well as any of the clock parameters defined in this section, may only be changed while the board is in the Standby operating mode.

The [SetPlaybackClockSourceXD48](#) function is used to select the ADC clock source.

2.2.2.1 Internal Clocks

If the 1200 MHz or 900 MHz oscillator is chosen, it is phase locked to a 10 MHz reference clock which may be the on-board reference or a reference supplied via the clock input.

10 MHz Clock Reference

The internal clock is locked to a 10 MHz (± 3 PPM max) reference clock and is used in conjunction with the phase lock loop (PLL), which is used to maintain the desired internal clock rate. The PXDAC4800 can be configured to use an externally supplied 10MHz (± 50 PPM max) reference clock. When selected, the external 10MHz clock is provided by the External Clock connector.

The [SetExternalReferenceClockEnableXD48](#) function is used to select the 10MHz reference clock source.

2.2.2.2 External Clock

When the external clock source is selected the board will use the clock provided on the external clock connector as the acquisition clock.

External Clock Specifications	
Signal Type	Sine or Square wave
Coupling	AC
Impedance	50 ohms
Termination	Ground
Frequency	200-1500 MHz
Amplitude	100 mV p-p to 2.0 V p-p

External Clock Rate

When using an external clock, the PXDAC4800 must be informed of the external clock rate so that it can properly synchronize with the clock.

The [SetExternalPlaybackClockRateXD48](#) function is used to specify the external clock rate.

2.2.2.3 Clock Division

When either the external clock or one of the internal oscillators is selected as the playback clock source, two clock dividers are available. Clock divider #1 can be any positive integer value from 1 to 32. Clock divider #2 can be any positive value from 1 to 6. These clock dividers operate in series for a 340 unique dividers spanning 1 (no division) to 1024.

The [SetClockDivider1XD48](#) and [SetClockDivider2XD48](#) library functions are used to set the clock dividers.

2.2.3 Trigger Selection

When a PXDAC4800 is placed into a playback mode, the card will be armed for playback but will not start outputting data until a trigger has been detected. A trigger is an event that is used to synchronize the start of data output to some external event.

The [SetExternalTriggerEnableXD48](#) function is used to enable or disable the trigger source.

External Trigger Source

When the external trigger is selected as the trigger source, a trigger event is defined by a pulse on the external trigger connector. An external trigger pulse is only considered a trigger event when the card is armed for a playback and waiting for a trigger event. Any trigger pulses that may occur while the card is not waiting for a trigger event will be ignored.

External Trigger Specifications	
Signal Type	LVPECL or TTL/CMOS;
Input characteristic	2V comparison with a hysteresis of ~50mV
Impedance	1 kohms to Gnd

External Trigger Direction

The PXDAC4800 can be configured to trigger on the rising edge or falling edge of the external trigger pulse. This is selected with the External Trigger Direction hardware setting.

The [SetExternalTriggerDirXD48](#) function is used to select the edge of the external pulse that will define a trigger event.

2.2.3.1 Software Generated Trigger

Regardless of whether the external trigger is enabled or not, the PXDAC4800 can be explicitly triggered by the software. A software generated trigger can be issued prior to, or while, the card is waiting for a trigger. When the software-generated trigger is issued, the PXDAC4800 will stop waiting for a trigger event and immediately begin playing back data.

The [IssueSoftwareTriggerXD48](#) function is used to issue a software generated trigger to the PXDAC4800.

2.2.3.2 Trigger Mode

The Trigger Mode setting is used to select how trigger events are used to start data playback. The PXDAC4800 implements three trigger modes which are defined in the following subsections.

The Trigger Mode may only be changed while the board is in the Standby operating mode.

The [SetTriggerModeXD48](#) function is used to select the Trigger Mode setting.

Per-trigger Mode

This trigger mode is not valid with Streaming Playback operations.

When this trigger mode is selected, each trigger event will result in the PXDAC4800 playing back a static number of samples (which may be the whole active memory region). The size of this segment is called the Playback Byte count.

Continuous Trigger Mode

When this trigger mode is selected, a single trigger event is used to begin a continuous looping of the playback data. In the case of a RAM playback, when the PXDAC4800 gets to the end of the playback data it will loop around back to the start of the playback data and resume playback with that data.

Single-Shot Trigger Mode

This is a special case of the Per-Trigger Trigger mode. With this trigger mode, a single trigger event is used to begin playback of a static number of samples. The difference is that only the first trigger is registered and all subsequent triggers are ignored. This is a rather special-case trigger mode that will not be needed by most users.

2.2.4 Active Memory Region

In most all cases, the PXDAC4800 software will automatically setup and manage the active memory region for transfer and acquisition operations.

The Active Memory region defines the area of PXDAC4800 RAM that is used for all subsequent playbacks and data transfers. The Active Memory region is defined by two parameters: the Byte Offset defines the offset into PXDAC4800 RAM and the Byte Count defines the total number of samples to acquire or transfer.

The [SetActiveMemoryRegionXD48](#) function is used to setup the Active Memory Region.

2.2.4.1 Byte Offset

The Byte Offset parameter defines the offset into PXDAC4800 RAM at which all subsequent playbacks or data transfers will begin. For transfer operations, the Byte Offset defines the 0-based byte index at which the transfer will begin. For playback operations, the Byte Offset defines the 0-based byte index at which RAM playback data will begin.

The Byte Offset must be an integer multiple of 8192 samples. The first sample in PXDAC4800 RAM is at byte index 0. See below for maximum allowed Byte Offset.

2.2.4.2 Byte Count

The Byte Count parameter defines the total number of bytes to consider for all subsequent transfers and playbacks. For transfer operations, the Byte Count defines the total number of bytes to transfer. For playback operations, the Byte Count defines the total number of bytes to playback. In either case, the Byte Count is independent of channel count. For a given Byte Count, the number of bytes played back/transferred per-channel is the Byte Count divided by the number of active channels

The Byte Count must be an integer multiple of 8192 samples. See below for maximum allowed Byte Count.

2.2.4.3 Configuring the Active Memory Region

Internally, the PXDAC4800 RAM has one logical bank of 1073741824 bytes (1GiB) in size. Playback data used for all channels come from the same RAM.

Because of this, addressing the acquisition data in memory depends on the current [Active Channel Mask](#) setting. PXDAC4800 software will automatically take the Active Channel Mask selection into consideration when configuring the PXDAC4800 hardware such that from the point of view of the user, the PXDAC4800 memory is a single contiguous block of memory, regardless of what the current Active Channels Mask selection is.

Maximum Byte Count

The maximum valid sample count depends on the current Active Channels setting:

Active Channels Selection	Maximum Byte Count
Four channel (Ch. 1, 2, 3, 4)	1073741824 (1GB) total samples, or 268435456 (256MB) samples per channel
Dual (Ch. 1, 2)	1073741824 (1GB) total samples, or 536870912 (512MB) samples per channel
Dual (Ch. 3, 4)	1073741824 (1GB) total samples, or 536870912 (512MB) samples per channel
Single (Ch. X)	1073741824 (1GB) total samples

The maximum allowed Byte Offset value is the equal to the maximum Byte Count for the currently selected Active Channels selection minus 8192.

Byte Offset Selection for Multichannel Data

For multichannel data, channel data is stored in a sample interleaved format:

Active Channels Selection	Sample Layout in Memory
Four channel (Ch. 1, 2, 3, 4)	Ch 1, Ch 2, Ch 3, Ch 4, Ch 1, Ch 2, ...
Dual (Ch. 1, 2)	Ch 1, Ch 2, Ch 1, Ch 2, ...
Dual (Ch. 3, 4)	Ch 3, Ch 4, Ch 3, Ch 4, ...
Single (Ch. X)	Ch X, Ch X, ...

When navigating multichannel data, the above must be taken into consideration when attempting to address a particular sample of a particular channel. To jump to sample N of channel C , the Starting Sample would be:

$$(N * C_{\text{total}}) + C$$

Where:

- N is the 0-based sample index
- C_{total} is the total number of channels of data
- C is the 0-based channel index, relative to the Active Channel selection.

Due to alignment restrictions, the starting sample will always be aligned to the first channel of the active channels selection.

2.2.5 Operating Mode

The state of a PXDAC4800 device is primarily driven by the current operating mode. The PXDAC4800 has several operating modes, which can be categorized into one of four types: idle, playback, streaming and transfer. Further, an ‘active’ operating mode refers to any of the playback or transfer modes.

PXDAC4800 software wraps each of the active operating modes with high-level functions that manage the Operating Mode as well as the Active Memory Region, which is closely related to the operating mode. That being said, the function [SetOperatingModeXD48](#) is used to change the board’s operating mode.

Each operating mode is detailed in the following subsections.

2.2.5.1 Standby Mode

Standby mode is the primary idle operating mode. This is the operating mode that should be used to configure the PXDAC4800 hardware settings prior to a data acquisition. This includes things like setting up the playback clock and trigger parameters.

Operating mode transitions should always be coming from or going to Standby mode. Direct transitions between any two active operating modes are not allowed.

Putting the board into Standby mode while a transfer or playback is armed or in progress will result in the transfer/playback being cancelled and all threads waiting for the respective operation to complete will be resumed with “cancelled” status.

2.2.5.2 Load RAM Mode

This is the operating mode used for transferring data from host PC to the PXDAC4800 RAM. This operating mode will transfer data into PXDAC4800 RAM only; playback from RAM occurs while the board is in RAM Playback mode.

Users should never directly enter this operating mode. The PXDAC4800 software implements data transfer routines that should be used.

2.2.5.3 Streaming Playback Mode

This is the operating mode used for streaming playback operations. This is a playback mode in which playback data is streamed from the host PC to the PXDAC4800 via DMA transfer. In this playback mode, the PXDAC4800 onboard memory is used as a large FIFO to buffer the data. For this mode of playback, the host PC must be able

to transfer playback data to the card at or above the playback data rate or the PXDAC4800 will run out of data (FIFO underflow) resulting in gaps in the playback data.

The [BeginStreamingPlaybackXD48](#) function is used to start a streaming playback operation.

2.2.5.4 RAM Playback Mode

This is the operating mode used for RAM playback operations. In this playback mode, the PXDAC4800 will playback a section of playback data located in PXDAC4800 RAM, previously loaded by using the [Load RAM mode](#).

The [BeginRamPlaybackXD48](#) function is used to start a RAM playback.

2.3 PXDAC4800 Playback Data Format

This section defines the format of the sample data played back by the PXDAC4800.

2.3.1 Sample Format

The PXDAC4800 is capable of playing back 8- or 14-bit samples (aligned to 16 bits). Further, samples may be either unsigned $[0 \dots N]$ or signed $[-N, +N]$.

The following table shows the minimum, midscale, and maximum DAC sample values for the various sample sizes and formats.

	Unsigned			Signed		
	Min.	Midscale	Max.	Min.	Midscale	Max.
14-bit (MSB pad)	0	8192 (0x2000)	16383 (0x3FFF)	-8192 (0x2000)	0	8191 (0x1FFF)
14-bit (LSB pad)	0	32768 (0x8000)	65532 (0xFFFC)	-32768 (0x8000)	0	32764 (0x7FFC)
8-bit	0	128 (0x80)	255 (0xFF)	-128 (0x80)	0	127 (0x7F)

For unsigned 8-bit data, a sample value of 0 is equivalent to the minimum output voltage range. A sample value of 255 is equivalent to the maximum output voltage range. A sample value of 128 is approximately equivalent to 0V.

For unsigned 14-bit data, a sample value of 0 is equivalent to the minimum output voltage range. A sample value of 65532 is equivalent to the maximum output voltage range. A sample value of 32768 is approximately equivalent to 0V.

For 14-bit data, the value of the two extra bits are ignored by the PXDAC4800 and can have any value.

2.3.2 Multichannel Data Format

When playing back multichannel data, the PXDAC4800 stores data in a sample interleaved format. This means that each channel is alternated in the input data:

Dual channel format: Ch. 1 Sample 1, Ch. 2 Sample 1, Ch. 1 Sample 2, Ch. 2 Sample 2, ...

Quad channel format: Ch. 1 Sample 1, Ch. 2 Sample 1, Ch. 3 Sample 1, Ch. 4 Sample 1, Ch. 1 Sample 2, Ch. 2 Sample 2, ...

The PXDAC4800 library implements routines to interleave ([InterleaveDataXbitYChanXD48](#)) or de-interleave ([DeInterleaveDataXbitYChanXD48](#)) multi-channel data. Be advised that de-interleaving channel data ‘on-the-fly’ while playing back will place additional strain on the CPU(s) and might not be sustainable for high data rate playback operations. For these circumstances, interleaving should be performed as a preprocess operation prior to the playback.

2.3.3 PXDAC4800 Playback Data Files (*.rd16, *.rd8)

When loading or streaming playback data from a file, the PXDAC4800 software assumes that the file is raw playback data with no additional context information. This is the native format used by Signatec data acquisition devices. This simple file format allows the software to read and transfer the data with no intermediate processing to slow things down since the file format is the same format understood by the PXDAC4800.

Signatec data acquisition software, such as one of the various product “Scope” applications, as well as certain data acquisition library routines have the ability to save acquisition data to a file. The native file format used by these entities is the RD16 file format for 16-bit data or RD8 file format for 8-bit data. The RD16 moniker is derived from “Raw Data 16-bit”. RD16 files are identified by the ‘.rd16’ file extension. The RD8 moniker is derived from “Raw Data 8-bit”. RD8 files are identified by the ‘.rd8’ file extension.

RD16/RD8 files are binary files that contain only sample data. There is no file header or additional information in the file. The first byte of the file is the first data sample. This simple file format has two big advantages. First, it’s very fast to read these files since data is read from the file exactly as it is sent to the PXDAC4800 hardware. If the underlying file system (e.g. a high-speed RAID system) can keep up with the data rate, data can be streamed from file to the PXDAC4800 card at the highest playback rate supported. The second advantage is that this file format is very generic which makes it easy for other software to generate or utilize the data. This includes custom software, or other software environments such as Matlab.

2.3.4 Signatec Recorded Data Context Files (*.srdc)

The main disadvantage of the RD16/RD8 file format is that, since no context information is stored in the file, it may not be apparent what kind of data is in the file. Is it single channel or multichannel? What was the input voltage range? The sampling rate? To get around this problem, Signatec software can also be configured to generate a small auxiliary context file (XML format) that sits in the same directory as the RD16/RD8 file. This auxiliary file can contain information such as the channel count, input voltage range, sampling rate, source board, operator notes, or even user-defined data.

By convention, the name of the auxiliary data file is the full pathname of the RD16/RD8 data file, appended with a ‘.srdc’ extension. For example, if acquisition data was being written to the file “C:\Data\Recordings\MyData.rd8” then the auxiliary data would be written to “C:\Data\Recordings\MyData.rd8.srdc”. The PXDAC4800 library has routines that can be used to read and write these files.

SRDC files are saved in XML format so they can easily be read in by any XML-aware software.

See the [Signatec Recorded Data Context \(SRDC\) Information](#) section for more details on these files.

2.4 Maximum DAC clock frequency

This section defines the maximum frequency of the DACs according of the selection modes because of a limitation of the data transfer rate. The general rule is a maximum bandwidth of 38.4 Gb/s

2.4.1 Clock frequency with 8-bit Data Format

With the 8-bits Data Format, the maximum DAC clock frequency is 1.2 GHz for all Active Channel Mask selection.

2.4.2 Clock frequency with 14-bit Data Format

With 14-bit Data Format (LSB and MSB pad), the maximum DAC clock frequency depends of the Active Channel Mask selection:

Active Channels Selection	Channel Count	Maximum DAC clock frequency
Four channel (Ch. 1, 2, 3, 4)	4	600 MHz
Dual (Ch. 1, 2 or Ch. 3, 4)	2	1.2 GHz
Single (Ch. 1, 2, 3 or 4)	1	1.2 GHz

3 PXDAC4800 Software Development Reference

3.1 Contents

The PXDAC4800 is shipped with a disk titled “PXDAC4800 Product Software” that contains a Windows Installer setup used to install PXDAC4800 drivers, libraries, documentation, components, applications, utilities, and programming examples. Some of the more important items are:

PXDAC4800 dynamic link library: This library is the interface to the PXDAC4800 driver. User applications will use the functions exported by this library to connect their software to the PXDAC4800 device. The library (32-bit: PXDAC4800.dll, 64-bit: PXDAC4800_64.dll) is installed to the product installation folder and the system library search path is updated to include this directory.

PXDAC4800 Operator’s Manual: This is the document that you are currently reading and is installed to the Documentation\ folder of the install folder.

The PXDAC4800 Playback Application: This is a full-featured application that demonstrates how to use the various features of the PXDAC4800 as well as perform PXDAC4800 data playbacks. This application is also a great starting point to getting familiar with the PXDAC4800. The source code for this application is installed to the Source\Examples\PXDAC4800 Playback Application\ folder of the install folder.

Various projects demonstrating how to interface with the PXDAC4800 located in the Source\ folder of the installation folder.

3.2 Installing Software

To install the PXDAC4800 software and documentation just insert the Signatec Product Software CD into your CD-ROM drive. Installation should automatically begin. If installation does not automatically begin, you can manually start installation by running “setup.exe” in the root directory of the CD. This installation will install all PXDAC4800 software and documentation to the folder selected during installation.

Note: It is important to run “setup.exe” and not run the installer file (.msi) directly. Running setup.exe will ensure that required runtime libraries are installed before installing the product software. If setup.exe is not executed, the product software will install but may have problems when executed.

3.3 Developing PXDAC4800 Software

PC applications interface the PXDAC4800 through the PXDAC4800 library. This library exports a variety of functions that are used to control the PXDAC4800. All user-mode code accesses PXDAC4800 devices through this library. This library, in turn, is the only entity that interacts with the underlying PXDAC4800 kernel-mode device driver. The PXDAC4800 device driver is the only software entity that directly interacts with the PXDAC4800 hardware. The library functions are documented in the [C Library Functions](#) section. The following sections describe how to set up the library and program the PXDAC4800.

3.3.1 Setting up the Build Environment – Windows Platform

The PXDAC4800 library is composed of three parts: the header file (pxdac4800.h), the import library (32-bit: PXDAC4800.lib, 64-bit: PXDAC4800_64.lib), and the dynamic link library (32-bit: PXDAC4800.dll, 64-bit: PXDAC4800_64.dll). The header and import library are required to build PXDAC4800 applications and the dynamic link library is required to run PXDAC4800 applications.

The header file, pxdac4800.h, is the C interface to the library. Your PXDAC4800 applications will “#include” this file to define PXDAC4800 data structures, function prototypes, constants, and macros. This file is installed to the include\ folder of the PXDAC4800 install directory. It is recommended that you add this path to your compiler’s header file search path¹. Adding the header file to the include file search path will allow you to keep a single copy of pxdac4800.h on your system and access it like a standard C header file. (That is, you can do an “#include <pxdac4800.h>” instead of an “#include “Path-to-pxdac4800.h”.”)

The import library, PXDAC4800.lib or PXDAC4800_64.lib, is the stub-library that the linker needs to resolve linkage issues when building your application. This stub-library is installed to the lib\ folder of the PXDAC4800 install directory. Like the library header file, it is recommended that you put this library in your build environment’s library file search path. This file needs to be included by the build process or you will get linker errors when you build your application. If you are using a Microsoft Visual C/C++ compiler, the stub-library will automatically be added to the build process when you include the library header so you do not have to manually add this file to your project. (Automatic linking with the stub library can be overridden by “#defining” XD48PP_NO_LINK_LIBRARY before including pxdac4800.h.)

The dynamic link library, PXDAC4800.dll or PXDAC4800_64.dll, is installed to the PXDAC4800 software installation folder (C:\Program Files\Signatec\PXDAC4800 by default). During software installation, the system library search path is updated to include this folder so the PXDAC4800 library can be located when needed.

3.3.2 Library Functions That Use Character Strings

For library functions that accept or generate character strings (e.g. GetErrorTextXD48), the PC platform library may actually implement two versions: one version that works with ANSI character strings (char*) and another

¹ Microsoft Visual C++ 6 users can do this by selecting *Options* from the *Tools* menu, and selecting the *Directories* tab. Microsoft Visual Studio .NET users can do this by selecting *Options* from the *Tools* menu, and selecting *VC++ Directories* under the *Projects* folder.

version that works with UNICODE strings (wchar_t*). The ANSI character version function names are suffixed with an 'AXD48' and the UNICODE version function names are suffixed with 'WXD48'.

