PDA16

Operator's Manual

DynamicSignals LLC
900 North State Street
Lockport, Illinois 60441-2200
USA

Tel: (815) 838-0005
Fax: (815) 838-4424
Web: http://www.signatec.com
Web: http://dynamicsignals.com

Revision 1.13 – 04/04/2012

Table of Contents

1 BEFORE USING THE PDA16!! 7

1.1 Package Contents 7
1.2 Unpacking and Handling 7
1.3 Checking for Damage 7
1.4 Warranty 7
1.5 System Requirements 7
1.6 Software 7
1.7 Physical Layout 8

2 FUNCTIONAL DESCRIPTION 8

2.1 Overview 8
2.2 Board Configuration – Master/Slave Setup 9
2.2.1 Standalone Board Operation 9
2.2.2 Master-Slave Operation 9
2.3 Active Channels 9
2.4 Operating Modes 11
2.4.1 Standby Mode 11
2.4.2 RAM-Buffered PCI Acquisition Mode 11
2.4.3 SAB Acquisition Mode 12
2.4.4 SAB Buffered Acquisition Mode 12
2.4.5 RAM Acquisition Mode 12
2.4.6 PCI Transfer Mode 13
2.4.7 SAB Transfer Mode 13
2.4.8 PCI Write RAM Mode 13
2.5 Analog Input Circuitry 13
2.6 ADC Clocking 14
2.7 Triggering the PDA16 15
2.7.1 Minimizing Jitter - Synchronized Triggering 15
2.7.1.1 External Clock and Trigger 15
2.7.1.2 Synchronizing the Trigger via Clock Out 15
2.7.1.3 Synchronizing the Trigger via Trigger Out 15
2.7.2 Trigger Modes 15
2.7.2.1 Post-Trigger Mode 15
2.7.2.2 Segmented Trigger Mode 15
2.7.3 Trigger Timing Options 16
2.7.3.1 Pre-trigger Samples 16
2.7.3.2 Delayed Trigger 16
2.7.4 Trigger Setup 16
2.7.5 Software Trigger 16
2.8 Digital IO 17
2.9 Active Memory Region 17
2.9.1 Starting Sample 17
2.9.2 Sample Count 17
2.9.3 Segment Size 17
2.10 PCI Interface 17
2.10.1 Plug and Play Features 18
2.10.2 DMA Transfers 18
2.11 Signatec Auxiliary Bus Interface 18
2.11.1 SAB Data Transfer 19
2.11.1.1 SAB Configuration 19
2.11.1.2 SAB Acquisition Mode 19
2.11.1.3 SAB Transfer Mode 19
2.11.2 SAB Commands 19
2.12 PDA16 Acquisition Data Format 20
2.12.1 Sample Format 20

2.12.2 Multichannel Data 20
2.12.3 PDA16 Sample Data Files (*.rd16) 21
2.12.4 Signatec Recorded Data Context Files (*.srdc) 21