Macros have been defined so that library users can write generic code that will work with either version. Consider GetErrorTextXD48 as an example. The library implements two versions of this function: _GetErrorTextAXD48 and _GetErrorTextWXD48. If the _UNICODE symbol is defined, then the GetErrorTextXD48 macro will expand to _GetErrorTextWXD48, else it will default to _GetErrorTextAXD48. In this manual library functions are documented using the character-size agnostic type TCHAR.

```
int _GetErrorTextAXD48 (int res, char** bufpp, int flags);
int _GetErrorTextWXD48 (int res, wchar_t** bufpp, int flags);

#ifdef _UNICODE
# define GetErrorTextXD48 _GetErrorTextWXD48
#else
# define GetErrorTextXD48 _GetErrorTextAXD48
#endif

// Effective library function prototype:
int GetErrorTextXD48 (int res, TCHAR** bufpp, int flags);
```

3.3.3 Active Channel Masks (How Playback Channels are Selected)

The library uses a generic bitmask to indicate which of the output channels are selected for output. The bitmask is just a simple integer value in which the first (least significant) 4 bits are used to indicate the state of the 4 output channels. A set bit means the respective channel is selected.

- Bit 0 (0x1) = Channel 1
- Bit 1 (0x2) = Channel 2
- Bit 2 (0x4) = Channel 3
- Bit 3 (0x8) = Channel 4

Note that not all 16 channel combinations are supported by the PXDAC4800 hardware. The [IsActiveChannelMaskSupportedXD48](#) function may be used to programmatically determine if a particular active channel mask is supported. The library defines symbolic constants for the currently defined active channel combinations:

PXDAC4800 Library Constant	Interpretation
XD48CHANMASK_4_CHANNEL (0xF)	Four channel: channels 1, 2, 3 and 4
XD48CHANMASK_2_CHANNEL_1_2 (0x3)	Dual channel: channels 1 and 2
XD48CHANMASK_2_CHANNEL_3_4 (0xC)	Dual channel: channels 3 and 4
XD48CHANMASK_1_CHANNEL_1 (0x1)	Single channel: channel 1
XD48CHANMASK_1_CHANNEL_2 (0x2)	Single channel: channel 2
XD48CHANMASK_1_CHANNEL_3 (0x4)	Single channel: channel 3
XD48CHANMASK_1_CHANNEL_4 (0x8)	Single channel: channel 4

Library functions that work with active channel masks:

- [SetActiveChannelMaskXD48](#)
- [IsActiveChannelMaskSupportedXD48](#)
- [GetMaxByteCountForActiveChanMaskXD48](#)

3.4 PXDAC4800 Library Functions

This section documents the various functions implemented by the PXDAC4800 library.

3.4.1 Library Utility Functions

The functions in this section are generic library utility functions.

3.4.1.1 CalculateCycleCountsXD48

Prototype

```
int CalculateCycleCountsXD48 (
    HXD48          hBrd,
    double         dPtsPerCycle,
    unsigned int*  pSampleCount,
    XD48S\_CYCLE\_CALC\_CTX\* ctxp = NULL);
```

Description

Calculate number of periodic waveform cycles needed to fit in RAM, aligned for playback

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *dPtsPerCycle*

The desired points-per-cycle of the data to be played back. See remarks.

[out] *pSampleCount*

On success, the function will write the integral, aligned sample count to the variable pointed to by this parameter.

[in/out] *ctxp*

An optional pointer to a XD48S_CYCLE_CALC_CTX structure that defines how the function operates. The user must provide a XD48S_CYCLE_CALC_CTX struct to find the closest match if dPtsPerCycle does not fit in a sufficient number of samples.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error; see remarks.

Remarks

This function is used to obtain the total number of samples that should be generated for a periodic signal with the given frequency such that the generated signal will fit in an aligned number of samples that can be used for a continuous playback without breaks in the data.

The calculated number of samples is dependent on points-per-cycle (PPC) as well as the PXDAC4800 DAC data rate. The PXDAC4800 DAC data rate is the playback rate divided by the interpolation factor and can be obtained

via the `GetPlaybackDataRateXD48` function. Points-per-cycle defines the number of samples that define one cycle of the periodic data and is the DAC data rate divided by the frequency of the output data.

This function works by first finding a number of cycles that result in an integral number of samples. Next, the function finds a number of these cycle-groups that result in an aligned number of samples.

For example, suppose we are given a points-per-cycle count of 2.25. 4 of these cycles would result in a whole number of samples:

$$4 \text{ cycles} * 2.25 \text{ points-per-cycle} = 9 \text{ points}$$

8 instances of those 9-point cycle-groups would result in 72 total samples which, is aligned to 8 samples.

For some frequency/PPC parameters, there is no integral number of samples that fit in the desired number of samples. In other words, a whole, aligned number of samples would not fit in RAM. For these situations this function can also be instructed to find the closest match. In this case, the function will adjust the points-per-cycle count to find one that will fit.

Points-per-cycle is playback data rate divided by output waveform rate. Playback data rate is playback clock rate divided by interpolation factor.

3.4.1.2 DumpLibErrorXD48

Prototype

```
int DumpLibErrorXD48 (int res, const TCHAR* pPreamble, HXD48 hBrd, FILE* filp);
```

Description

Dump library error description to given file or stream

Parameters

[in] *res*

The library error value for which to obtain information. This can be any (generic) `SIG_*`, or (PXDAC4800-specific) `SIG_XD48_*` error value.

[in] *pPreamble*

A pointer to a NULL-terminated string that will be displayed before the generated error text. This parameter can be NULL.

[in] *hBrd*

A handle to the PXDAC4800 device for which the error occurred. For some types of errors, the PXDAC4800 handle used when the error was generated may contain additional context information. This parameter may be `XD48_INVALID_HANDLE`.

[in] *filp*

The file or stream that will receive the error text. If this parameter is NULL then `stderr` (standard error output) will be used.

Return Value

Returns `SIG_SUCCESS` (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This is a utility function used to dump information on a specific library error.

DumpLibErrorXD48 is a macro that will expand to _DumpLibErrorAXD48 or _DumpLibErrorWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Example Usage

```
res = LoadFileIntoRamXD48(hBrd, 0, 0, _T("C:\\Data\\PlaybackData.rd16"), 0, 0, 0);
if (res != SIG_SUCCESS)
    DumpLibErrorXD48(res, _T("Failed to load file into RAM: "), hBrd);
```

Related Functions

[GetErrorTextXD48](#)

3.4.1.3 GetErrorMessXD48

Prototype

```
Char* GetErrorMessXD48 (int res, char* bufp, int flags, HXD48 hBrd);
```

Description

Obtain a user-friendly string describing the given SIG_* error value. This function is particularly useful when working in the Matlab environment. See Matlab SDK for programming examples.

Parameters

[in] *res*

The library error value for which to obtain information. This can be any (generic) SIG_*, or (PXDAC4800-specific) SIG_XD48_* error value.

[out] *bufp*

A pointer to a char buffer that will receive the error message associated to the error value.

[in] *flags*

A set of flags that dictate function behavior.

Flag	Interpretation
XD48ETF_IGNORE_SYSERROR (0x00000001)	Disregard system error information (Windows: GetLastError, Linux: errno)
XD48ETF_NO_SYSERROR_TEXT (0x00000002)	Do not generate system error code text
XD48ETF_FORCE_SYSERROR (0x00000004)	Display system error information even if it may not be relevant

[in] *hBrd*

A handle to the PXDAC4800 device for which the error occurred. For some types of errors, the PXDAC4800 handle used when the error was generated may contain additional context information. This parameter may be XD48_INVALID_HANDLE.

Return Value

Returns the pointer to the char buffer specified by the `bufp` parameter. This pointer is to be used in the Matlab environment.

Remarks

This function is used to get a user-friendly string describing the given library error value. For some error values, information on the current (thread-specific) system error state may also be provided if a valid device handle is passed.

`GetErrorTextXD48` is a macro that will expand to `_GetErrorTextAXD48` or `_GetErrorTextWXD48` depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Example Usage

```
res = LoadFileIntoRamXD48(hBrd, 0, 0, _T("C:\\Data\\PlaybackData.rd16"), 0, 0, 0);
if (res != SIG_SUCCESS)
{
    TCHAR* bufp;

    GetErrorTextXD48(res, &bufp, 0, hBrd);
    _tprintf (_T("Failed to load file into RAM: %s\n"), bufp);
    // Caller frees string memory when done with it
    FreeMemoryXD48(bufp);
}
```

Related Functions

[DumpLibErrorXD48](#)

3.4.1.4 GetErrorTextXD48

Prototype

<code>int GetErrorTextXD48 (int res, TCHAR** bufpp, int flags, HXD48 hBrd = XD48_INVALID_HANDLE);</code>
--

Description

Obtain a user-friendly string describing the given `SIG_*` error value

Parameters

[in] *res*

The library error value for which to obtain information. This can be any (generic) `SIG_*`, or (PXDAC4800-specific) `SIG_XD48_*` error value.

[out] *bufpp*

A pointer to a `TCHAR` pointer that will receive the address of the library allocated string containing the error text string. This buffer must be freed by caller by calling the [FreeMemoryXD48](#) library function

[in] *flags*

A set of flags that dictate function behavior.

Flag	Interpretation
XD48ETF_IGNORE_SYSERROR (0x00000001)	Disregard system error information (Windows: GetLastError, Linux: errno)
XD48ETF_NO_SYSERROR_TEXT (0x00000002)	Do not generate system error code text
XD48ETF_FORCE_SYSERROR (0x00000004)	Display system error information even if it may not be relevant

[in] *hBrd*

A handle to the PXDAC4800 device for which the error occurred. For some types of errors, the PXDAC4800 handle used when the error was generated may contain additional context information. This parameter may be XD48_INVALID_HANDLE.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to get a user-friendly string describing the given library error value. For some error values, information on the current (thread-specific) system error state may also be provided if a valid device handle is passed.

GetErrorTextXD48 is a macro that will expand to _GetErrorTextAXD48 or _GetErrorTextWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Example Usage

```
res = LoadFileIntoRamXD48(hBrd, 0, 0, _T("C:\\Data\\PlaybackData.rd16"), 0, 0, 0);
if (res != SIG_SUCCESS)
{
    TCHAR* bufp;

    GetErrorTextXD48(res, &bufp, 0, hBrd);
    _tprintf (_T("Failed to load file into RAM: %s\n"), bufp);
    // Caller frees string memory when done with it
    FreeMemoryXD48(bufp);
}
```

Related Functions

[DumpLibErrorXD48](#)

3.4.1.5 GetFPGAStatusXD48

Prototype

```
int GetFPGAStatus (HXD48 hBrd, unsigned int* status_val);
```

Description

This function will return the status buffer of the FPGA in **status_val**, you can obtain different information by applying different mask.

Mask	Interpretation
XD48DAC_PLL_LOCKED_STATUS (0x00000010)	If 0 = unlock , 1 = lock
XD48UNDERFLOW_STATUS (0x00000020)	If 0 = No data underflow , 1 = data underflow
XD48PLAYBACK_IN_PROGRESS_STATUS (0x00000080)	If 0 = No playback in progress , 1 = playback in progress

Parameters

[in] *hBrd*

A handle to the PXDAC4800 device for which the error occurred. For some types of errors, the PXDAC4800 handle used when the error was generated may contain additional context information. This parameter may be XD48_INVALID_HANDLE.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

3.4.1.6 GetMaxByteCountForActiveChanMaskXD48

Prototype

```
int GetMaxByteCountForActiveChanMaskXD48 (HXD48 hBrd, int chan_mask, unsigned int* pmax_bytes);
```

Description

Obtain maximum transfer/playback byte count for given active channel mask

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *chan_mask*

The [active channel mask](#) to consider.

[out] *pmax_bytes*

A set of flags that dictate function behavior.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to obtain the maximum number of bytes that can be used for a RAM playback or transfer. Because the PXDAC4800 memory is split between to discrete RAM banks, the maximum byte count is a function of the active channel mask setting.

See [Active Channel Masks](#) section for more details on channel masks.

Example Usage:

```
unsigned int playback_bytes;
int chan_mask = GetActiveChannelMaskXD48(hBrd);
// Determine largest possible RAM playback length for current active channel selection
GetMaxByteCountForActiveChanMaskXD48(hBrd, chan_mask, &playback_bytes);
{
    // ...load playback_bytes bytes of playback data to card...
}
// Begin the playback
BeginRamPlaybackXD48(hBrd, 0, playback_bytes);
```

Related Functions

[_SetActiveMemoryRegionXD48](#)

3.4.1.7 GetOutputVoltageRangeVoltsXD48

Prototype

```
int GetOutputVoltageRangeVoltsXD48 (
    int        val,
    double*    pPeakToPeakVolts,
    HXD48      hBrd          = XD48_INVALID_HANDLE);
```

Description

Obtain peak-to-peak voltage for given output voltage encoding [0, 1023]

Parameters

[in] *val*

The output voltage range encoding to consider. This can be any value in the range [0, 1023].

[out] *pPeakToPeakVolts*

A pointer to a double that will receive the peak-to-peak output voltage.

[in] *hBrd*

Handle to the device for which to consider or XD48_INVALID_HANDLE. If XD48_INVALID_HANDLE is passed then the output voltage range will be based on standard translation. If custom output voltage ranges are ever implemented, this parameter will allow for the software to obtain the custom information from the device.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to programmatically obtain the peak-to-peak voltage, in volts, that corresponds to the output voltage range selection passed to the [SetOutputVoltageChNXD48](#) functions.

Example Usage

```
double dVoltsPP;
int ch1_vr_sel = GetOutputVoltageCh1XD48(hBrd);
GetOutputVoltageRangeVoltsXD48 (ch1_vr_sel, &dVoltsPP, hBrd);
_tprintf (_T("Channel 1 output peak-to-peak voltage is %f Vp-p\n"), dVoltsPP);
```

Related Functions

[SetOutputVoltageCh1XD48](#)

3.4.1.8 SampleSizeFromTypeXD48

Prototype

<code>int SampleSizeFromTypeXD48 (int val);</code>
--

Description

Obtain DAC sample size in bytes for given sample type setting (XD48SAMPSIZE_*)

Parameters

[in] *val*

The sample size selection (XD48SAMPSIZE_*) for which to obtain the sample size.

Return Value

For 8-bit samples (XD48SAMPSIZE_8BIT), this function will return 1.

For 14-bit samples (XD48SAMPSIZE_14BIT_MSBPAD or XD48SAMPSIZE_14BIT_LSBPAD), this function will return 2.

Returns one of the [library error codes](#) (which are all negative) on error.

Remarks

This is a utility function that can be used to programmatically obtain the size of a DAC sample, in bytes, for the DAC Sample Size selection value used by the [SetDacSampleSizeXD48](#) function.

Related Functions

[SetDacSampleSizeXD48](#)

3.4.1.9 SetUserDataXD48

Prototype

<code>int SetUserDataXD48 (HXD48 hBrd, void* data);</code>
<code>int GetUserDataXD48 (HXD48 hBrd, void** datap);</code>

Description

Set or get user-defined data value associated with PXDAC4800 handle

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *data*

User-defined data value. Interpretation of this value is entirely user-defined and has no effect on the underlying library code.

[out] *datap*

A pointer to a void* variable that will receive the user-defined data value associated with the board handle.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Each PXDAC4800 device connection contains a user-defined pointer-precision data value set aside for application usage. The interpretation of this data value is application defined and is ignored by the library. The default value for the user-defined data value is NULL.

3.4.1.10 ValidateConfigEepromXD48

Prototype

```
int ValidateConfigEepromXD48 (HXD48 hBrd);
```

Description

Run an integrity check on configuration EEPROM data

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function will run an integrity check on the board's configuration EEPROM which is used to store read-only hardware configuration data such as serial number, calibration data, etc. If this integrity check fails then either the configuration has been corrupted or there is a hardware problem.

If this validation fails, the error text obtained by calling [GetErrorTextXD48](#) will contain additional details about what caused the validation to fail.

3.4.1.11 IsDcXD48

Prototype

```
int IsDcXD48 (HXD48 hBrd);
```

Description

Validation if the board is a DC version of the PXDAC4800.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Library Symbolic Constant	Value	Interpretation
XD48_AC	0	AC board
XD48_DC	1	DC board

Returns **XD48_DC** or **XD48_AC** on success or one of the [library error codes](#) (which are all negative) on error.

3.4.2 PXDAC4800 Device Enumeration and Connection Management

The functions in this section involve managing PXDAC4800 device connections and general PXDAC4800 device handle management.

3.4.2.1 ConnectToDeviceXD48

Prototype

```
int ConnectToDeviceXD48 (HXD48* phDev, unsigned int brdNum);
```

Description

This function is used to establish a connection to a PXDAC4800 data acquisition device, represented by a PXDAC4800 device handle of type HXD48.

Parameters

[out] *phDev*

A pointer to a HXD48 variable that will receive the PXDAC4800 device handle. This PXDAC4800 device handle is only valid in the current process. This handle should be treated as an opaque object; interpretation of the handle value is PXDAC4800 library specific.

[in] *brdNum*

This parameter is used to select which PXDAC4800 in the system to connect to. If this value is in the range [1, 32] then the number is assumed to be the ordinal number of the device in the system. A board's ordinal represents the system-specific enumeration of the device. This may or may not follow the order of the physical PCIe slots. If this value is outside the aforementioned range then it is assumed to be the PXDAC4800 serial number of the device in which to connect.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function does not interact with the underlying PXDAC4800 hardware device in any way. The device driver's hardware register cache is consulted for software register values and the operating mode is left unchanged. This allows a thread to attach to a currently running PXDAC4800 device in a non-intrusive manner.

Like other handle-based mechanisms, the actual value of a PXDAC4800 device handle should be treated as an opaque value. The exception to this is a special value, XD48_INVALID_HANDLE, which is used to identify an invalid PXDAC4800 device handle.

An application should close the connection to the PXDAC4800 device by calling the DisconnectFromDeviceXD48 library function. Failure to disconnect from the device can result in memory leaks in the calling process.

Example Usage

```
HXD48 hBrd;

// Connect to first device in system
ConnectToDeviceXD48(&hBrd, 1);
// ...
DisconnectFromDeviceXD48(hBrd);

// Connect to device with serial number 123456
ConnectToDeviceXD48(&hBrd, 123456);
// ...
DisconnectFromDeviceXD48(hBrd);
```

Related Functions

[DisconnectFromDeviceXD48](#), [ConnectToVirtualDeviceXD48](#)

3.4.2.2 ConnectToVirtualDeviceXD48

Prototype

```
int ConnectToVirtualDeviceXD48 (HXD48* phBrd, unsigned int serialNum, unsigned int brdNum);
```

Description

Establish a connection to a virtual (fake) PXDAC4800 device

Parameters

[out] *phBrd*

A pointer to a HXD48 variable that will receive the virtual PXDAC4800 device handle.

[in] *serialNum*

The serial number to use for the virtual device. This can be any number. The PXDAC4800 library ignores this value. This will be the number obtained via the GetSerialNumberXD48 function.

[in] *brdNum*

The board number to use for the virtual device. This can be any number; the PXDAC4800 library ignores this value. This will be the number obtained by the GetOrdinalNumberXD48 function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to establish a connection to a virtual PXDAC4800 device. A virtual device is one that is not connected to any real PXDAC4800 hardware. Virtual devices are mainly used for software development and debugging.

Related Functions

[DisconnectFromDeviceXD48](#), [ConnectToDeviceXD48](#)

3.4.2.3 DuplicateHandleXD48

Prototype

```
int DuplicateHandleXD48 (HXD48 hBrd, HXD48* phNew);
```

Description

Duplicate a PXDAC4800 device handle

Parameters

[in] *hBrd*

The PXDAC4800 device handle to duplicate. This handle must have been obtained from the current process.

[out] *phNew*

A pointer to a HXD48 variable that will receive the new, duplicated handle.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to duplicate an existing PXDAC4800 device handle. A duplicated handle is connected to the same device and has the same locally cached hardware settings and context information. The “duplicate” handle is

a normal device handle and does not require any special considerations. DisconnectFromDeviceXD48 should be called to release the handle just like normal.

Related Functions

[ConnectToDeviceXD48](#)

3.4.2.4 DisconnectFromDeviceXD48

Prototype

```
int DisconnectFromDeviceXD48 (HXD48 hBrd);
```

Description

This function is used to release the handle for the PXDAC4800 when it is no longer needed in the program. The function also frees any utility DMA buffers allocated by the library for the given PXDAC4800 device.

Parameters

[in] *hBrd*

The PXDAC4800 device handle to close. This handle ceases to be valid once this function returns successfully.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

In general, closing a PXDAC4800 device handle will have no effect on the underlying hardware. This allows one to connect/disconnect to a PXDAC4800 device in an unobtrusive manner. A slight exception to this is that the PXDAC4800 driver may interact with the PXDAC4800 hardware when all connections to a particular PXDAC4800 (over all processes in the system) have been closed.

This function does not alter the current operating mode. You will typically want to put the board into Standby or Off mode before your application exits. This function does not automatically do this because it is possible that the device may still be in use by another thread or process.

This function should also be used to disconnect from a virtual device.

Related Functions

[ConnectToDeviceXD48](#)

3.4.2.5 GetDeviceCountXD48

Prototype

```
int GetDeviceCountXD48();
```

Description

Obtain a count of all PXDAC4800 devices present in the local system

Return Value

Returns the number of physical PXDAC4800 devices present in the local system. If this function returns zero then either no PXDAC4800 devices are present or the PXDAC4800 driver has not been installed.

3.4.2.6 IsDeviceRemoteXD48

Prototype

```
int IsDeviceRemoteXD48 (HXD48 hBrd);
```

Description

Determines if a given PXDAC4800 device handle is for a remote device

Parameters

[in] *hBrd*

The PXDAC4800 device handle to check. The `ConnectToRemoteDeviceXD48` function is used to connect to a remote PXDAC4800 device.

Return Value

Returns a non-zero value if the given handle is connected to a remote PXDAC4800 or zero if the handle is not connected to a remote PXDAC4800.

Related Functions

[IsDeviceVirtualXD48](#), [IsHandleValidXD48](#)

3.4.2.7 IsDeviceVirtualXD48

Prototype

```
int IsDeviceVirtualXD48 (HXD48 hBrd);
```

Description

Determines if a given PXDAC4800 device handle is for a virtual device

Parameters

[in] *hBrd*

The PXDAC4800 device handle to check. The `ConnectToVirtualDeviceXD48` is used to connect to a virtual PXDAC4800 device.

Return Value

Returns a non-zero value if the given handle is connected to a virtual device or zero if the handle is not connected to a virtual device.

Remarks

Use this function to determine if the given handle is associated with a virtual PXDAC4800 device. A virtual PXDAC4800 is not connected to real PXDAC4800 hardware and is mainly used for software development and debugging.

Related Functions

[IsHandleValidXD48](#), [IsDeviceRemoteXD48](#)

3.4.2.8 IsHandleValidXD48

Prototype

```
int IsHandleValidXD48 (HXD48 hBrd);
```

Description

Determines if the given PXDAC4800 device handle is connected to a device

Parameters

[in] *hBrd*

The PXDAC4800 device handle to check. A PXDAC4800 device handle is obtained by calling the `ConnectToDeviceXD48` or `ConnectToVirtualDeviceXD48` function.

Return Value

Returns a non-zero value if the given handle is connected to a valid device or zero if the handle is not connected to a valid device.

Remarks

A handle is valid if it is connected to a local, remote, or virtual PXDAC4800 device.

Related Functions

[IsDeviceVirtualXD48](#), [IsDeviceRemoteXD48](#)

3.4.3 PXDAC4800 Device State and Configuration

The functions in this section are used to obtain state and configuration information on the PXDAC4800.

3.4.3.1 GetBoardNameXD48

Prototype

```
int GetBoardNameXD48 (HXD48 hBrd, TCHAR** bufpp, int flags = 0);
```

Description

Obtain user-friendly name for given board

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board for which to obtain the board name. If this parameter is XD48_INVALID_HANDLE the function will return “PXDAC4800” for the board name.

[out] *bufpp*

A pointer to a TCHAR pointer that will receive the address of the library allocated string containing the board name. This buffer must be freed by caller by calling the [FreeMemoryXD48](#) library function.

[in] *flags*

A set of flags that control how the board name is generated:

Flag	Interpretation
XD48GBNF_NO_SN (0x00000001)	Do not include board's serial (or ordinal) number.
XD48GBNF_USE_ORD_NUM (0x00000002)	Use ordinal number instead of serial number
XD48GBNF_INC_VIRT_STATUS (0x00000008)	Include virtual status
XD48GBNF_INC_REMOTE_STATUS (0x00000010)	Include remote status
XD48GBNF_ALPHANUMERIC_ONLY (0x00000020)	Alphanumeric characters only
XD48GBNF_USE_UNDERSCORES (0x00000040)	Use underscores '_' in place of spaces
XD48GBNF_INC_SUB_REVISION (0x00000080)	Include sub-revision info (DP/SP##)
XD48GBNF_USE_DASHES (0x00000100)	Use dashes '-' in place of spaces

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Use this function to have the library allocate and generate a string identifying the given board (e.g. “PXDAC4800 #123456”).

GetBoardNameXD48 is a macro that will expand to _GetBoardNameAXD48 or _GetBoardNameWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Example Usage

```
TCHAR* bufp;

GetBoardNameXD48(hBrd, &bufp);
_tprintf (_T("Board name is %s\n"), bufp);
FreeMemoryXD48(bufp);
```

3.4.3.2 GetBoardRevisionXD48

Prototype

```
int GetBoardRevisionXD48 (HXD48 hBrd, unsigned int* revp, unsigned int* sub_revp);
```

Description

Obtain board revision and/or sub-revision

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board for which to obtain the board name. If this parameter is XD48_INVALID_HANDLE the function will return “PXDAC4800” for the board name.

[out] *revp*

A pointer to the variable that will receive the revision of the device. This parameter may be NULL if the revision is not needed

[out] *sub_revp*

A pointer to the variable that will receive the sub-revision of the device. This parameter may be NULL if the sub-revision is not needed

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The board revision defines the major revision of the underlying device. Currently defined revisions:

- XD48BRDREV_PXDAC4800 - 4 channel 1.2GHz 14-bit DAC PCIe card

Interpretation of the sub-revision depends on the board revision. Currently defined sub-revisions are:

- Revision XD48BRDREV_PXDAC4800
 - XD48BRDREVSUB_0_PXDAC4800_DP - Standard PXDAC4800 device
 - XD48BRDREVSUB_0_PXDAC4800_SP95 - PXDAC4800 with FPGA processing capabilities

3.4.3.3 GetItemVersionXD48

Prototype

```
int GetItemVersionXD48 (HXD48 hBrd, int ver_id, unsigned long long* verp);
```

Description

Obtain the version number of the given item

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *ver_id*

The identifier (XD48VERID_*) of the item for which to obtain the version:

Library Symbolic Constant	Value	Interpretation
XD48VERID_HARDWARE	0	Current device hardware revision
XD48VERID_FIRMWARE	1	Current device firmware package version; overall version of firmware
XD48VERID_DRIVER	2	Current kernel-mode device driver version
XD48VERID_LIBRARY	3	Current user-mode library version
XD48VERID_PRODUCT_SOFTWARE	4	Version of the current PXDAC4800 software installation

[out] *verp*

A pointer to the variable that will receive the version

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

All versioned items have the same format, a 64-bit value broken up into four 16-bit fields:

Field	Mask
Major version	0xFFFF000000000000ULL
Minor version	0x0000FFFF00000000ULL
Sub-minor version	0x00000000FFFF0000ULL
Package number	0x000000000000FFFFULL

In general:

- Major versions are only incremented for large, fundamental changes to the underlying item.
- A non-zero sub-minor version usually indicates an item that was released outside of a general software release. (This is less of a generality for software components on the Linux platform.)
- A non-zero package number usually indicates that the packaging of the item has changed, but the underlying entity has not been changed. An example of this would be the regeneration of a firmware update file in which the underlying firmware has not changed.

Related Functions

[GetVersionTextXD48](#)

3.4.3.4 GetOrdinalNumberXD48

Prototype

```
int GetOrdinalNumberXD48 (HXD48 hBrd, unsigned int* onp);
```

Description

Obtain the ordinal number of the PXDAC4800 connected to the given handle

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *onp*

A pointer to the variable that will receive the PXDAC4800's ordinal number

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

A PXDAC4800's ordinal number is the number of the PXDAC4800 in the system, as in the first, second, third, etc board. This order is determined by the system and may or may not be the same order as the physical PCIe slots.

Related Functions

[GetSerialNumberXD48](#)

3.4.3.5 GetPlaybackClockRateXD48

Prototype

```
int GetPlaybackClockRateXD48 (HXD48 hBrd, double* pRateMHz);
```

Description

Obtain current playback clock rate considering clock source and dividers

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *pRateMHz*

A pointer to a float variable that will receive the effective clock rate in MHz.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Related Functions

[GetPlaybackDataRateXD48](#)

3.4.3.6 GetPlaybackDataRateXD48

Prototype

```
int GetPlaybackDataRateXD48 (HXD48 hBrd, double* pRateMHz);
```

Description

Obtain current playback data rate considering clock source, dividers, and interpolation

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *pRateMHz*

A pointer to a double variable that will receive the effective data rate in MHz.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Related Functions

[GetPlaybackDataRateXD48](#)

3.4.3.7 GetSampleRamSizeXD48

Prototype

```
int GetSampleRamSizeXD48 (HXD48 hBrd, unsigned int* pbyte_count);
```

Description

Get size of PXDAC4800 sample RAM in bytes

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *pbyte_count*

A pointer to an unsigned integer that will receive the total PXDAC4800 RAM size in bytes.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The PXDAC4800 RAM is split into two equal-sized banks. One bank is used for channels 1 and 2 and the other bank is used for channels 3 and 4. This function obtains the aggregate total RAM size over both banks.

Related Functions

[GetMaxByteCountForActiveChanMaskXD48](#)

3.4.3.8 GetSerialNumberXD48

```
int GetSerialNumberXD48 (HXD48 hBrd, unsigned int* snp);
```

Description

Obtain the PXDAC4800 board's serial number

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *snp*

A pointer to the variable that will receive the PXDAC4800's serial number

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Each PXDAC4800 device has a unique serial number over all devices in the same family.

Related Functions

[GetOrdinalNumberXD48](#)

3.4.3.9 GetVersionTextXD48

Prototype

```
int GetVersionTextXD48 (HXD48 hBrd, int ver_id, TCHAR** bufpp, int flags);
```

Description

Obtains a string describing the version of an item

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *ver_id*

The identifier (XD48VERID_*) of the item for which to obtain the version:

Library Symbolic Constant	Value	Interpretation
XD48VERID_HARDWARE	0	Current device hardware revision
XD48VERID_FIRMWARE	1	Current device firmware package version; overall version of firmware

XD48VERID_DRIVER	2	Current kernel-mode device driver version
XD48VERID_LIBRARY	3	Current user-mode library version
XD48VERID_PRODUCT_SOFTWARE	4	Version of the current PXDAC4800 software installation

[out] *bufpp*

A pointer to a char/wchar_t pointer that will receive the address of a library allocated buffer containing the NULL terminated version string. It is the caller's responsibility to free this memory by calling FreeMemoryXD48.

[in] *flags*

A set of flags (XD48VERF_*) that control function behavior:

Symbolic Constant	Value	Interpretation
XD48VERF_NO_PREREL	0x00000001	Do not include pre-release information
XD48VERF_NO_SUBMIN	0x00000006	Do not include sub-minor info; implies XD48VERF_NO_PACKAGE
XD48VERF_NO_PACKAGE	0x00000004	Do not include package info
XD48VERF_NO_CUSTOM	0x00000008	Do not include custom enumeration info
XD48VERF_COMPACT_VER	0x00000010	Compact version string if possible (XX.YY.00.00 -> XX.YY)
XD48VERF_ZERO_PAD_SINGLE_DIGIT_VER	0x00000020	Use 0 padding for one digit version numbers (1.1 -> 1.01)
XD48VERF_ALLOW_ALIASES	0x00000040	Allow function to return aliases for known versions

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to generate a NULL-terminated string representing the version of a particular item.

GetVersionTextXD48 is a macro that will expand to _GetVersionTextAXD48 or _GetVersionTextWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Related Functions

[GetItemVersionXD48](#)

3.4.3.10 ReadConfigEepromXD48

```
int ReadConfigEepromXD48 (HXD48 hBrd, unsigned int eeprom_addr, unsigned short* eeprom_datap);
```

Description

Read an element from the PXDAC4800 configuration EEPROM

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *eeeprom_addr*

The EEPROM address that to be read. Valid addresses that user applications may use are within the range [0xC0, 0xFF]. All values below address 0xC0 are reserved for internal use and are read/write protected.

[out] *eeeprom_datap*

A pointer to the variable that will receive the EEPROM data.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Each PXDAC4800 board has a small configuration EEPROM that is used to contain board-specific configuration data. A region of this EEPROM is set aside for application specific configuration data. The user-defined region is initialized to zeroes as part of the initial board configuration.

Related Functions

[WriteConfigEepromXD48](#)

3.4.3.11 WriteConfigEepromXD48

```
int WriteConfigEepromXD48 (HXD48 hBrd, unsigned int eeeprom_addr, unsigned short eeeprom_data);
```

Description

Write an element from the PXDAC4800 configuration EEPROM

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *eeeprom_addr*

The EEPROM address to be written. Valid addresses that user applications may use are within the range [0xC0, 0xFF]. All values below address 0xC0 are reserved for internal use and are read/write protected.

[in] *eeeprom_data*

The value that will be written to the specified EEPROM address.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Related Functions

[ReadConfigEepromXD48](#)

3.4.4 PXDAC4800 Device Status

The functions in this section are used for obtaining various device and driver status information.

3.4.4.1 GetDacOutputFifoFullFlagXD48

Prototype

```
int GetDacOutputFifoFullFlagXD48 (HXD48 hBrd);
```

Description

Get state of DAC output FIFO almost full flag from PXDAC4800 hardware

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns 1 if DAC output FIFO is full, 0 if DAC output FIFO is not full, or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to read the state of the DAC Output FIFO full flag from the PXDAC4800 hardware. (There is actually one output DAC Output FIFO per channel, but they are all synchronous and so go full at the same time.)

The DAC Output FIFO is the final FIFO just before the actual DAC. When this FIFO is full then the card is considered primed and ready to begin playback. If a trigger is received before the DAC Output FIFO has had a chance to full there is a chance that the FIFO will go empty which will have the effect of gaps in the playback data. When doing a playback operation, the programmer may consider disabling the external trigger until the DAC Output FIFO has gone full.

The BeginRamPlaybackXD48 library function takes the DAC Output FIFO full flag into consideration. After arming the card for RAM playback, the function will wait (up to 100ms) for the DAC Output FIFO to go full before returning.

The Streaming Playback Session implementation takes the DAC Output FIFO into consideration. When playback data is being uploaded to the card, the playback thread will check the status of the DAC Output FIFO full flag and when set, it will set the XD48SPBPROGF_DAC_FIFOS_FILLED flag (which can be read by the main thread by calling SessionStreamProgressXD48). At this point, if the XD48STREAMF_FORCE_SW_TRIGGER flag was specified the playback thread will issue a software-generated trigger.

3.4.4.2 GetDriverStatsXD48

Prototype

```
int GetDriverStatsXD48 (HXD48 hBrd, XD48S\_DRIVER\_STATS* statsp, int bReset = 0);
```

Description

Obtain PXDAC4800 driver/device statistics

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *statsp*

A pointer to a XD48S_DRIVER_STATS object that will receive driver statistics. The caller must initialize the struct_size member to the size of the XD48S_DRIVER_STATS object before calling the function.

[in] *bReset*

If this parameter is non-zero then all driver stat counters will be reset to zero after they are obtained for the request.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Example Usage

```
XD48S_DRIVER_STATS drv_stats;  
drv_stats.struct_size = sizeof(XD48S_DRIVER_STATS);  
GetDriverStatsXD48(hBrd, &drv_stats);
```

3.4.4.3 GetSamplesCompleteInterruptCountXD48

Prototype

```
int GetSamplesCompleteInterruptCountXD48 (HXD48 hBrd, unsigned int* intr_counterp);
```

Description

Obtain the current Samples Complete interrupt counter value

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[out] *intr_counterp*

A pointer to an unsigned integer variable that will receive the current Samples Complete interrupt counter value.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Each time the PXDAC4800 driver handles a Samples Complete interrupt, the Samples Complete interrupt counter is incremented. This function can be used to obtain this counter value. This information may be of interest when the card is configured for externally triggered output. The counter value can be monitored by the software to ensure that card is receiving triggers.

Caution: Programmers should be careful about using the interrupt counter obtained from this function in conjunction with the [WaitForSamplesCompleteXD48](#) function. The reason for this is that it is possible for interrupts to come in between the point at which the interrupt count was obtained and the call to [WaitForSamplesCompleteXD48](#).

Related Functions

[WaitForSamplesCompleteXD48](#)

3.4.4.4 InIdleModeXD48

Prototype

```
int InIdleModeXD48 (HXD48 hBrd);
```

Description

Determine if the PXDAC4800 is idle; in Standby or Off mode

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns a positive value if device is idle (in Standby or Off mode).

Returns 0 if device is in an active (playback or transfer) operating mode.

Returns one of the [library error codes](#) (which are all negative) on error.

Remarks

This function always goes to the device driver to obtain the current operating mode rather than relying on the local register cache.

Related Functions

[_SetOperatingModeXD48](#), [InPlaybackModeXD48](#)

3.4.4.5 InPlaybackModeXD48

Prototype

```
int InPlaybackModeXD48 (HXD48 hBrd);
```

Description

Determine if the PxDAC4800 is in a playback mode

Parameters

[in] *hBrd*

A handle to the PxDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns a positive value if the board is in a playback mode.

Returns 0 if board is not in a playback operating mode.

Returns one of the [library error codes](#) (which are all negative) on error.

Remarks

This function always goes to the device driver to obtain the current operating mode rather than relying on the local register cache.

Related Functions

[_SetOperatingModeXD48](#), [InIdleModeXD48](#)

3.4.5 PxDAC4800 Hardware Settings

The functions in this section control the getting and setting of the various PxDAC4800 hardware settings. These settings include the numerous data playback, clock, and trigger parameters.

3.4.5.1 IssueSoftwareTriggerXD48

Prototype

```
int IssueSoftwareTriggerXD48 (HXD48 hBrd);
```

Description

Issue a [software-generated trigger](#) event to a PxDAC4800

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The function IssueSoftwareTriggerXD48 allows the user to issue a software trigger to the PXDAC4800 board.

Related Functions

[SetTriggerModeXD48](#), [SetExternalTriggerEnableXD48](#)

3.4.5.2 SetActiveChannelMaskXD48

Prototype

<code>int SetActiveChannelMaskXD48 (HXD48 hBrd, int val);</code>
<code>int GetActiveChannelMaskXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the [Active Channels](#) selection; defines which channels are played back

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The [active channel mask](#) to set. The library defines symbolic constants for all valid channel masks:

PXDAC4800 Library Constant	Interpretation
XD48CHANMASK_4_CHANNEL (0xF)	Four channel: channels 1, 2, 3 and 4
XD48CHANMASK_2_CHANNEL_1_2 (0x3)	Dual channel: channels 1 and 2
XD48CHANMASK_2_CHANNEL_3_4 (0xC)	Dual channel: channels 3 and 4
XD48CHANMASK_1_CHANNEL_1 (0x1)	Single channel: channel 1
XD48CHANMASK_1_CHANNEL_2 (0x2)	Single channel: channel 2
XD48CHANMASK_1_CHANNEL_3 (0x4)	Single channel: channel 3
XD48CHANMASK_1_CHANNEL_4 (0x8)	Single channel: channel 4

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetActiveChannelMaskXD48 will return the current active channel selection.

Remarks

See [Active Channel Masks](#) section for more details on channel masks.

Related Functions

[GetChanCountFromMaskXD48](#)

3.4.5.3 SetClockDividerNXD48

Prototype

<code>int</code>	<code>SetClockDivider1XD48</code>	<code>(</code>	<code>HXD48</code>	<code>hBrd,</code>	<code>int</code>	<code>div1);</code>
<code>int</code>	<code>GetClockDivider1XD48</code>	<code>(</code>	<code>HXD48</code>	<code>hBrd,</code>	<code>int</code>	<code>bFromCache = 1);</code>
<code>int</code>	<code>SetClockDivider2XD48</code>	<code>(</code>	<code>HXD48</code>	<code>hBrd,</code>	<code>int</code>	<code>div1);</code>
<code>int</code>	<code>GetClockDivider2XD48</code>	<code>(</code>	<code>HXD48</code>	<code>hBrd,</code>	<code>int</code>	<code>bFromCache = 1);</code>

Description

Get or set clock divider values

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *div1*

The divider to use for clock divider #1. This can be any value from 1 to 32 (except 3).

[in] *div2*

The divider to use for clock divider #2. This can be any value from 1 to 6 (except 3).

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetClockDivider1XD48 and returns the current clock divider #1 setting.

On success, GetClockDivider2XD48 and returns the current clock divider #2 setting.

Remarks

Do not change the clock source or frequency while a playback or data transfer is in progress.

The PXDAC4800 has two external clock dividers that operate in series. Clock divider #1 can be any value from 1 to 32. Clock divider #2 can be any value from 1 to 6. The two dividers yield 108 unique clock division combinations for a maximum division of 192.

Related Functions

[SetPlaybackClockSourceXD48](#)

3.4.5.4 SetDacInterpolationEnableXD48

Prototype

<code>int SetDacInterpolationEnableXD48 (HXD48 hBrd, int bEnable);</code>
<code>int GetDacInterpolationEnableXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the DAC interpolation (2x) enable

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *bEnable*

If this parameter is non-zero then the DAC 2x interpolation will be enabled. If this parameter is zero then the DAC 2x interpolation will be disabled.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetDacInterpolationEnableXD48 returns the current DAC interpolation enable setting.