3 PDA16 SOFTWARE DEVELOPMENT REFERENCE 21

3.1 Getting Started 21
3.1.1 Contents 21
3.1.2 Installing Software 22
3.1.3 Developing PDA16 Software 22
3.1.3.1 Setting up the Build Environment – Windows Platform 22
3.1.3.2 Setting up the Build Environment – Linux Platform 23
3.1.3.3 Library Functions That Use Character Strings 23
3.2 Library Utility Functions 24
3.2.1 GetErrorTextD16 24
3.3 PDA16 Device Enumeration and Connection Management 25
3.3.1 ConnectToDeviceD16 25
3.3.2 ConnectToVirtualDeviceD16 26
3.3.3 DuplicateHandleD16 26
3.3.4 DisconnectFromDeviceD16 27
3.3.5 GetDeviceCountD16 28
3.3.6 IsDeviceRemoteD16 28
3.3.7 IsDeviceVirtualD16 28
3.3.8 IsHandleValidD16 29
3.4 PDA16 Device State and Configuration 29
3.4.1 GetActualAdcAcqRateD16 30
3.4.2 GetEffectiveAcqRateD16 30
3.4.3 GetFifoFullFlagD16 31
3.4.4 GetFirmwareVersionD16 31
3.4.5 GetHardwareRevisionD16 32
3.4.6 GetOrdinalNumberD16 33
3.4.7 GetSamplesCompleteFlagD16 33
3.4.8 GetSerialNumberD16 34
3.4.9 ReadConfigEepromD16 34
3.4.10 WriteConfigEepromD16 35
3.5 PDA16 Hardware Settings 36
3.5.1 IssueSoftwareTriggerD16 36
3.5.2 SetActiveChannelsD16 37
3.5.3 SetAdcClockReferenceD16 37
3.5.4 SetAdcClockSourceD16 38
3.5.5 SetAdcDitheringEnableD16 39
3.5.6 SetAdcRandomizedOutputDataEnableD16 40
3.5.7 SetAddressCounterAutoResetOverrideD16 40
3.5.8 SetBoardProcessingEnableD16 41
3.5.9 SetBoardProcessingParamD16 42
3.5.10 SetClockDivider*D16 43
3.5.11 SetDigitalOutputEnableD16 44
3.5.12 SetDigitalOutputModeD16 44
3.5.13 SetExternalClockRateD16 45
3.5.14 SetInternalClockRateD16 46
3.5.15 SetMasterSlaveConfigurationD16 47
3.5.16 SetOperatingModeD16 48
3.5.17 SetPostAdcClockDividerD16 49
3.5.18 SetPreTriggerSamplesD16 50
3.5.19 SetSabBoardNumberD16 51
3.5.20 SetSabClockD16 52
3.5.21 SetSabConfigurationD16 53
3.5.22 SetSegmentSizeD16 54
3.5.23 SetSampleCountD16 55
3.5.24 SetStartSampleD16 55

3.5.25 SetTimestampCounterModeD16 56
3.5.26 SetTimestampModeD16 57
3.5.27 SetTriggerDelaySamplesD16 59
3.5.28 SetTriggerDirectionAD16 59
3.5.29 SetTriggerDirectionBD16 60
3.5.30 SetTriggerDirectionExtD16 61
3.5.31 SetTriggerLevelAD16 62
3.5.32 SetTriggerLevelBD16 63
3.5.33 SetTriggerModeD16 64
3.5.34 SetTriggerSourceD16 64
3.5.35 SetVoltRangeCh1D16 65
3.5.36 SetVoltRangeCh2D16 66
3.6 Device Register State Functions 67
3.6.1 CopyHardwareSettingsD16 67
3.6.2 LoadSettingsFromBufferXmlD16 68
3.6.3 LoadSettingsFromFileXmlD16 68
3.6.4 ReadAllDeviceRegistersD16 69
3.6.5 RefreshLocalRegisterCacheD16 70
3.6.6 RewriteHardwareSettingsD16 70
3.6.7 SaveSettingsToBufferXmlD16 71
3.6.8 SaveSettingsToFileXmlD16 72
3.6.9 SetPowerupDefaultsD16 73
3.7 Memory/DMA Buffer Allocation Routines 73
3.7.1 AllocateDmaBufferD16 73
3.7.2 FreeDmaBufferD16 75
3.7.3 FreeMemoryD16 75
3.8 Data Acquisition Routines 76
3.8.1 AcquireToBoardRamD16 76
3.8.2 AcquireToSabD16 77
3.8.3 BeginBufferedPciAcquisitionD16 78
3.8.4 EndBufferedPciAcquisitionD16 79
3.8.5 IsAcquisitionInProgressD16 79
3.8.6 WaitForAcquisitionCompleteD16 80
3.9 Data Transfer Routines 81
3.9.1 GetPciAcquisitionDataD16 81
3.9.2 IsTransferInProgressD16 82
3.9.3 ReadSampleRamFastD16 83
3.9.4 ReadSampleRamD16 84
3.9.5 ReadSampleRamDualChannelD16 85
3.9.6 ReadSampleRamFileFastD16 86
3.9.7 ReadSampleRamFileD16 87
3.9.8 TransferSampleRamToSabD16 88
3.9.9 WaitForTransferCompleteD16 89
3.10 Data Manipulation Routines 90
3.10.1 DeInterleaveDataD16 90
3.10.2 InterleaveDataD16 91
3.11 PDA16 Recording Session Management Routines 92
3.11.1 AbortRecordingSessionD16 92
3.11.2 ArmRecordingSessionD16 92
3.11.3 CreateRecordingSessionD16 93
3.11.4 DeleteRecordingSessionD16 94
3.11.5 GetRecordingSessionOutFlagsD16 94
3.11.6 GetRecordingSessionProgressD16 95
3.11.7 GetRecordingSnapshotD16 96
3.12 Remote PDA16 Operation 96
3.12.1 ConnectToRemoteDeviceD16 97
3.12.2 FreeServiceReponseD16 98
3.12.3 GetHostServerInfoD16 99