Remarks

This function is not available while a playback or data transfer is in progress.

When interpolation is enabled the DACs will interpolate playback data by a factor of 2. When interpolation is enabled, the playback clock rate will be the same but the playback data rate (the rate at which data is consumed) will be halved since the DAC is providing one additional sample for every sample of input.

This hardware setting affects all DACs; mixed interpolation is not allowed.

3.4.5.5 SetDacSampleFormatXD48

Prototype

<code>int SetDacSampleFormatXD48 (HXD48 hBrd, int val);</code>
<code>int GetDacSampleFormatXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the DAC sample format; e.g. signed or unsigned data

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The DAC sample format to set:

Library Symbolic Constant	Interpretation
XD48SAMPFMT_UNSIGNED (0)	Data samples are interpreted as unsigned 14 bits: [0, 16383], 8 bits: [0, 256] (power up default)
XD48SAMPFMT_SIGNED (1)	Data samples are interpreted as signed 14 bits: [-8192, 8191], 8 bits: [-128, 127]

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetDacSampleFormatXD48 returns the current DAC sample format setting.

Remarks

This hardware setting affects all channels; mixed channel formats are not allowed.

Related Functions

[SetDacSampleSizeXD48](#)

3.4.5.6 SetDacSampleSizeXD48

Prototype

<code>int SetDacSampleSizeXD48 (HXD48 hBrd, int val);</code>
<code>int GetDacSampleSizeXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the DAC sample size/padding

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The DAC sample size to set:

Library Symbolic Constant	Interpretation
XD48SAMP_SIZE_8BIT (0)	8-bit (0xFF)
XD48SAMP_SIZE_14BIT_MSBPAD (1)	14-bit, 16-bit aligned with MSB zero-padded (0x3FFF)
XD48SAMP_SIZE_14BIT_LSBPAD (2)	14-bit, 16-bit aligned with LSB zero-padded (0xFFFC) (power-up default)

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetDacSampleSizeXD48 returns the current DAC sample size/padding setting.

Remarks

This hardware setting affects all channels; mixed channel formats are not allowed.

Related Functions

[SetDacSampleFormatXD48](#)

3.4.5.7 StartDacAutoCalibrationXD48

Prototype

```
int StartDacAutoCalibrationXD48 (HXD48 hBrd);
```

Description

Perform a manual DAC's calibration.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

3.4.5.8 SetExternalPlaybackClockRateXD48

Prototype

<code>int SetExternalPlaybackClockRateXD48 (HXD48 hBrd, double dRateMHz);</code>
<code>int GetExternalPlaybackClockRateXD48 (HXD48 hBrd, double* ratep, int bFromCache = 1);</code>

Get or set the [external clock](#) rate in MHz

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *dRateMHz*

The external clock rate in MHz. This value must be a number in the range [1, 1200] MHz.

[out] *ratep*

A pointer to the variable that will receive the current assumed external clock rate.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Do not change the external playback clock rate while a playback or data transfer is in progress.

This setting is only relevant when the external clock is selected as the playback clock source.

This function should be called any time the external clock rate changes when the external clock is selected as the acquisition clock source. This will ensure that the PXDAC4800 firmware is synchronized properly with the playback clock. Failure to call this function when the external clock frequency is changed can result in bad playback data.

The GetPlaybackClockRateXD48 function implementation will use this rate when calculating the effective clock rate when the external clock is selected.

Related Functions

[SetPlaybackClockSourceXD48](#)

3.4.5.9 SetExternalReferenceClockEnableXD48

Prototype

<code>int SetExternalReferenceClockEnableXD48 (HXD48 hBrd, int bEnable);</code>
<code>int GetExternalReferenceClockEnableXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the external 10MHz reference clock enable; overrides internal 10MHz reference

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *bEnable*

If this parameter is non-zero then the PXDAC4800 will be configured to use an externally provided 10MHz reference. If this parameter is zero then the PXDAC4800 will be configured to use the internal 10MHz reference clock.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetExternalReferenceClockEnableXD48 will return the current external reference enable selection.

Remarks

The 10MHz reference is only used when one of the internal playback clocks are selected.

Calling this function while a playback or data transfer is in progress can cause issues.

Related Functions

[SetPlaybackClockSourceXD48](#)

3.4.5.10 SetExternalTriggerDirXD48

Prototype

<code>int SetExternalTriggerDirXD48 (HXD48 hBrd, int val);</code>
<code>int GetExternalTriggerDirXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the external trigger pulse edge to use for a trigger event

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The identifier of the external trigger pulse edge to set.

PXDAC4800 Library Constant	Interpretation
XD48TRIGDIR_POS (0)	Positive-going (rising) edge (power-up default)
XD48TRIGDIR_NEG (1)	Negative-going (falling) edge

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetExternalTriggerDirXD48 will return the current external trigger pulse edge selection.

Remarks

This setting is relevant only when an external trigger is being used to trigger data playback.

Related Functions

[SetExternalTriggerEnableXD48](#)

3.4.5.11 SetExternalTriggerEnableXD48

Prototype

<code>int SetExternalTriggerEnableXD48 (HXD48 hBrd, int bEnable);</code>
<code>int GetExternalTriggerEnableXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or sets the external trigger enable; allows externally provided triggers to trigger playback

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *bEnable*

If this parameter is non-zero then the external trigger will be enabled. If this parameter is zero then external triggers will be ignored by the PXDAC4800.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetExternalTriggerEnableXD48 returns the current external trigger enable setting.

Related Functions

[SetTriggerModeXD48](#), [IssueSoftwareTriggerXD48](#)

3.4.5.12 SetDigitalIoCfgXD48

Prototype

<code>int SetDigitalIoCfgXD48 (HXD48 hBrd, int bOutput);</code>
<code>int GetDigitalIoCfgXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the digital IO configuration (output or input)

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *bOutput*

The identifier of the digital IO configuration

PXDAC4800 Library Constant	Interpretation
XD48DIGIOCFG_OUT (0)	Digital output (power-up default)
XD48DIGIOCFG_IN (1)	Digital input

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetDigitalIoCfgXD48 will return the current digital IO configuration

3.4.5.13 SetDigitalIoModeXD48

Prototype

```
int SetDigitalIoModeXD48 (HXD48 hBrd, int val);  
int GetDigitalIoModeXD48 (HXD48 hBrd, int bFromCache = 1);
```

Description

Get or set the digital IO mode (output or input)

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The identifier of the digital IO mode

PXDAC4800 Library Constant	Interpretation
XD48DIGIOMODE_OUT_CLOCK_DIV_8 (0)	DATA_CLK / 8 : max of 150MHz (power up default)
XD48DIGIOMODE_OUT_PULSE_BEGIN_PLAYBACK (1)	Pulse at the beginning of a playback. Not synchronous with the data
XD48DIGIOMODE_OUT_PULSE_END_PLAYBACK (2)	Pulse at the end of a playback. Not synchronous with the data
XD48DIGIOMODE_OUT_DAC_PLAYING_DATA (3)	DAC are playing data. Not synchronous with the data
XD48DIGIOMODE_OUT_DAC_PULSE_UNDERFLOW (4)	Pulse at Underflow error
XD48DIGIOMODE_OUT_FRAME_START (5)	Pulse at start of frame. See Remarks.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetDigitalIoModeXD48 will return the current digital IO mode

Remarks

No input modes defined yet

The Frame Start Output is only available in RAM playback operating mode.

The Frame Start Output is useful for simulating raster scanning signals, which often incorporate a fast Line trigger and a slow Frame trigger. For example, an old-school cathode ray television uses a fast Line trigger to trigger the 525 picture lines and a slow Frame Trigger to determine which is Line #1 in order to correctly display the picture. In RAM play back mode, the user can upload a portfolio of N waveform segments to PXDAC48000 memory. Successive triggers will create the endless waveform sequence 1,2,3...N-1, N,1,2,3... In this case, when selected as the Digital Output, Frame Start will output a pulse upon generation of waveform segment #1 only. Accordingly, the Frame Start pulse will occur only once for every N triggers.

3.4.5.14 SetOutputVoltageChNXD48

Prototype

int SetOutputVoltageCh1XD48 (HXD48 hBrd, int val);
int SetOutputVoltageCh2XD48 (HXD48 hBrd, int val);
int SetOutputVoltageCh3XD48 (HXD48 hBrd, int val);
int SetOutputVoltageCh4XD48 (HXD48 hBrd, int val);
int GetOutputVoltageCh1XD48 (HXD48 hBrd, int bFromCache = 1);
int GetOutputVoltageCh2XD48 (HXD48 hBrd, int bFromCache = 1);
int GetOutputVoltageCh3XD48 (HXD48 hBrd, int bFromCache = 1);
int GetOutputVoltageCh4XD48 (HXD48 hBrd, int bFromCache = 1);

Description

Get or set the channel's output voltage range

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

An integer in the range of [0, 1023] that represents the output voltage range for the channel. A value of 0 is equivalent to the minimum output voltage range and value of 1023 is equivalent to the maximum output voltage range.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetOutputVoltageCh\XD48 will return the current output voltage range setting.

3.4.5.15 SetPlaybackClockSourceXD48

Prototype

<code>int SetPlaybackClockSourceXD48 (HXD48 hBrd, int val);</code>
<code>int GetPlaybackClockSourceXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the PXDAC4800's playback [Clock Source](#) setting

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The identifier of the DAC clock source to set. See remarks.

PXDAC4800 Library Constant	Interpretation
XD48CLKSRC_INT_1200_MHZ (0)	Internal 1200 MHz oscillator
XD48CLKSRC_INT_900_MHZ (1)	Internal 900 MHz oscillator
XD48CLKSRC_EXTERNAL (2)	Externally provided clock

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetPlaybackClockSourceXD48 will return the current playback clock source selection.

Remarks

Do not change the clock source or frequency while a playback or data transfer is in progress.

Internal Clocks

The PXDAC4800 has two internal clocks. The internal clocks are phase locked to a 10 MHz reference clock, which may be externally provided.

- XD48CLKSRC_INT_1200_MHZ (0) : 1200 MHz oscillator
- XD48CLKSRC_INT_900_MHZ (1) : 900 MHz oscillator

External clock

The PXDAC4800 supports an externally provided clock. When the external clock is selected, the SetExternalPlaybackClockRateXD48 function should be called to specify the external clock rate. The PXDAC4800 software will need this information so that it can properly configure the hardware.

- XD48CLKSRC_EXTERNAL (2) : External clock

Related Functions

[SetExternalPlaybackClockRateXD48](#), [SetClockDivider1XD48](#), [SetClockDivider2XD48](#)

3.4.5.16 SetTriggerModeXD48

Prototype

<code>int SetTriggerModeXD48 (HXD48 hBrd, int val);</code>
<code>int GetTriggerModeXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set [trigger mode](#); relates how trigger events affect data playback.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

Specifies the trigger mode to set and may be one of the following:

PXDAC4800 Library Constant	Interpretation
XD48TRIGMODE_PLAY_PER_TRIGGER (0)	Single start trigger runs memory data once
XD48TRIGMODE_CONTINUOUS (1)	Single start trigger runs memory data repeatedly (power up default)
XD48TRIGMODE_SINGLE_SHOT (2)	Trigger runs memory data once; subsequent triggers ignored

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetTriggerModeXD48 returns the current trigger mode setting.

Related Functions

[SetExternalTriggerEnableXD48](#)

3.4.5.17 **_SetActiveMemoryRegionXD48**

Prototype

```
int _SetActiveMemoryRegionXD48 (  
    HXD48 hBrd,  
    unsigned int offset_bytes,  
    unsigned int length_bytes,  
    int flags = 0);  
  
int _GetActiveMemoryRegionXD48 (  
    HXD48 hBrd,  
    unsigned int* poffset_bytes,  
    unsigned int* plength_bytes,  
    int bFromCache = 1);
```

Description

Get or set the PXDAC4800 [Active Memory Region](#) for subsequent playbacks and transfers. The PXDAC4800 library usually handles this automatically. See Remarks

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *offset_bytes*

The byte offset into PXDAC4800 memory at which to begin playing or transferring data. This parameter should be an integer multiple of 8192 and less than the maximum total sample count. See remarks.

[in] *length_bytes*

This parameter defines the total number of bytes that will be considered for subsequent playback or data transfer operations. This parameter is independent of channel count. This parameter should be an integer multiple of 8192 and not larger than 2147483648. See Remarks. The function will realign the parameter down if necessary. The function will clip the parameter to the maximum sample count if necessary. If this parameter is XD48_FREE_RUN (0), then the board will be configured for free-run mode. Free-run mode is only available for RAM-buffered PCIe playback and is used to playback an infinite, continuous stream of data (or an infinite stream of segments if using segmented triggering).

[out] *poffset_bytes*

A pointer to an unsigned integer that will receive the current byte offset. This parameter may be NULL.

[out] *plength_bytes*

A pointer to an unsigned integer that will receive the current byte length. This parameter may be NULL.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to setup the [Active Memory Region](#). The active memory region defines the area of PXDAC4800 memory that will be used for subsequent RAM acquisition or data transfer operations.

This function is only explicitly needed in cases where the code is explicitly controlling the board's operating mode. All library functions that wrap a data acquisition or transfer (e.g. [BeginRamPlaybackXD48](#), [LoadStreamDataFastXD48](#), etc) will manage the setup of the active memory region.

Caution: The PXDAC4800 supports playback of both 8- and 14-bit (16-bit aligned) data samples. Because of the varying sample size, this function works in terms of bytes, not samples. If the DAC sample size is set to one of the 14-bit sample formats, then each sample is two bytes.

Internally, the PXDAC4800 RAM is broken up into two equal sized logical banks; one bank for channels 1 and 2 and one bank for channels 3 and 4. Each logical bank is 1073741824 bytes (1GiB) in size making for an aggregate total RAM size of 2147483648 bytes (2GiB). Because of this, addressing the acquisition data in memory depends on the current Active Channel Mask setting (i.e. [SetActiveChannelMaskXD48](#)). This function takes the current active channels setting into consideration when configuring the active memory selection such that from the point of view of the caller, the PXDAC4800 memory is a single contiguous block of memory, regardless of what the current active channel selection is.

Maximum Byte Length

Because of the internally split memory, the maximum valid byte length depends on the current Active Channel Mask ([SetActiveChannelMaskXD48](#)) setting:

Active Channels Selection	Maximum Sample Count
XD48CHANMASK_4_CHANNEL	2,147,483,648 total bytes (or 536,870,912 bytes per channel)
XD48CHANMASK_2_CHANNEL_1_3	2,147,483,648 total bytes (or 1,073,741,824 bytes per channel)
XD48CHANMASK_2_CHANNEL_1_2	1,073,741,824 total bytes (or 536,870,912 bytes per channel)
XD48CHANMASK_2_CHANNEL_3_4	1,073,741,824 total bytes (or 536,870,912 bytes per channel)
XD48CHANMASK_1_CHANNEL_X	1,073,741,824 total bytes

Byte Offset Selection

The `offset_bytes` parameter is interpreted as the 0-based index of the byte in memory. For multichannel data, channel data is sample interleaved:

Active Channels Selection	Sample Index 0, 1, 2, 3, ...
XD48CHANMASK_4_CHANNEL	Ch 1, Ch 2, Ch 3, Ch 4, Ch 1, Ch 2, ...
XD48CHANMASK_2_CHANNEL_1_3	Ch 1, Ch 3, Ch 1, Ch 3, ...
XD48CHANMASK_2_CHANNEL_1_2	Ch 1, Ch 2, Ch 1, Ch 2, ...
XD48CHANMASK_2_CHANNEL_3_4	Ch 3, Ch 4, Ch 3, Ch 4, ...
XD48CHANMASK_1_CHANNEL_X	Ch X, Ch X, ...

For *CT* total channels of data, to jump to 0-based sample index *N* of 0-based channel index *c*, the starting sample would be:

$$(N * CT) + c$$

Due to alignment restrictions, the starting sample will always be aligned to the first channel of the active channels selection.

For example, if the active channel selection is XD48CHANMASK_4_CHANNEL and we wanted to start transferring from byte 1048576 of channel 1:

```
CT = 4           (4 channels of data)
c = 0           (first channel, index 0)
N = 1048576     (0-based byte index)
offset_bytes = (1048576 * 4) + 0 = 4194304
```

Related Functions

[SetActiveChannelMaskXD48](#)

3.4.5.18 _SetOperatingModeXD48

Prototype

<code>int _SetOperatingModeXD48 (HXD48 hBrd, int val);</code>
<code>int _GetOperatingModeXD48 (HXD48 hBrd, int bFromCache = 1);</code>

Description

Get or set the PXDAC4800's [operating mode](#). See remarks

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PXDAC4800 handle, which will result in no hardware or driver access. If zero, the setting is obtained from the driver which may or may not result in an actual PXDAC4800 device register read.

[in] *val*

The operating mode to switch to. This can be any of the following values:

PXDAC4800 Library Constant	Interpretation
XD48MODE_STANDBY (1)	Standby (ready) mode
XD48MODE_PLAY_RAM (2)	RAM playback mode
XD48MODE_PLAY_PCIE_BUF (3)	RAM-buffered PCIe playback mode; data buffered through PXDAC4800 RAM. <u>Never set this mode directly; instead use BeginStreamingPlaybackXD48 library function.</u>
XD48MODE_LOAD_RAM (4)	Load RAM mode; load data for RAM playback. <u>Never set this mode directly; instead use LoadRamBufXD48 library function.</u>

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, _GetOperatingModeXD48 returns the current operating mode setting.

Remarks

For most all active operating modes the PXDAC4800 library implements higher level routines that automatically manage the operation performed with the operating mode. These higher-level routines are shown in the following table.

Underlying Operating Mode	Wrapper Function(s)
XD48MODE_PLAY_RAM	BeginRamPlaybackXD48
XD48MODE_PLAY_PCIE_BUF	BeginStreamingPlaybackXD48
XD48MODE_LOAD_RAM	LoadRamBufXD48 LoadFileIntoRamXD48

3.4.5.19 SetFiltersCHXD48

Prototype

<code>int SetFiltersCHXD48 (HXD48 hBrd, int val);</code>
<code>int GetFiltersCHXD48 (HXD48 hBrd);</code>

Description

Enable or disable the 50 MHz low pass filter on the different channel of the PXDAC4800.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *value*

The library defines symbolic constants for all valid filter masks:

PXDAC4800 Library Constant	Interpretation
XD48CHAN_FILTERMASK_NONE (0)	None filter channel enable
XD48CHAN_FILTERMASK_1 (0x1)	Filter enable: Ch1
XD48CHAN_FILTERMASK_2 (0x2)	Filter enable: Ch2
XD48CHAN_FILTERMASK_3 (0x4)	Filter enable: Ch3
XD48CHAN_FILTERMASK_4 (0x8)	Filter enable: Ch4
XD48CHAN_FILTERMASK_1_2 (0x3)	Filter enable: Ch1, Ch2
XD48CHAN_FILTERMASK_1_3 (0x5)	Filter enable: Ch1, Ch3
XD48CHAN_FILTERMASK_1_4 (0x9)	Filter enable: Ch1, Ch4
XD48CHAN_FILTERMASK_2_3 (0x6)	Filter enable: Ch2, Ch3
XD48CHAN_FILTERMASK_2_4 (0xA)	Filter enable: Ch2, Ch4
XD48CHAN_FILTERMASK_3_4 (0xC)	Filter enable: Ch3, Ch4

XD48CHAN_FILTERMASK_1_2_3 (0x7)	Filter enable: Ch1, Ch2, Ch3
XD48CHAN_FILTERMASK_1_2_4 (0xB)	Filter enable: Ch1, Ch2, Ch4
XD48CHAN_FILTERMASK_1_3_4 (0xD)	Filter enable: Ch1, Ch3, Ch4
XD48CHAN_FILTERMASK_2_3_4 (0xE)	Filter enable: Ch2, Ch3, Ch4
XD48CHAN_FILTERMASK_1_2_3_4 (0xF)	Filter enable: Ch1, Ch2, Ch3, Ch4

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetFilterXD48 returns the current filters state.

3.4.5.20 SetCustomDacValueEnableXD48

Prototype

int SetCustomDacValueEnableXD48 (HXD48 hBrd, int val);
int GetCustomDacValueEnableXD48 (HXD48 hBrd);

Description

Enable or disable the possibility for the user to use a custom (non-zero) default value. This way, the DAC will output whatever the user wants when the card is not in playback.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *value*

The library defines symbolic constants for all valid DAC default enable masks:

PXDAC4800 Library Constant	Interpretation
XD48_DACDEFAULT_ENABLE_NONE (0)	None DAC default value enable
XD48DACDEFAULT_ENABLE_1 (0x1)	DAC default value enable: Ch1
XD48DACDEFAULT_ENABLE_2 (0x2)	DAC default value enable: Ch2
XD48DACDEFAULT_ENABLE_3 (0x4)	DAC default value enable: Ch3
XD48DACDEFAULT_ENABLE_4 (0x8)	DAC default value enable: Ch4
XD48DACDEFAULT_ENABLE_1_2 (0x3)	DAC default value enable: Ch1, Ch2
XD48DACDEFAULT_ENABLE_1_3 (0x5)	DAC default value enable: Ch1, Ch3
XD48DACDEFAULT_ENABLE_1_4 (0x9)	DAC default value enable: Ch1, Ch4
XD48DACDEFAULT_ENABLE_2_3 (0x6)	DAC default value enable: Ch2, Ch3
XD48DACDEFAULT_ENABLE_2_4 (0xA)	DAC default value enable: Ch2, Ch4
XD48DACDEFAULT_ENABLE_3_4 (0xC)	DAC default value enable: Ch3, Ch4
XD48DACDEFAULT_ENABLE_1_2_3 (0x7)	DAC default value enable: Ch1, Ch2, Ch3
XD48DACDEFAULT_ENABLE_1_2_4 (0xB)	DAC default value enable: Ch1, Ch2, Ch4
XD48DACDEFAULT_ENABLE_1_3_4 (0xD)	DAC default value enable: Ch1, Ch3, Ch4
XD48DACDEFAULT_ENABLE_2_3_4 (0xE)	DAC default value enable: Ch2, Ch3, Ch4
XD48DACDEFAULT_ENABLE_1_2_3_4 (0xF)	DAC default value enable: Ch1, Ch2, Ch3, Ch4

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetCustomDacDefaultValueEnableXD48 returns the current state.

Remarks

Custom DAC values are disabled during some function call, like clock settings changes or sample format. At that moment, the DAC output is 0V.

3.4.5.21 SetCustomDacDefaultValueXD48

Prototype

<code>int SetCustomDacDefaultValueXD48 (HXD48 hBrd, int channel, int value);</code>
<code>int GetCustomDacDefaultValueXD48 (HXD48 hBrd, int channel);</code>

Description

Get or set a custom default value for a specific channel.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *channel*

The channel ID (1 to 4).

[in] *value*

The value in sample.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

On success, GetCustomDacDefaultValueXD48 returns the default value of the specific channel.

Remarks

Custom DAC values are disabled during some function call, like clock settings changes or sample format. At that moment, the DAC output is 0V.

3.4.6 Device Register State Functions

The functions in this section involve manipulation of PXDAC4800 device registers

3.4.6.1 CopyHardwareSettingsXD48

Prototype

```
int CopyHardwareSettingsXD48 (HXD48 hBrdDst, HXD48 hBrdSrc, int flags);
```

Description

Copy hardware settings from another PXDAC4800 device

Parameters

[in] *hBrdDst*

A handle to the PXDAC4800 board that hardware settings will be copied to. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *hBrdSrc*

A handle to the PXDAC4800 board that hardware settings will be copied from. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *flags*

A set of flags (XD48CHS_*) that control function behavior. This parameter should be zero for most callers. Currently defined flags:

Flag	Interpretation
XD48CHS_SIMPLE_COPY (0x80000000)	The function will blindly copy all software-selectable hardware settings. This will override all of the special case rules outlined in the Remarks section.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Calling this function will copy all hardware settings of the PXDAC4800 associated with the source device to the PXDAC4800 associated with the destination device. The function will use the hardware settings as they are defined in the handle-specific software register cache.

Operating mode is not copied and the destination board is placed into Standby operating mode before any settings are applied.

If source and destination devices are the same, none of the subsequent special cases apply; all settings are "copied".

3.4.6.2 SetPowerupDefaultsXD48

Prototype

```
int SetPowerupDefaultsXD48 (HXD48 hBrd);
```

Description

Restores all PXDAC4800 settings to power-up default values

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Call this function to cancel all current operations and restore the PXDAC4800 back to power up conditions. The following are side-effects of this function:

- Any DMA transfers (synchronous or asynchronous) in progress are cancelled
- Any playback operations in progress are cancelled
- Any threads waiting for a data transfer or playback to complete will be resumed with cancelled status. In this case, library functions will return SIG_CANCELLED to indicate the operation was cancelled.
- These side-effects are system-wide, including other processes or threads, but only for the device associated with the given handle.

The board is put into Standby mode prior to applying default values.

Calling this function will reset all hardware settings to their default power up values. The following table lists all power up default values:

Hardware Setting	Power Up Default	Related Library Function
Active Channel Mask	Four channel (XD48CHANMASK_4_CHANNEL)	SetActiveChannelMaskXD48
DAC Sample Format	Unsigned (XD48SAMPFMT_UNSIGNED)	SetDacSampleFormatXD48
DAC Sample Size	16-bit, LSB padded (XD48SAMPSIZE_14BIT_LSBPAD)	SetDacSampleSizeXD48
Playback Clock Source	Internal 1200MHz Oscillator (XD48CLKSRC_INT_1200_MHZ)	SetPlaybackClockSourceXD48
Clock Divider #1	1 (no division)	SetClockDivider1XD48
Clock Divider #2	1 (no division)	SetClockDivider2XD48
External Reference Clock Enable	Disabled; internal 10MHz reference used	SetExternalReferenceClockEnableXD48
External Playback Clock Rate	1200 MHz	SetExternalPlaybackClockRateXD48
DAC Interpolation Enable	Disabled; no interpolation	SetDacInterpolationEnableXD48
External Trigger Direction	Positive-going (XD48TRIGDIR_POS)	SetExternalTriggerDirXD48
Trigger Mode	Continuous (XD48TRIGMODE_CONTINUOUS)	SetTriggerModeXD48
External Trigger Enable	Enabled	SetExternalTriggerEnableXD48
Output Voltage Range	600mV _{p-p}	SetOutputVoltageChN/XD48

3.4.7 Memory/DMA Buffer Allocation Routines (Advanced users ONLY)

These functions in this section pertain to memory allocation and freeing.

3.4.7.1 AllocateDmaBufferXD48

```
int AllocateDmaBufferXD48 (HXD48 hBrd, unsigned int bytes, void** bufpp);
```

Description

Allocate a DMA buffer and map into address space of calling process

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bytes*

The number of bytes to allocate for the buffer. There is no explicit upper bound to the size of a DMA buffer, it is entirely dependent on the available resources and the host operating system. This parameter will be aligned up to the underlying system's page size.

[out] *bufpp*

A pointer to a DMA buffer pointer that will receive the virtual address of the DMA buffer. This buffer is fully mapped into the address space of the calling process. That is, it can be treated just like normal memory.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to allocate physically contiguous, non-paged space in PC memory for DMA transfers. The allocated buffer (or any 8-byte aligned region therein) may then be used by functions that require a DMA buffer. Note that the only way to get data from the board directly to the PC is via a DMA transfer. A DMA buffer must be freed via the [FreeDmaBufferXD48](#) function when it is no longer needed.

The DMA buffer may only be used with the PXDAC4800 device that it was allocated for. A DMA buffer can be used with another PXDAC4800 device handle (HXD48) of the same process as long as both device handles refer to the same underlying XD48 device.

The maximum size of a DMA buffer is system-dependent. The longer a system has been up and running, the more the physical memory becomes fragmented and the harder it is for the operating system to allocate large DMA buffers. It is highly recommended to allocate your DMA buffers as early as possible after the system starts up and to hang on to them.

On most platforms, the actual amount of memory allocated will usually be rounded up to the system page boundary. (Typically 4096 bytes.)

Microsoft Windows: The virtual address of any allocated DMA buffer should be on a 64KB boundary. This allows DMA buffers to be used for non-buffered file IO routines. (See FILE_FLAG_NO_BUFFERING documentation for the Win32 CreateFile function.)

The PXDAC4800 driver will ensure that all un-freed DMA buffers for a given process are freed when that process' last handle is closed. That is to say, like normal, heap-allocated memory, the underlying PXDAC4800 software will ensure that all DMA buffers are properly cleaned up when a process exits, gracefully or not.

Related Functions

[FreeDmaBufferXD48](#)

3.4.7.2 EnsureUtilityDmaBufferXD48

Prototype

```
int EnsureUtilityDmaBufferXD48 (HXD48 hBrd, int buf_idx, unsigned int byte_count);
```

Description

Ensures that the library managed utility DMA buffer is of the given size

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *buf_idx*

The 0-based index of the utility DMA buffer to consider. The PXDAC4800 library currently supports 2 utility DMA buffers per device handle so this parameter may be 0 or 1.

[in] *byte_count*

The minimum byte count that the utility DMA buffer for the given PXDAC4800 handle should be. Note that this is a byte count, not a sample count. See remarks.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Each PXDAC4800 device handle can have up to two application specific utility DMA buffers associated with the device handle. These utility DMA buffers are not used by the PXDAC4800 software and are reserved for application specific use. When the PXDAC4800 device handle is obtained, no utility DMA buffers are allocated. Utility DMA buffers are not copied for duplicated handles (DuplicateHandleXD48).

This function is used to ensure that the specified application-specific utility DMA buffer is at least as large as the given sample count. In the event that the current buffer is too small, it will be freed and a new buffer will be allocated. Previous data is not copied.

The address of the DMA buffer is obtained by calling GetUtilityDmaBufferXD48.

The buffer may be freed by calling `FreeUtilityDmaBufferXD48`. Utility buffers are also automatically freed when the associated device handle is closed (`DisconnectFromDeviceXD48`).

Warning: Do not free utility DMA buffers with the `FreeDmaBufferXD48` function. Doing so will free the DMA buffer but not invalidate the internal reference to the buffer held by the library implementation which may result in an access violation when the device handle is closed.

Related Functions

[GetUtilityDmaBufferXD48](#), [FreeUtilityDmaBufferXD48](#)

3.4.7.3 FreeDmaBufferXD48

Prototype

```
int FreeDmaBufferXD48 (HXD48 hBrd, void* bufp);
```

Description

Free a DMA buffer previously allocated by the [AllocateDmaBufferXD48](#) function

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *bufp*

The address of the DMA buffer to free.

Return Value

Returns `SIG_SUCCESS` (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The PXDAC4800 driver will automatically free any DMA buffers that have not been freed by a process when it exits.

Related Functions

[AllocateDmaBufferXD48](#)

3.4.7.4 FreeMemoryXD48

Prototype

```
int FreeMemoryXD48 (void* p);
```

Description

Free memory allocated by the PXDAC4800 library

Parameters

[in] *p*
The address of the PXDAC4800 library-allocated data

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Certain PXDAC4800 library functions allocate memory on behalf of the caller and it is the caller's responsibility to free this memory when they are done using it. Use this function to free the memory. Do not use *free* or *delete* because the PXDAC4800 library may have allocated from another heap and freeing could result in memory corruption or an application crash.

Do not free DMA buffers with this function; use [FreeDmaBufferXD48](#)

3.4.7.5 FreeUtilityDmaBufferXD48

Prototype

```
int FreeUtilityDmaBufferXD48 (HXD48 hBrd, int buf_idx);
```

Description

Frees the utility buffer associated with the given PXDAC4800 handle

Parameters

[in] *hBrd*
A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *buf_idx*
The index of the utility DMA buffer to free [0,1] or -1 to delete all utility DMA buffers.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to free the utility DMA buffer with the given index. The library will also automatically free utility DMA buffers when the associated handle is closed.

Utility DMA buffers are allocated by calling [EnsureUtilityDmaBufferXD48](#).

Warning: Do not free utility DMA buffers with the [FreeDmaBufferXD48](#) function. Doing so will free the DMA buffer but not invalidate the internal reference to the buffer held by the library implementation which may result in an access violation when the device handle is closed.

Related Functions

[EnsureUtilityDmaBufferXD48](#), [GetUtilityDmaBufferXD48](#)

3.4.7.6 GetUtilityDmaBufferXD48

Prototype

```
int GetUtilityDmaBufferXD48 (HXD48 hBrd, int buf_idx, void** bufpp, unsigned int* buf_bytesp);
```

Description

Get the library managed utility DMA buffer

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *buf_idx*

The index of the utility DMA buffer to obtain the address of.

[out] *bufpp*

A pointer to a void* variable that will receive the address of the specified utility DMA buffer, or NULL if no utility DMA buffer has been allocated. This parameter may be NULL.

[out] *buf_bytesp*

A pointer to an integer variable that will receive the length of the specified utility DMA buffer in bytes, or zero if no utility DMA buffer has been allocated. This parameter may be NULL.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Utility DMA buffers are allocated by calling [EnsureUtilityDmaBufferXD48](#).

Related Functions

[EnsureUtilityDmaBufferXD48](#), [FreeUtilityDmaBufferXD48](#)

3.4.8 Data Playback Routines

The functions in this section are used to perform data playback operations.

3.4.8.1 BeginRamPlaybackXD48

```
int BeginRamPlaybackXD48 (  
    HXD48 hBrd,  
    unsigned int ram_offset_bytes,  
    unsigned int ram_length_bytes,
```



```
unsigned int playback_bytes = 0);
```

Description

Begin RAM playback from the given region in PXDAC4800 RAM

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *ram_offset_bytes*

The byte offset into PXDAC4800 RAM at which to begin playing back data. This parameter must be an integer multiple of 8192 bytes. See Remarks

[in] *ram_length_bytes*

The total number of bytes of data to use for play back data. This value is independent of channel count: If the card is configured for dual channel playback then the card will play back (length_bytes / 2) bytes per channel. If the card is configured for four channel playback then the card will play back (length_bytes / 4) bytes per channel. See Remarks

[in] *playback_bytes*

The total number of bytes that the PXDAC4800 will play back per trigger. This should be an integer multiple of (8 * number of active channels) samples and will be aligned down if necessary. This parameter is ignored if the Trigger Mode setting is configured for Continuous playback. If this parameter is zero then the length_bytes parameter will define the playback length. If this parameter is greater than ram_length_bytes then the PXDAC4800 will loop back around to the start of the playback data. See Remarks

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is the primary method for starting a RAM playback operation. A RAM playback operation is a playback operation in which all playback data has been previously loaded into PXDAC4800 RAM. The PXDAC4800 can then play (or loop on) this data at rates up to the maximum playback rate which is 1.2GHz.

Calling this function will arm the PXDAC4800 for RAM playback and return immediately. This function will not wait for the playback to complete. Actual hardware playback will not begin until a trigger event is detected by the PXDAC4800.

A RAM playback operation is stopped by calling EndRamPlaybackXD48. Putting the board into the Standby operating mode or calling the SetPowerupDefaultsXD48 will also implicitly stop a RAM playback operation.

The LoadRamBufXD48 function is the primary means by which data is loaded into PXDAC4800 RAM. The LoadFileIntoRamXD48 function will synchronously load a file into PXDAC4800 RAM.

Note that this function deals with playback data in terms of bytes. If you are using one of the 14-bit sample formats, each data sample will be two bytes.

The `ram_offset_bytes` and `ram_length_bytes` parameters define the segment of PXDAC4800 memory that will be used as the source for playback data. Depending on the `playback_samples` parameter or the Trigger Mode setting, the PXDAC4800 may try to playback more than `ram_length_bytes` bytes. In this case, the PXDAC4800 will loop around back to the start of the playback data as necessary.

If the current Trigger Mode setting is configured for Continuous then the `playback_bytes` parameter is ignored and the PXDAC4800 will loop around the specified RAM segment indefinitely after it receives a single trigger.

If the current Trigger Mode setting is configured for Play-Per-Trigger then the PXDAC4800 will playback `playback_bytes` bytes of data and then stop playback and rearm for another trigger. When another trigger is received the PXDAC4800 will playback another `playback_bytes` bytes of data, resuming where it stopped in the RAM segment, looping as necessary. This process is repeated until the RAM playback is stopped.

If the current Trigger Mode setting is configured for Single Shot, then the PXDAC4800 will playback `playback_bytes` bytes of data and then stop playback altogether, ignoring any further trigger events. In order to resume playback, the RAM playback will be need to be stopped and rearmed.

Related Functions

[EndRamPlaybackXD48](#)

3.4.8.2 BeginStreamingPlaybackXD48

```
int BeginStreamingPlaybackXD48 (HXD48 hBrd);
```

Description

Begin a local streaming playback operation

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns `SIG_SUCCESS` (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function will place the PXDAC4800 into Streaming Playback mode. In this mode of playback, the software streams playback data from the host PC to the PXDAC4800 by iteratively calling [LoadStreamDataFastXD48](#) to transfer data to the PXDAC4800. Streaming playback is ended by calling `EndStreamingPlaybackXD48`.

All Streaming Playback operations run in a free-run mode in which the length of the playback is assumed to infinite. When the desired amount of playback data has been transferred to the PXDAC4800, the `NotifyAllStreamDataUploadedXD48` function can be called to notify the PXDAC4800 that all playback data has been uploaded and that it should play whatever is left in its FIFOs and not expect more data. The `WaitForSamplesCompleteXD48` function can then be used to wait for all remaining data to be played back.

The following pseudo-code demonstrates a streaming playback operation.

```

BeginStreamingPlaybackXD48(hBrd);
while (...)
{
    ReadFile (source_file, bufp, buf_bytes, ...);
    LoadStreamDataFastXD48(hBrd, bufp, buf_bytes);
    if (IsLastTransfer)
        NotifyAllStreamDataUploadedXD48(hBrd);
}
WaitForSamplesCompleteXD48(hBrd);
EndStreamingPlaybackXD48(hBrd);

```

At the time of this writing, Streaming Playback mode requires that the trigger mode selection be set to Continuous trigger mode. The software will return an error if an invalid trigger mode has been selected.

If external trigger is being used it is highly recommended that it be disabled in the software (SetExternalTriggerEnableXD48) before calling this function. The software can re-enable the external trigger after enough playback data has been loaded and the DAC output FIFOs are full (GetDacOutputFifoFullFlagXD48). If the external trigger is not disabled, a trigger may come in before enough data has been moved through the data pipeline resulting in gaps in playback.

Remote devices: This function is not available for remote PXDAC4800 devices. The Streaming Playback Session interface must be used to streaming playback operations on remote devices.

Related Functions

[LoadStreamDataFastXD48](#), [NotifyAllStreamDataUploadedXD48](#), [EndStreamingPlaybackXD48](#)

3.4.8.3 EndRamPlaybackXD48

Prototype

```
int EndRamPlaybackXD48 (HXD48 hBrd);
```

Description

End the current RAM playback operation

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to end the current PXDAC4800 RAM playback operation. Calling this function will result in the PXDAC4800 being put into Standby mode and DAC outputs will be set to idle setting.

Related Functions

[BeginRamPlaybackXD48](#)

3.4.8.4 EndStreamingPlaybackXD48

```
int EndStreamingPlaybackXD48 (HXD48 hBrd);
```

Description

End a local streaming playback operation

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to end a streaming playback operation previously started with the BeginStreamingPlaybackXD48 function. Calling this function will result in the board being put into Standby mode and all DAC outputs set to idle setting.

Related Functions

[BeginStreamingPlaybackXD48](#)

3.4.8.5 IsPlaybackInProgressXD48

Prototype

```
int IsPlaybackInProgressXD48 (HXD48 hBrd);
```

Description

Determine if a playback is in progress

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns 1 if a playback is in progress, 0 if no playbacks are in progress, or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function bypasses the handle-specific cached hardware settings and queries the device driver directly.

Related Functions

3.4.9 Data Transfer Routines

The functions in this section are used to perform data transfers

The transfer of acquisition data between the PXDAC4800 and local host memory are implemented using Direct Memory Access (DMA) transfers. These are high speed data transfers in which data is transferred independently of local system processors. The PXDAC4800 library, driver, and hardware manage all aspects of DMA transfers.

Before a DMA transfer may be performed a DMA buffer needs to be allocated. A DMA buffer is a contiguous, non-paged region of memory. A DMA buffer is used just like normal memory, the only difference being that it is allocated and freed by calling PXDAC4800 library functions instead of malloc/free or new/delete. The [AllocateDmaBufferXD48](#) function is used to allocate a DMA buffer and map it into the calling process' address space. The [FreeDmaBufferXD48](#) function is used to free a DMA buffer when it is no longer needed by the application.

Once a DMA buffer has been allocated the PXDAC4800 may transfer data to or from it via DMA transfer using one of the data transfer routines:

Data Transfer Routine	Notes
LoadRamBufXD48	Transfer data from PC to PXDAC4800 for RAM playback using buffered transfers which do not require a DMA buffer.
LoadStreamDataFastXD48	Transfer data from PC to PXDAC4800 for streaming playback.

By default, all DMA transfer operations are synchronous. This means that when the data transfer routine returns the data transfer is complete and all data has been transferred. Some of the data transfer functions can be setup to do an asynchronous DMA transfer. In an asynchronous DMA transfer, the transfer is setup and initiated but the function returns immediately without waiting for all data to be transferred, which allows the calling thread to continue working. The [WaitForTransferCompleteXD48](#) function can then be used to wait for the transfer to complete.