3.12.4 GetRemoteDeviceCountD16 99
3.12.5 GetServiceSocketD16 100
3.12.6 SendServiceRequestD16 101
3.12.7 SocketsCleanupD16 102
3.12.8 SocketsInitD16 103
3.13 Signatec Recorded Data Context (SRDC) Information 103
3.13.1 CloseSrdcFileD16 106
3.13.2 EnumSrdcItemsD16 106
3.13.3 GetRecordedDataInfoD16 107
3.13.4 GetSrdcItemD16 108
3.13.5 IsSrdcFileModifiedD16 109
3.13.6 OpenSrdcFileD16 109
3.13.7 RefreshSrdcParametersD16 110
3.13.8 SaveSrdcFileD16 111
3.13.9 SetSrdcItemD16 112
3.14 PDA16 Timestamp Management Routines 112
3.14.1 GetTimestampAvailabilityD16 113
3.14.2 GetTimestampFifoDepthD16 113
3.14.3 GetTimestampOverflowFlagD16 114
3.14.4 ReadTimestampDataD16 114
3.15 PDA16 Library Data Types 116
3.15.1 Data Type: HPDA16 116
3.15.2 Data Type: HD16RECORDING 116
3.15.3 Data Type: HD16SRDC 116
3.15.4 Data Type: pda16_sample_t 116
3.15.5 Data Type: pda16_timestamp_t 116
3.15.6 Structure: D16S_FILE_WRITE_PARAMS 116
3.15.7 Structure: D16S_REC_SESSION_PARAMS 121
3.15.8 Structure: D16S_REMOTE_CONNECT_CTX 123
3.15.9 Callback Function: D16_FILEIO_CALLBACK 124

4 APPENDIX A - PDA16 SPECIFICATIONS & ORDERING INFORMATION 126

5 APPENDIX B - MASTER-SLAVE CONNECTIONS 127

6 APPENDIX C - SIGNATEC AUXILIARY BUS CONNECTIONS 128

7 APPENDIX F – PDA16 LIBRARY ERROR CODES 130

8 APPENDIX G – REVISION HISTORY 133

IMPORTANT NOTICE
ON
HARDWARE COMPATIBILITY

The PDA16 is a PCI Local Bus compliant device. As such the PDA16 contains the configuration space register organization as defined by the PCI Local Bus Specification. Among the functions of the configuration registers is the storage of unique identification values for the PDA16 as well as storage of base address size requirements for PDA16 operation.

The host computer that the PDA16 is installed in is responsible for reading and writing to/from the PCI 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 PDA16 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 PDA16 inoperable. It is beyond the ability of Signatec hardware or software to force a non-PnP system to operate a PDA16 board.

In addition to PnP requirements, the PDA16 must be operated in a 64-bit PCI or PCI-X slot. The PDA16 is not compatible with 32-bit PCI.

1 Before Using the PDA16!!

1.1 Package Contents

The PDA16 package shipped to you will normally contain (as a minimum) the following:

- PDA16 circuit board assembly
- A binder containing the following:
  - PDA16 Operator's Manual
  - PDA16 Windows Software Disk
- Two BNC to SMA coaxial cables

1.2 Unpacking and Handling

The PDA16, 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 PCI 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 3-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

The PDA16 requires the following minimum hardware configuration:

DOS and Windows NT (3.x/4.x)/95/98/ME are NOT supported by the PDA16 software.

For Windows 2000/XP/Vista (32-bit)
Pentium 133 or higher CPU with at least 64MB RAM with at least one open 64-bit PCI slot and PnP BIOS.

1.6 Software

The PDA16 is supplied with a library of software functions that may be used in client programs to simplify the job of creating custom PDA16 applications. The source code is provided for all the functions so that they may be used with all C/C++ language compilers and may be modified to suit the user's needs. Though Signatec gives the user this additional option or flexibility of modifying the PDA16 software library, making such changes is seldom necessary and the vast majority of users simply use the pre-built libraries provided by Signatec.

The PDA16 library functions have been tested and compiled using Microsoft's Visual Studio .NET 2008. Hereafter when Windows is mentioned it is assumed to mean Windows 2000/XP/Vista/7 unless otherwise noted. It is possible that some modification may be needed for other vendor's compilers. All names of functions in this manual are either in *italics* or else in blue and underlined, indicating they are linked references. For more information on a function, please refer to the section on that specific function in the PDA16 Software Development Reference section.