3.4.9.1 LoadFileIntoRamXD48

Prototype

```
int LoadFileIntoRamXD48 (
    HXD48          hBrd,
    unsigned int    dst_offset_bytes,
    unsigned int    dst_length_bytes,
    const TCHAR*    srcp,
    unsigned long long src_offset_bytes = 0,
    unsigned int     src_length_bytes  = 0,
    int             flags               = 0);
```

Description

Synchronously load a file into PXDAC4800 RAM

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *dst_offset_bytes*

The offset, in bytes, into PXDAC4800 RAM at which to begin loading playback data. This parameter must be an integer multiple of 8192.

[in] *dst_length_bytes*

The number of bytes to load from the file. This parameter must be an integer multiple of 8192 and will be aligned down if necessary. If this parameter is zero, then the amount of data to be loaded will be derived from the source length.

[in] *srcp*

A pointer to a NULL-terminated string containing the pathname of the file to load.

[in] *src_offset_bytes*

The offset, in bytes, into the source file at which to begin copying.

[in] *src_length_bytes*

The number of bytes to copy from the file. If this parameter is zero then the file size (minus *src_offset_bytes*) will be used.

[in] *flags*

A set of flags (XD48LFF_*) that define function behavior:

Library Symbolic Constant	Interpretation
XD48LFF_NO_UNBUFFERED_FILE_IO (0x00000001)	Do not try to use faster, unbuffered IO

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The source length should always be greater than the destination length.

This function will synchronously load the given file into PXDAC4800 RAM for a future RAM playback operation. The function will not return until all data has been loaded.

LoadFileIntoRamXD48 is a macro that will expand to _LoadFileIntoRamAXD48 or _LoadFileIntoRamWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

3.4.9.2 LoadRamBufXD48

Prototype

```
int LoadRamBufXD48 (  
    HXD48 hBrd,
```

```

unsigned int offset_bytes,
unsigned int length_bytes,
const void* bufp,
int bAsynchronous = 0);

```

Description

Load PXDAC4800 RAM with playback data for RAM playback using buffered transfers

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *offset_bytes*

The byte offset into PXDAC4800 RAM at which to begin loading data. If *bAsynchronous* is nonzero then this parameter must be an integer multiple of 8192 bytes. If *bAsynchronous* is zero then this parameter need not be on any particular boundary.

[in] *length_bytes*

The total number of bytes to load into PXDAC4800 RAM. If *bAsynchronous* is nonzero then this parameter must be an integer multiple of 8192 bytes. If *bAsynchronous* is zero then this parameter need not be on any particular boundary.

[in] *bufp*

A pointer to a buffer that will be used for the data transfer. This buffer must be at least *length_bytes* bytes. This buffer need not be a DMA buffer, but a DMA buffer is okay to use with this function.

[in] *bAsynchronous*

If this parameter is nonzero then an asynchronous data transfer will be performed. See remarks.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Synchronous transfers: This function will return SIG_CANCELLED (-10) if the data transfer is cancelled before it finishes. A data transfer is cancelled by putting the board into Standby mode, usually from another thread (or process).

Remarks

This function is only used to load data that will be played back in a RAM Playback operation. For Streaming Playback operations, the LoadStreamDataFastXD48 function is used while in streaming playback mode.

By default, this function will not return until all of the requested samples have been transferred. This function can also start an asynchronous transfer. By default, this function will not return until the data transfer completes. If the *bAsynchronous* parameter is nonzero, then the function will start the data transfer and return immediately without waiting for it to finish allowing the calling thread to continue working. The caller can use the IsTransferInProgressXD48 function to determine if the transfer is still in progress. To wait (sleep) for the transfer to complete, call the WaitForTransferCompleteXD48 function.

3.4.9.3 NotifyAllStreamDataUploadedXD48

```
int NotifyAllStreamDataUploadedXD48 (HXD48 hBrd);
```

Description

Notifies the PXDAC4800 that all streaming playback data has been uploaded

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is only used for streaming playback operations. This function should not be used for RAM playback or RAM load operations.

When the PXDAC4800 is in streaming playback mode the PXDAC4800 will playback an infinite amount of data. As long as the PC continues to stream data to the card, it will continue to play back the data. For instances in which a concrete amount of data is to be streamed, this function is used to tell the PXDAC4800 that all playback data has been uploaded and that it should not expect any more. After calling this function, the PXDAC4800 will play back all currently buffered data until the internal RAM FIFO goes empty at which point it will send a Samples Complete interrupt to the host system. The PC software can use this event as an indication that the streaming playback has completed and that it is safe to end the streaming playback (EndStreamingPlaybackXD48).

This function should be called after the very last data transfer (LoadStreamDataFastXD48) has completed. If asynchronous DMA transfers are being used then this function should be called after the last asynchronous data transfer has completed (WaitForTransferCompleteXD48).

Related Functions

[BeginStreamingPlaybackXD48](#), [WaitForSamplesCompleteXD48](#)

3.4.9.4 IsTransferInProgressXD48

Prototype

```
int IsTransferInProgressXD48 (HXD48 hBrd);
```

Description

Determine if an asynchronous data transfer is currently in progress

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

Return Value

Returns 1 if a transfer is currently in progress, 0 if a transfer is not in progress, or one of the [library error codes](#) (which are all negative) on error. See remarks.

Remarks

This function is used to determine if a data transfer to or from the PXDAC4800 is currently in progress and is only relevant when doing asynchronous DMA transfer operations.

A return value 1 should be interpreted as: A DMA transfer operation has been started, but the software has not been notified by the hardware that the operation has completed.

A return value 0 should be interpreted as: The PXDAC4800 driver is not currently expecting a DMA complete notification from the hardware. This may be because the operation has already finished or because it was never attempted.

This function will always return 0 for virtual devices.

Related Functions

[WaitForTransferCompleteXD48](#)

3.4.9.5 WaitForTransferCompleteXD48

Prototype

```
int WaitForTransferCompleteXD48 (HXD48 hBrd, unsigned int timeout_ms = 0);
```

Description

Wait for a DMA transfer operation to complete with optional timeout

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *timeout_ms*

An optional timeout value in milliseconds. If this value is 0 then the function will not timeout. In the event of a timeout the function will return SIG_XD48_TIMED_OUT.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error; see remarks.

This function will return SIG_XD48_TIMED_OUT (-541) if the timeout elapses before the acquisition operation completes.

This function will return SIG_CANCELLED (-10) if the operation is cancelled before it finishes. An operation may be cancelled by putting the PXDAC4800 into the Standby operating mode.

Remarks

This function is used when doing asynchronous DMA transfer operations: LoadStreamDataFastXD48 in which the bAsynchronous parameter is non-zero.

Calling this function while a DMA transfer is in progress (see [IsTransferInProgressXD48](#)) will result in the current thread being blocked until the operation finishes, times out, or is cancelled by another thread putting the board into Standby mode.

IMPORTANT: If the asynchronous transfer completes (SIG_SUCCESS) or times out (SIG_XD48_TIMED_OUT) then the current operating mode will not be changed. In all other cases, this function will put the board back into Standby mode before returning. The operating mode is not changed when the transfer completes because more data transfers may be forthcoming (e.g. in a streaming playback mode) which will require the current operating mode. The operating mode is not changed when a transfer times out because the software could potentially want to come back and wait again. This means that the programmer is responsible for putting the board back into Standby mode in either of these cases.

RAM-Buffered PCIe acquisition mode: When the underlying transfer completes the software will check the status of the onboard RAM FIFO underflow flag. If this flag is set then the PXDAC4800 RAM-FIFO went empty at some point prior to or during the transfer. This means that DAC output was held up for an indeterminate amount of time. This will happen if data cannot be transferred to the card faster than it is playing back the data. If the flag is set, the function will return SIG_XD48_FIFO_UNDERFLOW.

Related Functions

[IsTransferInProgressXD48](#)

3.4.10 Data Manipulation Routines

3.4.10.1 DeInterleaveDataXbitYChanXD48

Prototype

<code>int DeInterleaveData16bit2ChanXD48 (</code>
<code> const unsigned short* srcp,</code>
<code> unsigned int samples_in,</code>
<code> unsigned short* dst_ch1p,</code>
<code> unsigned short* dst_ch2p);</code>
<code>int DeInterleaveData16bit4ChanXD48 (</code>
<code> const unsigned short* srcp,</code>
<code> unsigned int samples_in,</code>
<code> unsigned short* dst_ch1p,</code>
<code> unsigned short* dst_ch2p,</code>
<code> unsigned short* dst_ch3p,</code>
<code> unsigned short* dst_ch4p);</code>
<code>int DeInterleaveData8bit2ChanXD48 (</code>
<code> const unsigned char* srcp,</code>

<code>unsigned int</code>	<code>samples_in,</code>
<code>unsigned char*</code>	<code>dst_ch1p,</code>
<code>unsigned char*</code>	<code>dst_ch2p);</code>
<code>int DeInterleaveData8bit4ChanXD48 (</code>	
<code>const unsigned char*</code>	<code>srcp,</code>
<code>unsigned int</code>	<code>samples_in,</code>
<code>unsigned char*</code>	<code>dst_ch1p,</code>
<code>unsigned char*</code>	<code>dst_ch2p,</code>
<code>unsigned char*</code>	<code>dst_ch3p,</code>
<code>unsigned char*</code>	<code>dst_ch4p);</code>

Description

De-interleave multichannel data into separate buffers.

Parameters

[in] *srcp*

A pointer to a buffer containing interleaved dual-channel data

[in] *samples_in*

The total number of samples contained in the buffer pointed to by *srcp*

[out] *dst_ch1p*

A pointer to a buffer that will receive channel 1 data. This parameter may be NULL if channel 1 data isn't needed.

[out] *dst_ch2p*

A pointer to a buffer that will receive channel 2 data. This parameter may be NULL if channel 2 data isn't needed.

[out] *dst_ch3p*

A pointer to a buffer that will receive channel 3 data. This parameter may be NULL if channel 3 data isn't needed.

[out] *dst_ch4p*

A pointer to a buffer that will receive channel 4 data. This parameter may be NULL if channel 4 data isn't needed.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Multichannel PXDAC4800 playback data is sample interleaved: Channel 1 Sample 1, Channel 2 Sample 1, Channel 1 Sample 2, Channel 2 Sample 2, ... Use this function to extract out one or more channels of multichannel data into separate buffers.

Related Functions

[InterleaveDataXbitYChanXD48](#)

3.4.10.2 InterleaveDataXbitYChanXD48

Prototype

<code>int</code>	<code>InterleaveData16bit2ChanXD48</code>	<code>(</code>
	<code>const unsigned short*</code>	<code>src_ch1p,</code>
	<code>const unsigned short*</code>	<code>src_ch2p,</code>
	<code>unsigned int</code>	<code>samps_per_chan,</code>
	<code>unsigned short*</code>	<code>dstp);</code>
<code>int</code>	<code>InterleaveData16bit4ChanXD48</code>	<code>(</code>
	<code>const unsigned short*</code>	<code>src_ch1p,</code>
	<code>const unsigned short*</code>	<code>src_ch2p,</code>
	<code>const unsigned short*</code>	<code>src_ch3p,</code>
	<code>const unsigned short*</code>	<code>src_ch4p,</code>
	<code>unsigned int</code>	<code>samps_per_chan,</code>
	<code>unsigned short*</code>	<code>dstp);</code>
<code>int</code>	<code>InterleaveData8bit2ChanXD48</code>	<code>(</code>
	<code>const unsigned char*</code>	<code>src_ch1p,</code>
	<code>const unsigned char*</code>	<code>src_ch2p,</code>
	<code>unsigned int</code>	<code>samps_per_chan,</code>
	<code>unsigned char*</code>	<code>dstp);</code>
<code>int</code>	<code>InterleaveData8bit4ChanXD48</code>	<code>(</code>
	<code>const unsigned char*</code>	<code>src_ch1p,</code>
	<code>const unsigned char*</code>	<code>src_ch2p,</code>
	<code>const unsigned char*</code>	<code>src_ch3p,</code>
	<code>const unsigned short*</code>	<code>src_ch4p,</code>
	<code>unsigned int</code>	<code>samps_per_chan,</code>
	<code>unsigned char*</code>	<code>dstp);</code>

Description

Interleave multiple channels of data into a single interleaved buffer

Parameters

[in] *src_ch1p*

A pointer to a buffer that contains channel 1 data. If this parameter is NULL then no data is copied over the channel 1 data samples in the destination buffer

[in] *src_ch2p*

A pointer to a buffer that contains channel 2 data. If this parameter is NULL then no data is copied over the channel 2 data samples in the destination buffer

[in] *src_ch3p*

A pointer to a buffer that contains channel 3 data. If this parameter is NULL then no data is copied over the channel 3 data samples in the destination buffer

[in] *src_ch4p*

A pointer to a buffer that contains channel 4 data. If this parameter is NULL then no data is copied over the channel 4 data samples in the destination buffer

[in] *samps_per_chan*

The number of samples contained in each of the input channel data buffers

[out] *dstp*

A pointer to the buffer that will receive the interleaved data. This buffer must be at least (2 * *samps_per_chan*) samples in size

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function may be used to (re)create interleaved multichannel data

Related Functions

[DeInterleaveDataXbitYChanXD48](#)

3.4.11 PXDAC4800 Streaming Playback Session Routines

The functions in this group define a higher-level API that may be used to do a library managed streaming playback operation. This interface allows the programmer to specify a playback data source (e.g. one or more files) that the PXDAC4800 library will play back. With this API, the library handles all the details of the playback: file and buffer management, active memory configuration, operating mode changes, etc.

A Streaming Playback session runs in a background thread allowing the primary thread to continue working while the streaming playback is in progress. This allows for easier integration into other environments such as LabVIEW, Matlab, or .NET languages that cannot access arbitrary unmanaged memory.

Functions exist for querying the status/progress of the playback as well as obtain snapshots of the data currently being loaded.

This interface is available for remote PXDAC4800 devices.

3.4.11.1 SessionStreamCreateXD48

Prototype

```
int SessionStreamCreateXD48 (
    HXD48                hBrd,
    const TCHAR*          data_srcp,
    XD48S\_STREAM\_SES\_CREATE* ses_createp,
    HXD48STREAM*          handlep);
```

Description

Create a Streaming Playback session instance

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *data_srcp*

A pointer to a NULL terminated string containing the pathname of the file containing the playback data. If this parameter is NULL then no playback data will be loaded during the streaming playback and the library will just loop around transferring the data already in the transfer buffers.

[in] *ses_createp*

A pointer to a [XD48S_STREAM_SES_CREATE](#) structure that define the parameters of the streaming playback session. This structure and its members are detailed in the Structure XD48S_STREAM_SES_CREATE section.

[out] *handlep*

A pointer to a HXD48STREAM variable that will receive a handle that identifies the streaming playback session.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function is used to create a streaming playback session instance. A streaming playback session is a library-managed streaming playback in which data is streamed to the PXDAC4800 from a file on the host system. The library manages all aspects of the playback and runs in a background thread. Routines exist to check on streaming playback progress as well as obtain data snapshots of the current input playback data.

On success, the streaming playback will be in progress in a background thread. For external trigger users, it is recommended to disable the external trigger before creating the streaming playback. This will allow the card to buffer enough data to fill the internal data pipeline such that when the external trigger is re-enabled at some point after creating the streaming playback that data will be allowed to play back uninterrupted.

During the streaming playback, SessionStreamProgressXD48 can be periodically called to obtain the current status/progress of the streaming playback session.

A streaming playback session can be stopped at any time by calling SessionStreamEndXD48.

A streaming playback instance is deleted by calling SessionStreamDeleteXD48. This should be called at some point after the streaming playback has completed. Failure to delete the streaming playback instance will result in memory/resource leaks for the current process.

SessionStreamCreateXD48 is a macro that will expand to _SessionStreamCreateAXD48 or _SessionStreamCreateWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Related Functions

[SessionStreamDeleteXD48](#)

3.4.11.2 SessionStreamCreateParmsXD48

Prototype

```
int SessionStreamCreateParmsXD48 (
    HXD48                hBrd,
    const TCHAR*          data_srcp,
```

```

int          stream_flags,
double       stream_bytes,
double       stream_samples,
double       src_offset_bytes,
double       src_length_bytes,
unsigned int xfer_bytes,
unsigned int ss_len_bytes,
unsigned int ss_period_xfer,
unsigned int ss_period_ms,
unsigned int* sessionp);

```

Description

Create a Streaming Playback session instance without the use of the [XD48S_STREAM_SES_CREATE](#) structure. This function can be called within Labview or Matlab. The parameters are the same as the [XD48S_STREAM_SES_CREATE](#) structure, but they are set individually.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *data_srcp*

A pointer to a NULL terminated string containing the pathname of the file containing the playback data. If this parameter is NULL then no playback data will be loaded during the streaming playback and the library will just loop around transferring the data already in the transfer buffers.

[in] *stream_flags*

A set of flags that control streaming playback session behavior:

Symbolic Constant	Value	Interpretation
XD48STREAMF_FORCE_SW_TRIGGER	0x00000001	Force playback to begin without waiting for an external trigger
XD48STREAMF_DO_SNAPSHOTS	0x00000002	Periodically grab snapshots of playback data. (See ss_* fields below)
XD48STREAMF_NO_UNBUFFERED_IO	0x00000004	Do not try to use unbuffered IO
XD48STREAMF_INIT_STATIC_DATA	0x80000000	Pre-initialize input buffers with static playback data

[in] *stream_bytes*

The total number of bytes to playback or one of XD48STREAMLEN_* values defined below. This parameter will be aligned up to the next DMA transfer boundary (128 bytes).

Symbolic Constant	Value	Interpretation
XD48STREAMLEN_INFINITE	0	Stream infinitely, looping around source data as necessary.
XD48STREAMLEN_SOURCE_LENGTH	0xFFFFFFFFFFFFFFFF	Defines stream size to be length of source data (minus source offset).

[in] *src_offset_bytes*

This parameter specifies the byte offset into the source data at which to begin uploading playback data.

[in]*src_length_bytes*

The number of bytes of the source data to upload. If this parameter is zero then all bytes of the data source (minus any bytes skipped by *src_offset_bytes*) will be used.

[in]*xfer_bytes*

The data transfer size to use when uploading playback data. It is recommended to pass zero for this parameter, which lets the library decide the best transfer size.

[in]*ss_len_bytes*

This parameter defines the desired length of the playback data snapshot that will be periodically grabbed by the streaming playback. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

[in]*ss_period_xfer*

The playback data snapshot period, in DMA transfers. For long, non-segmented streaming playbacks, a snapshot period in DMA transfers can be used to obtain synchronized playback snapshots over multiple independently running playback sessions. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

[in]*ss_period_ms*

The playback data snapshot update period, in milliseconds. If the *ss_period_xfer* field is zero, the library will default back to a simple, approximate-timed approach for playback data snapshot updates. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

[out] *sessionp*

A pointer to a HXD48STREAM variable that will receive a handle that identifies the streaming playback session.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

Same as for SessionStreamCreateXD48. Use this function in combination with the following functions:

- SessionStreamGetProgressXD48
- SessionStreamEndNoStructXD48
- SessionStreamDeleteNoStructXD48

SessionStreamCreateParmsXD48 is a macro that will expand to _SessionStreamCreateParmsAXD48 or _SessionStreamCreateParmsWXD48 depending on the native character type. See [Library Functions That Use Character Strings](#) section for details.

Related Functions

[SessionStreamDeleteXD48](#)

3.4.11.3 _SessionStreamCreateStdXD48

Prototype


```

int SessionStreamCreateStdXD48 (
    HXD48          hBrd,
    const wchar_t* data_srcp,
    int           SampleSize,
    int           SessionType,
    unsigned int  SnapShotLen_Bytes,
    unsigned int  SnapShotPeriod_ms,
    HXD48STREAM\*  handlep);

```

Description

Creates a streaming session using standard configuration parameters. This function can be used the same way as the SessionStreamCreateXD48 function, but without the need of configuring the [XD48S_STREAM_SES_CREATE](#) structure. The streaming session parameters will be set to standard values depending on the SessionType.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *data_srcp*

A pointer to a string containing the path to the data file to be streamed. The string can be in [char](#) or [wchar_t](#) depending if Unicode is used or not.

[in] *SampleSize*

The size of the samples contained in the data file.

[in] *SessionType*

A constant defining the type of streaming session to be used:

Library Symbolic Constant		Interpretation
XD48_STRMSESS_ONESHOT	(0)	One shot streaming playback
XD48_STRMSESS_INFINITE	(1)	Infinite streaming playback
XD48_STRMSESS_ONESHOT_SS	(2)	One shot streaming playback with snap shots.
XD48_STRMSESS_INFINITE	(1)	Infinite streaming playback with snap shots.

[in] *SnapShotLen_Bytes*

The length, in bytes, of the snap shots to be periodically taken. This parameter is ignored if the SessionType does not mention to use snap shots.

[in] *SnapShotPeriod_ms*

The period of time, in milliseconds, between each snap shots. This parameter is ignored if the SessionType does not mention to use snap shots.

[out] *sessionp*

A pointer to a HXD48STREAM variable (unsigned in type) that will receive a handle that identifies the streaming playback session.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

This function works the same way as the [SessionStreamCreateXD48](#) function. It is particularly useful when the original function cannot be used because of the [XD48S_STREAM_SES_CREATE](#) structure cannot be handled by the programming language (ex: Labview). The behavior of the streaming session will be the exact same as if created by the original function.

When using this function, it is not necessary to call the [_SetStreamingLengthXD48](#) function. This function will be called automatically.

Related Functions

[SessionStreamDeleteXD48](#)

3.4.11.4 SessionStreamDeleteXD48

Prototype

```
int SessionStreamDeleteXD48 (HXD48STREAM hSes);
```

Description

Delete a PXDAC4800 recording session

Parameters

[in] *hSes*

A handle to a PXDAC4800 streaming playback session. This handle is obtained by calling the [SessionStreamCreateXD48](#) function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

If a streaming playback is currently in progress when this function is called, it will automatically be stopped via a call to [SessionStreamEndXD48](#).

After calling this function the given streaming playback session handle ceases to be valid.

Related Functions

[SessionStreamCreateXD48](#)

3.4.11.5 SessionStreamDeleteNoStructXD48

Prototype

```
int SessionStreamDeleteXD48 (unsigned int * hSes);
```

Description

Delete a PXDAC4800 recording session without the use of the [HXD48STREAM](#) handle.

Parameters

[in] *hSes*

A pointer to an unsigned int variable containing a handle to a PXDAC4800 streaming playback session. This handle is obtained by calling the SessionStreamCreateParmsXD48 function.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

If a streaming playback is currently in progress when this function is called, it will automatically be stopped via a call to SessionStreamEndNoStructXD48.

After calling this function the given streaming playback session handle ceases to be valid.

Related Functions

[SessionStreamCreateParmsXD48](#)

3.4.11.6 SessionStreamEndXD48

Prototype

```
int SessionStreamEndXD48 (HXD48STREAM hSes, int bDelete);
```

Description

End a Streaming Playback session

Parameters

[in] *hSes*

A handle to a PXDAC4800 streaming playback session. This handle is obtained by calling the SessionStreamCreateXD48 function.

[in] *bDelete*

If this parameter is non-zero then the library will delete the streaming playback session instance by calling SessionStreamDeleteXD48. If this parameter is zero, then the caller is responsible for freeing the session instance.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Related Functions

[SessionStreamCreateXD48](#)

3.4.11.7 SessionStreamEndNoStructXD48

Prototype

```
int SessionStreamEndXD48 (unsigned int * hSes, int bDelete);
```

Description

End a Streaming Playback session without the use of the [HxD48Stream](#) handle.

Parameters

[in] *hSes*

A pointer to an unsigned int variable containing a handle to a PxDAC4800 streaming playback session. This handle is obtained by calling the [SessionStreamCreateParmsXD48](#) function.

[in] *bDelete*

If this parameter is non-zero then the library will delete the streaming playback session instance by calling [SessionStreamDeleteXD48](#). If this parameter is zero, then the caller is responsible for freeing the session instance.

Return Value

Returns [SIG_SUCCESS](#) (0) on success or one of the [library error codes](#) (which are all negative) on error.

Related Functions

[SessionStreamCreateParmsXD48](#)

3.4.11.8 SessionStreamProgressXD48

Prototype

```
int SessionStreamProgressXD48 (HxD48Stream hSes, XD48S\_STREAM\_SES\_PROG* prog, int flags = 0);
```

Description

Obtain Streaming Playback session progress/status

Parameters

[in] *hSes*

A handle to a PxDAC4800 streaming playback session. This handle is obtained by calling the [SessionStreamCreateXD48](#) function.

[out] *prog*

A pointer to a XD48S_STREAM_SES_PROG structure that will receive the current progress/status of the PXDAC4800 streaming playback session. The caller must initialize the struct_size field of this structure before calling this function.

[in] *flags*

A set of flags (XD48SPBPROGF_*) that control function behavior:

Library Symbolic Constant	Interpretation
XD48SPBPROGF_NO_ERROR_TEXT (0x00000001)	Do not generate error text on error
XD48SPBPROGF_NO_FILE_TEXT (0x00000002)	Do not generate file pathname text

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

It is okay to call this function after a streaming playback has been stopped.

Related Functions

[SessionStreamCreateXD48](#), [SessionStreamSnapshotXD48](#)

3.4.11.9 SessionStreamGetProgressXD48

Prototype

```
int SessionStreamProgressXD48 ( unsigned int* hSes,
                                int* StreamStatus,
                                double* bytes_uploaded,
                                double* bytes_to_upload,
                                unsigned int* elapsed_time_ms,
                                unsigned int* xfer_bytes,
                                unsigned int* xfer_count,
                                unsigned int* snapshot_count,
                                int* progress_flags,
                                int* err_res,
                                char* err_textp,
                                int flags_XD48_DEF(XD48SPBPROGF__DEFAULT));
```

Description

Obtain Streaming Playback session progress/status without the use of the [XD48S_STREAM_SES_PROG](#) structure for Matlab and Labview.

Parameters

[in] *hSes*

A handle to a PXDAC4800 streaming playback session. This handle is obtained by calling the SessionStreamCreateXD48 function.

[in] *stream_status*

The current status of the streaming playback session. This can be one of the following values:

Library Constant	Interpretation
XD48SPBSTATUS_IDLE (0)	Idle; playback session not yet started
XD48SPBSTATUS_ACTIVE (1)	Active; Playback session is in progress. Hardware is armed for playback or playing back.
XD48SPBSTATUS_COMPLETE (2)	Complete; Playback session complete or stopped by user
XD48SPBSTATUS_ERROR (3)	Error; Playback session has ended due to error

[in] *bytes_uploaded*

A pointer to a double variable to get the total number of bytes uploaded to the card so far.

[in] *bytes_to_upload*

A pointer to a double variable to get the total number of bytes to upload and playback for the session. If this parameter is zero then the session is configured for an infinite playback.

[in] *elapsed_time_ms*

A pointer to an unsigned int variable to get the total milliseconds that have elapsed since the streaming playback session was started. Note that this is not an exact measure of how long the hardware has been playing back data. The software has no knowledge of when the hardware is actually playing back data or when the hardware is armed and waiting for a trigger.

[in] *xfer_bytes*

A pointer to an unsigned int variable to get the data transfer size used by the recording session. This is for information purposes only.

[in] *xfer_count*

A pointer to an unsigned int variable to get the current transfer counter. Each time a data transfer of playback data completes the transfer counter is incremented.

[in] *snapshot_count*

A pointer to an unsigned int variable to get the current data snapshot counter value. Each time the streaming playback session collects a new playback data snapshot the snapshot counter is incremented. This value is only relevant if the session was configured to collect snapshots. See XD48_STREAM_SES_PROG::stream_flags for more details. Data snapshots are obtained by calling SessionStreamSnapshotXD48.

[in] *progress_flags*

A pointer to an int variable to get a set of flags indicating that certain events have taken place:

Symbolic Constant	Value	Interpretation
XD48SPBPROGF_DAC_FIFOS_FILLED	0x00000001	DAC FIFOs have filled; safe to trigger
XD48SPBPROGF_SW_TRIG_ISSUED	0x00000002	A software trigger has been generated

[in] *err_res*

A pointer to a int variable to get the error code from the streaming session. This field is only relevant if the *stream_status* member is XD48SPBSTATUS_ERROR. This is the [library error code](#) (SIG_*) of the error that caused the recording session to prematurely end.

[in] *err_textp*

A pointer to a char buffer to get the error message. This field is only relevant if the *stream_status* member is XD48SPBSTATUS_ERROR. This is a library-allocated string containing a description of the error that occurred during the recording session. The caller must free the memory for this string when it is no longer

needed by passing the address of the string to FreeMemoryXD48. If the XD48SPBPROGF_NO_ERROR_TEXT flag is passed to [SessionStreamProgressXD48](#) then no error text will be generated and this member will be set to NULL.

[in] *flags*

A set of flags (XD48SPBPROGF_*) that control function behavior:

Library Symbolic Constant	Interpretation
XD48SPBPROGF_NO_ERROR_TEXT (0x00000001)	Do not generate error text on error
XD48SPBPROGF_NO_FILE_TEXT (0x00000002)	Do not generate file pathname text

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

It is okay to call this function after a streaming playback has been stopped.

Related Functions

[SessionStreamCreateXD48](#), [SessionStreamSnapshotXD48](#)

3.4.11.10 SessionStreamSnapshotXD48

Prototype

```
int SessionStreamSnapshotXD48 (  
    HXD48STREAM      hSes,  
    void*            bufp,  
    unsigned int      buf_bytes,  
    unsigned int*     bytes_copiedp = NULL,  
    int*              snapshot_counterp = NULL);
```

Description

Obtain data snapshot from current recording

Parameters

[in] *hSes*

A handle to a PXDAC4800 streaming playback session. This handle is obtained by calling the SessionStreamCreateXD48 function.

[out] *bufp*

A pointer to a buffer that will receive the snapshot of the input playback data

[in] *buf_bytes*

The size, in bytes, of the buffer pointed to by the bufp parameter

[out] *bytes_copiedp*

The address of an unsigned int variable that will receive the number of bytes copied into the snapshot buffer. This parameter may be NULL.

[out] *snapshot_counterp*

The address of an integer variable that will receive the snapshot counter value. Each data snapshot taken by the playback is given a unique counter value. The first snapshot will have a snapshot counter value of 1. This allows client software to differentiate between unique data snapshots. Pass NULL if this information is not needed.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

The playback data snapshot obtained is a snapshot of the data as it is being loaded into the PXDAC4800 and is not in any way synchronized with the output of the PXDAC4800 DACs.

Related Functions

[SessionStreamCreateXD48](#), [SessionStreamProgressXD48](#)

3.4.11.11 _SetStreamingLengthXD48

Prototype

<pre>int _SetStreamingLengthXD48 (HXD48 hBrd, UINT64 length_bytes);</pre>

Description

Sets the number of bytes to be handled during the streaming session, for loading and playback.

Parameters

[in] *hBrd*

A handle to the PXDAC4800 board. This handle is obtained by calling the [ConnectToDeviceXD48](#) function.

[in] *length_bytes*

The number of bytes of data to be handled during the streaming session.

Return Value

Returns SIG_SUCCESS (0) on success or one of the [library error codes](#) (which are all negative) on error.

Remarks

If the streaming session is configured to loop infinitely, set the streaming length to 0xFFFFFFFFFFFFFFFF.

Related Functions

[SessionStreamCreateXD48](#)

3.5 PXDAC4800 Library Data Types

The following sections define some of the data types used by the PXDAC4800 library

3.5.1 Data Type: HXD48

This data type is used to represent a handle to a PXDAC4800 device.

A special library constant XD48_INVALID_HANDLE (NULL) represents an invalid PXDAC4800 device handle. Any other value should be considered opaque; this value only has relevance to the PXDAC4800 library implementation.

PXDAC4800 handles are only valid in the process for which they are obtained.

The DisconnectFromDeviceXD48 library function should be called to release a PXDAC4800 device handle when it is no longer needed. This will free handle-specific resources allocated by the library.

3.5.2 Data Type: HXD48STREAM

This data type is used to represent a handle to [Streaming Playback Session](#) instance.

A special library constant XD48_INVALID_HANDLE (NULL) represents an invalid streaming playback session handle. Any other value should be considered opaque; this value is only has relevance to the PXDAC4800 library implementation.

3.5.3 Structure: XD48S_CYCLE_CALC_CTX

This structure is optionally used by the [CalculateCycleCountsXD48](#) library function and is used provide additional parameters for cycle count calculation.

Structure definition

```
/// Context structure used for CalculateCycleCountsXD48
typedef struct _XD48S_CYCLE_CALC_CTX_tag
{
    unsigned int    struct_size;

    unsigned int    flags;                // XD48CYCLECALCF_*
    unsigned int    max_samples;          // 0,Default: Maximal
    unsigned int    align_override;       // 0,Default: 64 bytes
    double          dDacDataRateMHz;      // 0,Default: Board's current

    // -- Used when finding closest match

    double          dSearchAlignMHz;      // 0, Default: ~100KHz
    double          dSearchDeltaMHz;      // 0, Default: 128KHz
    double          dMaxDeviationMHz;     // 0, Default: No max
    double          dClosestPPC;          // out: Closest points-per-cycle
    double          dClosestMHz;          // out: Output frequency for match
} XD48S_CYCLE_CALC_CTX;
```

Fields

struct_size

This field will be used to discriminate against possible future versions of this structure. This field should be initialized to the structure size (e.g. `sizeof(XD48S_RECORDED_DATA_INFO)`) by the caller.

flags

Flags XD48CYCLECALCF_* that further define function behavior. If closest-match finding is enabled and performed, the library will set the XD48CYCLECALCF_ADJUSTED_PPC_COUNT flag.

Symbolic Constant	Value	Interpretation
<code>XD48CYCLECALCF_FIND_CLOSEST</code>	<code>0x00000100</code>	If given cycle count cannot match, try to find closest that will
<code>XD48CYCLECALCF_FINDHOW__MASK</code>	<code>0x0000000F</code>	Defines bits that govern how to find closest
<code>XD48CYCLECALCF_FINDHOW_CLOSEST</code>	<code>0</code>	Find closest match to requested PPC; higher or lower okay
<code>XD48CYCLECALCF_FINDHOW_LOWER</code>	<code>1</code>	Find closest match to PPC without going over
<code>XD48CYCLECALCF_FINDHOW_HIGHER</code>	<code>2</code>	Find closest match to PPC without going under
<code>XD48CYCLECALCF_ADJUSTED_PPC_COUNT</code>	<code>0x80000000</code>	Function had to adjust PPC count; set by <code>CalculateCycleCountsXD48</code>

max_samples

Defines the maximum number of samples to consider when finding sample count. Passing zero for this member will result in the function using the maximum total sample count for the current active channel selection.

align_override

Defines the required alignment required for the resultant sample count. Passing zero for this member will result in the function using the default PXDAC4800 sample alignment which is 64 bytes (32 16-bit samples or 64 b-bit samples)

dDacDataRateMHz

Defines the DAC input data rate that will be used to playback the signal. The DAC input data rate is the DAC's playback rate divided by the current interpolation factor. Passing zero for this member will result in the function using the given PXDAC4800's current DAC input rate.

dSearchAlignMHz

Defines an alignment for frequencies used in finding the closest match for a points-per-cycle that cannot fit in an acceptable number of samples. Passing zero for this member will result in the function using an alignment near 100KHz. The actual value will be optimized to yield more aligned results.

dSearchDeltaMHz

Defines the delta used when searching for closest match. This value will be incrementally added/subtracted from the realigned (via *dSearchAlignMHz*) output frequency. Passing zero for this member will result in the function using an optimized value near 100KHz.

dMaxDeviationMHz

Defines the maximum frequency deviation (+/-) to allow when finding closest match. Passing zero for this member will allow any valid frequency to be used.

dClosestPPC

Resultant points-per-cycle count. The function will write the resultant closest-match points-per-cycle value to this field.

dClosestMHz

Resultant PXDAC4800 output signal frequency. The function will write the resultant closest-match output frequency value to this member.

3.5.4 Structure: XD48S_DRIVER_STATS

This structure is used by the [GetDriverStatsXD48](#) library function and is used to store the obtained statistics.

Structure definition

```
/// Device driver statistics; see GetDriverStatsXD48
typedef struct _XD48_DRIVER_STATS_tag
{
    unsigned int    struct_size;        // IN: Structure size
    int             nConnections;       // Active connection count
    unsigned int    isr_cnt;            // Total ISR invocations
    unsigned int    dcm_reset_cnt;      // DCM reset operation count

    unsigned int    dmar_finished_cnt;  // DMA transfers finished (reads)
    unsigned int    dmar_started_cnt;   // DMA transfers started (reads)
    unsigned int    dmaw_finished_cnt;  // DMA transfers finished (writes)
    unsigned int    dmaw_started_cnt;   // DMA transfers started (writes)

    unsigned int    pb_started_cnt;     // Playbacks started
    unsigned int    pb_finished_cnt;    // Playbacks completed
    unsigned int    samp_comp_int_cnt;   // Samples Complete interrupt count
    unsigned int    reserved2;

    unsigned long long dmar_bytes;      // Total bytes read by DMA
    unsigned long long dmaw_bytes;     // Total bytes wrote by DMA
} XD48S_DRIVER_STATS;
```

Fields

struct_size

This field will be used to discriminate against possible future versions of this structure. This field should be initialized to the structure size (e.g. sizeof(XD48S_DRIVER_STATS)) by the caller.

nConnections

The total number of active connections (device handles) to the PXDAC4800. This is the count over all threads and processes.

isr_cnt

This is the total number of times that the PXDAC4800 interrupt service routine has been executed. This is not necessarily the total number of times that the device has interrupted the system since interrupt lines (IRQ) can be shared by multiple devices.

dcm_reset_cnt

These fields count the number of times the digital clock manager (DCM) has been reset. Any time the playback clock rate changes, the DCM will need to be reset at some point before the next playback operation. The underlying software manages the resetting of DCMs.

dmaw_finished_cnt
dmaw_started_cnt
dmaw_finished_cnt
dmaw_started_cnt

These fields count the total number of DMA transfers started/completed between the PXDAC4800 and host system. DMA transfers to the card are represented by the *dmaw_** fields. DMA transfers from the card to the system are represented by the *dmaw_** fields.

pb_started_cnt

The total number of playback operations started. This includes both RAM and Streaming playback operations.

pb_finished_cnt

The total number of concrete-length playback operations completed. This includes only RAM playbacks since Streaming playback operations are always infinite at the firmware/hardware layer.

samp_comp_int_cnt

The current Samples Complete Interrupt counter value. Each time a Samples Complete interrupt is received from the card this counter is incremented.

reserved2

Reserved for future use.

dmaw_bytes
dmaw_bytes

The total number of bytes transferred to the card (*dmaw_bytes*) or transferred from the card (*dmaw_bytes*).

3.5.5 Structure: XD48S_RECORDED_DATA_INFO

This structure is used by the [GetRecordedDataInfoXD48](#) library function and is used to obtain details on previously saved acquisition data. This information is obtained by reading auxiliary SRDC data that can be saved along with data acquired by Signatec data acquisition devices.

Structure definition

```
/// Recorded data information; see GetRecordedDataInfoXD48
typedef struct _XD48S_RECORDED_DATA_INFO_tag
{
    unsigned int    struct_size;        // struct size in bytes

    unsigned int    boardSerialNum;     // Serial number
    char            boardName[16];      // Name of board

    unsigned int    channelCount;       // Channel count
    unsigned int    channelNum;         // Channel ID; single channel data
    int             channelMask;        // Channel mask or 0

    unsigned int    sampSizeBytes;      // Sample size in bytes
    unsigned int    sampSizeBits;      // Sample size in bits
    int             bSignedSamples;     // Signed or unsigned
}
```

```

    unsigned int    segment_size;        // Segment size or zero
    unsigned int    trig_offset;         // Relates first sample to trig
    int             bPreTrigger;         // ? Pre-trigger : trigger delay

    unsigned int    header_bytes;        // Size of app-specific header
    int             bTextData;           // Data is text (versus binary)?
    int             textRadix;           // Radix of text data (10/16)

    double          sampleRateMHz;       // Sampling rate in MHz
    double          inputVoltRngPP;      // Peak-to-peak input volt range
} XD48S_RECORDED_DATA_INFO;

```

Fields

struct_size

This field will be used to discriminate against possible future versions of this structure. This field should be initialized to the structure size (e.g. `sizeof(XD48S_RECORDED_DATA_INFO)`) by the caller.

boardName

This field will contain a NULL-terminated string of the name of the device that generated the underlying data, such as “PDA16”, “PX14400”, “PX1500-4”, etc.

boardSerialNum

The serial number of the device that generated the underlying data.

channelCount

The channel count of the underlying data.

channelNum

For multichannel data this value will be 0 or -1 (0xFFFFFFFF). For single channel data, this will be the one-based channel index that acquired the data. (1 = channel 1, 2 = channel 2, etc)

sampSizeBytes

The size of a single data sample in bytes.

sampSizeBits

The size of a single data sample in bits.

bSignedSamples

This field will be non-zero if the underlying data samples are signed ($\pm N$).

sampleRateMHz

The sampling rate, in MHz, used to acquire the underlying data.

inputVoltRngPP

The peak-to-peak input voltage range used to acquire the underlying data.

segment_size

This field is used to identify data acquired using segmented trigger mode. If this value is non-zero, then it is the segment size in samples. If this value is zero, then the underlying data is not segmented.

trigger_delta

This field is used to identify the trigger position relative to the start of the acquisition data (or segment if data is segmented). Certain data acquisition devices can begin storing data before or a certain number of acquisition clock cycles after the trigger event. If the value of this field is zero, the first sample is the trigger point. If the value of this field is positive then the trigger point is this many samples into the record. If the value of this field is negative then the trigger point is this many samples before the start of data, meaning a certain number of samples were ignored after the trigger event.

header_bytes

The number of bytes set aside for a user-defined file header. If this value is non-zero then acquisition data begins at *header_bytes* into the file.

bTextData

The underlying acquisition data is saved as text, not binary. Signatec software, such as the PXDAC4800 Playback Application will not read text data.

textRadix

If data is saved as text, this field defines the radix of the output data: 10 for decimal, 16 for hexadecimal.

3.5.6 Structure: XD48S_STREAM_SES_CREATE

This structure is used by the [SessionStreamCreateXD48](#) library function and is used to specify the various parameters that control how a streaming playback session will run. A streaming playback session is a library-controlled streaming playback operation in which playback data is streamed to the PXDAC4800 as it is being played back.

Structure definition

```
typedef struct _XD48S_STREAM_SES_tag
{
    unsigned int      struct_size;           // struct size in bytes

    int               stream_flags;          // XD48STREAMF_*
    unsigned long long stream_bytes;         // Bytes to stream or XD48STREAMLEN_*
    unsigned long long src_offset_bytes;     // Source data offset of playback data
    unsigned long long src_length_bytes;     // Source data length of playback data
    unsigned int      xfer_bytes;            // Transfer size or 0
    int               reserved;              // Pass zero for now

    unsigned int      ss_len_bytes;          // Playback snapshot size in bytes
    unsigned int      ss_period_xfer;        // Snapshot period in 1MB DMA transfers
    unsigned int      ss_period_ms;         // or snapshot period in milliseconds
    int               reserved2;             // Pass zero for now
} XD48S_STREAM_SES_CREATE;
```

Fields

struct_size

This field must be initialized to the size of the structure, in bytes, prior to using the structure with any of the library functions.

stream_flags

A set of flags that control streaming playback session behavior:

Symbolic Constant	Value	Interpretation
-------------------	-------	----------------

XD48STREAMF_FORCE_SW_TRIGGER	0x00000001	Force playback to begin without waiting for an external trigger
XD48STREAMF_DO_SNAPSHOTS	0x00000002	Periodically grab snapshots of playback data. (See <i>ss_*</i> fields below)
XD48STREAMF_NO_UNBUFFERED_IO	0x00000004	Do not try to use unbuffered IO
XD48STREAMF_INIT_STATIC_DATA	0x80000000	Pre-initialize input buffers with static playback data

stream_bytes

The total number of bytes to playback or one of XD48STREAMLEN_* values defined below. This parameter will be aligned up to the next DMA transfer boundary (128 bytes).

Symbolic Constant	Value	Interpretation
XD48STREAMLEN_INFINITE	0	Stream infinitely, looping around source data as necessary.
XD48STREAMLEN_SOURCE_LENGTH	0xFFFFFFFFFFFFFFFF	Defines stream size to be length of source data (minus source offset).

src_offset_bytes

This parameter specifies the byte offset into the source data at which to begin uploading playback data.

src_length_bytes

The number of bytes of the source data to upload. If this parameter is zero then all bytes of the data source (minus any bytes skipped by *src_offset_bytes*) will be used.

xfer_bytes

The data transfer size to use when uploading playback data. It is recommended to pass zero for this parameter, which lets the library decide the best transfer size.

reserved

This field is reserved for future use. Pass zero.

ss_len_bytes

This parameter defines the desired length of the playback data snapshot that will be periodically grabbed by the streaming playback. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

ss_period_xfer

The playback data snapshot period, in DMA transfers. For long, non-segmented streaming playbacks, a snapshot period in DMA transfers can be used to obtain synchronized playback snapshots over multiple independently running playback sessions. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

ss_period_ms

The playback data snapshot update period, in milliseconds. If the *ss_period_xfer* field is zero, the library will default back to a simple, approximate-timed approach for playback data snapshot updates. This field is only considered if the XD48STREAMF_DO_SNAPSHOTS flag is specified in *stream_flags*.

reserved2

This field is reserved for future use. Pass zero.

3.5.7 Structure: XD48S_STREAM_SES_PROG

This structure is used by the [SessionStreamProgressXD48](#) function to obtain the status and progress of a [streaming playback session](#).

Structure definition

```
/// Streaming Playback session progress; @see SessionStreamProgressXD48
typedef struct _XD48S_STREAM_SES_PROG_tag
{
    unsigned int      struct_size;          // struct size in bytes

    int               stream_status;        // XD48SPBSTATUS_*

    unsigned long long bytes_uploaded;       // Total bytes uploaded so far
    unsigned long long bytes_to_upload;     // Total bytes to upload or 0 for infinite

    unsigned int      elapsed_time_ms;      // Current elapsed uptime
    unsigned int      xfer_bytes;           // Transfer size in bytes
    unsigned int      xfer_count;           // Current transfer count
    unsigned int      snapshot_count;       // Current snapshot count

    int               progress_flags;       // XD48SPBPROGF_*

    // Valid when stream_status == XD48SPBSTATUS_ERROR
    int               err_res;              // SIG_*
    wchar_t*          err_textp;           // Caller frees with FreeMemoryXD48

    void*             reserved;            // Reserved for future use
} XD48S_STREAM_SES_PROG;
```

Fields

struct_size

This field must be initialized to the size of the structure, in bytes, prior to using the structure with any of the library functions.

stream_status

The current status of the streaming playback session. This can be one of the following values:

Library Constant	Interpretation
XD48SPBSTATUS_IDLE (0)	Idle; playback session not yet started
XD48SPBSTATUS_ACTIVE (1)	Active; Playback session is in progress. Hardware is armed for playback or playing back.
XD48SPBSTATUS_COMPLETE (2)	Complete; Playback session complete or stopped by user
XD48SPBSTATUS_ERROR (3)	Error; Playback session has ended due to error

bytes_uploaded

The total number of bytes uploaded to the card so far.

bytes_to_upload

The total number of bytes to upload and playback for the session. If this parameter is zero then the session is configured for an infinite playback.

elapsed_time_ms

The total milliseconds that have elapsed since the streaming playback session was started. Note that this is not an exact measure of how long the hardware has been playing back data. The software has no knowledge of when the hardware is actually playing back data or when the hardware is armed and waiting for a trigger.

xfer_bytes

The data transfer size used by the recording session. This is for information purposes only.

xfer_count

The current transfer counter. Each time a data transfer of playback data completes the transfer counter is incremented.

snapshot_count

The current data snapshot counter value. Each time the streaming playback session collects a new playback data snapshot the snapshot counter is incremented. This value is only relevant if the session was configured to collect snapshots. See XD48_STREAM_SES_PROG::stream_flags for more details. Data snapshots are obtained by calling SessionStreamSnapshotXD48.

progress_flags

A set of flags indicating that certain events have taken place:

Symbolic Constant	Value	Interpretation
XD48SPBPROGF_DAC_FIFOS_FILLED	0x00000001	DAC FIFOs have filled; safe to trigger
XD48SPBPROGF_SW_TRIG_ISSUED	0x00000002	A software trigger has been generated

err_res

This field is only relevant if the *stream_status* member is XD48SPBSTATUS_ERROR. This is the [library error code](#) (SIG_*) of the error that caused the recording session to prematurely end.

err_textp

This field is only relevant if the *stream_status* member is XD48SPBSTATUS_ERROR. This is a library-allocated string containing a description of the error that occurred during the recording session. The caller must free the memory for this string when it is no longer needed by passing the address of the string to FreeMemoryXD48. If the XD48SPBPROGF_NO_ERROR_TEXT flag is passed to [SessionStreamProgressXD48](#) then no error text will be generated and this member will be set to NULL.

reserved

This field is reserved for future use. Currently this field will always be set to NULL.

4 Appendix A – PXDAC4800 Library Error Codes

The table below lists all of the currently defined PXDAC4800 library error codes.

The GetErrorTextXD48 library function can be used to generate a user-friendly string for any of these error codes.

Symbolic constant	Error Code	Interpretation
SIG_SUCCESS	0	Operation successful
SIG_XD48_QUASI_SUCCESSFUL	512	Operation was quasi-successful; one or more items failed
SIG_ERROR	-1	Generic error; platform's system error may provide more info
SIG_INVALIDARG	-2	An invalid argument was specified
SIG_OUTOFBOUNDS	-3	An argument is out of valid bounds
SIG_NODEV	-4	Invalid board device
SIG_OUTOFMEMORY	-5	Error allocating memory
SIG_DMABUFALLOCFAIL	-6	Error allocating a DMA buffer
SIG_NOSUCHBOARD	-7	Board with given serial or ordinal number not found
SIG_NT_ONLY	-8	This feature is only available on Windows NT platforms.
SIG_INVALID_MODE	-9	Invalid operation for current operating mode
SIG_CANCELLED	-10	Operation was cancelled by user
SIG_XD48_FIRST	-512	First PXDAC4800-specific error code value
SIG_XD48_NOT_IMPLEMENTED	-512	This operation is not currently implemented
SIG_XD48_INVALID_HANDLE	-513	An invalid PXDAC4800 device handle (HXD48) was specified
SIG_XD48_BUFFER_TOO_SMALL	-514	A specified buffer is too small
SIG_XD48_INVALID_ARG_1	-515	Argument 1 is invalid
SIG_XD48_INVALID_ARG_2	-516	Argument 2 is invalid
SIG_XD48_INVALID_ARG_3	-517	Argument 3 is invalid
SIG_XD48_INVALID_ARG_4	-518	Argument 4 is invalid
SIG_XD48_INVALID_ARG_5	-519	Argument 5 is invalid
SIG_XD48_INVALID_ARG_6	-520	Argument 6 is invalid
SIG_XD48_INVALID_ARG_7	-521	Argument 7 is invalid
SIG_XD48_INVALID_ARG_8	-522	Argument 8 is invalid
SIG_XD48_XFER_SIZE_TOO_SMALL	-523	Requested transfer size is too small
SIG_XD48_XFER_SIZE_TOO_LARGE	-524	Requested transfer size is too large
SIG_XD48_INVALID_DMA_ADDR	-525	Given address is not part of a DMA buffer
SIG_XD48_WOULD_OVERRUN_BUFFER	-526	Operation would overrun given buffer
SIG_XD48_BUSY	-527	Device is busy; try again later
SIG_XD48_INVALID_CHAN_IMP	-528	Incorrect function for board's channel implementation
SIG_XD48_XML_MALFORMED	-529	Invalid XML data was encountered
SIG_XD48_XML_INVALID	-530	XML data was well formed, but not valid
SIG_XD48_XML_GENERIC	-531	Generic XML related error
SIG_XD48_RATE_TOO_FAST	-532	The specified rate is too fast

SIG_XD48_RATE_TOO_SLOW	-533	The specified rate is too slow
SIG_XD48_RATE_NOT_AVAILABLE	-534	The specified frequency is not available; see operator's manual
SIG_XD48_UNEXPECTED	-535	An unexpected error occurred; debug builds will have failed assertion
SIG_XD48_SOCKET_ERROR	-536	A socket error occurred
SIG_XD48_NETWORK_NOT_READY	-537	Network subsystem is not ready for network communication
SIG_XD48_SOCKETS_TOO_MANY_TASKS	-538	Limit on number of tasks/processes using sockets has been reached
SIG_XD48_SOCKETS_INIT_ERROR	-539	Generic sockets implementation start up failure
SIG_XD48_NOT_REMOTE	-540	Not connected to a remote PXDAC4800 device
SIG_XD48_TIMED_OUT	-541	Operation timed out
SIG_XD48_CONNECTION_REFUSED	-542	Connection refused by service; service may not be running
SIG_XD48_INVALID_CLIENT_REQUEST	-543	Received an invalid client request
SIG_XD48_INVALID_SERVER_RESPONSE	-544	Received an invalid server response
SIG_XD48_REMOTE_CALL_RETURNED_ERROR	-545	Remote service call returned with an error
SIG_XD48_UNKNOWN_REMOTE_METHOD	-546	Undefined method invoked on remote server
SIG_XD48_SERVER_DISCONNECTED	-547	Server closed the connection
SIG_XD48_REMOTE_CALL_NOT_AVAILABLE	-548	Remote call for this operation is not implemented or available
SIG_XD48_UNKNOWN_FW_FILE	-549	Unknown firmware file type
SIG_XD48_FIRMWARE_UPLOAD_FAILED	-550	Firmware upload failed
SIG_XD48_INVALID_FW_FILE	-551	Invalid firmware upload file
SIG_XD48_DEST_FILE_OPEN_FAILED	-552	Failed to open destination file
SIG_XD48_SOURCE_FILE_OPEN_FAILED	-553	Failed to open source file
SIG_XD48_FILE_IO_ERROR	-554	File IO error
SIG_XD48_INCOMPATIBLE_FIRMWARE	-555	Firmware is incompatible with PXDAC4800
SIG_XD48_UNKNOWN_STRUCT_SIZE	-556	Unknown structure size specified to library function (X::struct_size)
SIG_XD48_INVALID_REGISTER	-557	An invalid hardware register read/write was attempted
SIG_XD48_FIFO_UNDERFLOW	-558	An internal FIFO went empty during playback; could not keep up with playback data rate
SIG_XD48_DCM_SYNC_FAILED	-559	PXDAC4800 firmware could not synchronize to acquisition clock
SIG_XD48_DISK_FULL	-560	Could not write all data; disk is full
SIG_XD48_INVALID_OBJECT_HANDLE	-561	An invalid object handle was used
SIG_XD48_THREAD_CREATE_FAILURE	-562	Failed to create a thread
SIG_XD48_CG_PLL_LOCK_FAILED	-563	Playback clock phase lock loop (PLL) failed to lock; clock may be bad
SIG_XD48_THREAD_NOT_RESPONDING	-564	Recording thread is not responding
SIG_XD48_PLAYBACK_SESSION_ERROR	-565	A playback session error occurred
SIG_XD48_PLAYBACK_SESSION_CANNOT_ARM	-566	Cannot arm playback session; already armed or stopped
SIG_XD48_SNAPSHOTS_NOT_ENABLED	-567	Snapshots not enabled for given recording session

SIG_XD48_SNAPSHOT_NOT_AVAILABLE	-568	No data snapshot is available
SIG_XD48_SRDC_FILE_ERROR	-569	An error occurred while processing .SRDC file
SIG_XD48_NAMED_ITEM_NOT_FOUND	-570	Named item could not be found
SIG_XD48_CANNOT_FIND_SRDC_DATA	-571	Could not find Signatec Recorded Data Context info
SIG_XD48_NOT_IMPLEMENTED_IN_FIRMWARE	-572	Feature is not implemented in current firmware version; upgrade firmware
SIG_XD48_CANNOT_DETERMINE_FW_REQ	-573	Cannot determine which firmware needs to be uploaded; update software
SIG_XD48_REQUIRED_FW_NOT_FOUND	-574	Required firmware not found in firmware update file
SIG_XD48_FIRMWARE_IS_UP_TO_DATE	-575	Loaded firmware is up to date with firmware update file
SIG_XD48_NO_VIRTUAL_IMPLEMENTATION	-576	Operation not implemented for virtual devices
SIG_XD48_DEVICE_REMOVED	-577	PXDAC4800 device has been removed from system
SIG_XD48_JTAG_IO_ERROR	-578	JTAG IO error; likely firmware upload IO error
SIG_XD48_ACCESS_DENIED	-579	Access denied
SIG_XD48_CG_PLL_LOCK_FAILED_UNSTABLE	-580	Playback clock phase lock loop (PLL) failed to lock; lock not stable
SIG_XD48_UNREACHABLE	-581	Item could not be set with the given constraints
SIG_XD48_INVALID_MODE_CHANGE	-582	Invalid operating mode change; all changes must go or come from Standby
SIG_XD48_DEVICE_NOT_READY	-583	Underlying device is not yet ready; try again shortly
SIG_XD48_ALIGNMENT_ERROR	-584	A parameter is not aligned properly
SIG_XD48_INVALID_OP_FOR_BRD_CONFIG	-585	Invalid operation for current board configuration
SIG_XD48_UNKNOWN_JTAG_CHAIN	-586	Unknown JTAG chain configuration; out-of-date software or hardware error
SIG_XD48_INVALIDARG_NULL_POINTER	-587	An invalid pointer was specified
SIG_XD48_NOT_IMPLEMENTED_IN_DRIVER	-589	Feature is not implemented in current driver version; upgrade software
SIG_XD48_TEXT_CONV_ERROR	-590	An error occurred while converting text from one encoding to another
SIG_XD48_UNKNOWN_STATE	-591	Unknown, invalid, or corrupted internal software state
SIG_XD48_CALIBRATION_ERROR	-592	Calibration failed
SIG_XD48_INVALID_CHANNEL_MASK	-593	An invalid active channel mask specified; usually no channels selected
SIG_XD48_INVALID_TRIGGER_MODE	-594	Invalid trigger mode for the requested operation
SIG_XD48_INVALID_OFFSET	-595	Invalid offset specified; likely exceeds data length
SIG_XD48_NO_PLAYBACK_DATA_SELECTED	-596	No playback data selected; may be empty data file
SIG_XD48_DAC_PLL_LOCK_FAILED	-597	DAC phase lock loop (PLL) failed to lock;

		clock may be bad
SIG_XD48_DAC_PLL_LOCK_FAILED_UNSTABLE	-598	DAC phase lock loop (PLL) failed to lock; lock not stable
SIG_XD48_CFG_EEPROM_VALIDATE_ERROR	-599	Configuration EEPROM validation failed
SIG_XD48_RESOURCE_ALLOC_FAILURE	-600	Failed to allocate or create a system object such as semaphore, mutex, etc
SIG_XD48_CANNOT_FIT_FREQ	-601	Cannot fit data with specified frequency into a whole, aligned sample count
SIG_XD48_GENERIC_LIB_ERROR	-602	Placeholder for generic library error; error text (GetErrorTextXD48) will be a specific error description
SIG_XD48_BIST	-603	DAC auto-calibration failed (General)
SIG_XD48_BIST_VALIDATION	-604	DAC auto-calibration failed (Validation)
SIG_XD48_BIST_UPLOAD_FAILED	-605	Unable to upload the file calibration file
SIG_XD48_NEED_UPLOAD	-606	You need to do a upload before proceeding
SIG_XD48_OVERLOAD	-607	The card cannot be in full speed(>600 MHz), full resolution(16 bits) and 4 channels at the same time
SIG_XD48_PROD_ERROR	-608	Reserved: Internal usage only
SIG_XD48_STREAM_TRIG_ERROR	-609	Streaming playback requires single shot trigger mode
SIG_XD48_PROD_ERROR_READ	-610	Reserved: Internal usage only
SIG_XD48_FUNCTION_NOT_SUPPORTED_AC	-611	This function is not supported with the PXDAC4800A. (Only with the DC version)
SIG_XD48_CUSTOM_DAC_VALUE_NOT_ENABLE	-612	You need to enable the custom DAC value option before calling this function. (SetCustomDacValueEnableXD48)

5 Appendix B – Revision History

Revision 1.0

- First official revision with full release info

Revision 1.1

- Replaced GetFirmwareVersionXD48 and GetHardwareRevisionXD48 functions with the newer GetItemVersionXD48 function
- Added GetVersionTextXD48
- Corrected function name: [GS]etExtTriggerDirXD48 is [GS]etExternalTriggerDirXD48
- Added real picture of device to physical layout
- Added a number of library functions
- Cleaned up format of function prototypes

Revision 1.2

- Removed deprecated API functions and features
- Modified API function:
 - GetItemVersionXD48

Revision 1.3

- Added API function
 - GetStatusFPGAXD48
- Modified API function
 - IsPlaybackInProgressXD48

Revision 1.4

- Added API functions
 - SetDigitalIoCfgXD48
 - GetDigitalIoCfgXD48
 - SetDigitalIoModeXD48
 - GetDigitalIoModeXD48
 - SessionStreamCreateStdXD48
 - _SetStreamingLengthXD48

Revision 1.5

- Added API functions
 - IsDcXD48
 - SetFiltersCHXD48
 - GetFiltersCHXD48
 - SetCustomDacDefaultValueXD48
 - GetCustomDacDefaultValueXD48
 - SetCustomDacValueEnableXD48
 - GetCustomDacValueEnableXD48
 - GetErrorMessXD48
 - SessionStreamCreateParmsXD48
 - SessionStreamDeleteNoStructXD48
 - SessionStreamEndNoStructXD48
 - SessionStreamGetProgressXD48
 - StartDacAutoCalibrationXD48
- Removed master/slave documentation (Deprecated)

Revision 1.6

- Add remarks about DC boards

Revision 1.7

- Add description for “Frame Start” Digital I/O.

Revision 1.71

- Copyright Vitrek LLC.