1.7 Physical Layout

The PDA16 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, the fourth connector is the analog input signal for channel 1, and the fifth connector is the analog input signal for channel 2.

At the top of the board, near the left-hand side, is the master-slave connector that is used to connect a master board to up to three slave boards. A slave board derives its clock and trigger signals from the master board so that data on all boards in the master-slave system is completely synchronous (or aligned), even for large amounts of memory.

Along the right portion of the board are two 100-pin connectors designated as J2 and J3. These connectors are used for the Signatec Auxiliary Bus (SAB) interface. Using the SAB, the PDA16 is capable of transferring data at a maximum rate of 500 megabytes per second. See section 2.11 for details of the SAB interface.

The PDA16 employs a PCIX bus via the 64-bit Card Edge Connector and can be configured at the factory for PCI-X 100/PCI-X 66 MHz or PCI-64 66 MHz (for non PCI-X slots). The default configuration is PCI-X 66 MHz. PCI 32-bit busses are not supported.

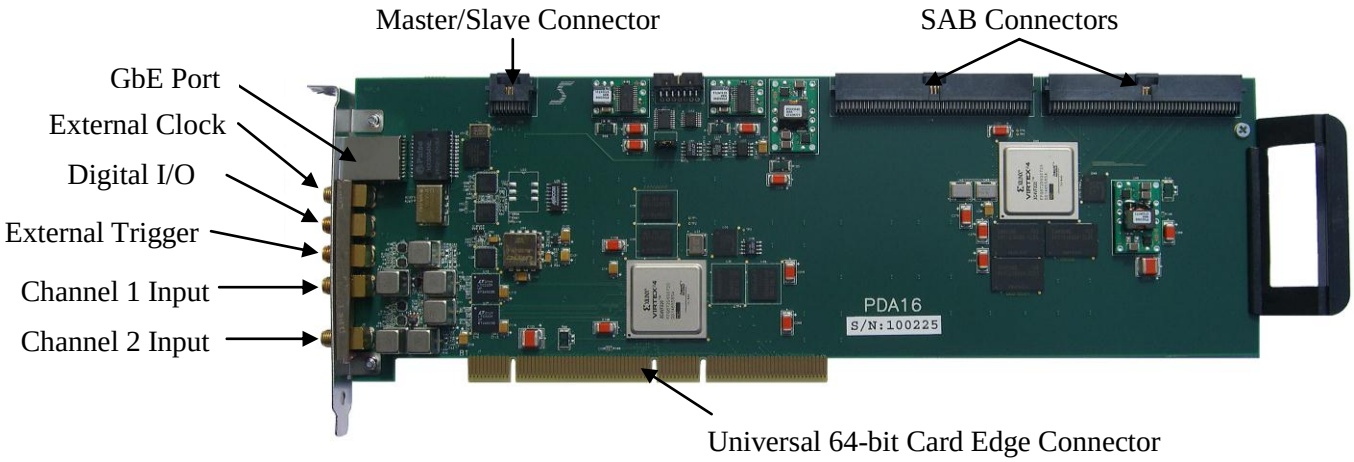


Figure 1

2 Functional Description

2.1 Overview

The PDA16 is a circuit board assembly designed and tested per the Peripheral Component Interconnect (PCI) Local Bus Specification Revision 2.2. This powerful waveform capture board is designed with the capability of capturing very large amounts of signal information with high amplitude resolution and at high digitization rates.

The PDA16 features 2 analog signal channels, each with 256 mebi-samples (268435456 samples) of on-board RAM and a maximum digitization rate of 160 mega-samples per second.

The PDA16 implements two high-speed buses: PCIX and the internal Signatec Auxiliary Bus (SAB). The PCIX bus implementation can sustain data flow at 640 megabytes/sec in Buffered (real time) Acquisition mode and even greater in Data Transfer Mode (for data already in system RAM).

The Signatec Auxiliary Bus (SAB) has a data width of 64 bits and is capable of sustained data transfer rates up to 500 megabytes per second. This interface is used for transferring data acquired by the PDA16 to other devices such as DSP boards or high-speed disk systems. The PDA16's SAB can also be operated as a 32-bit bus.

The on-board data acquisition memory may be used either as the target for acquisition data or as a large FIFO for real-time data acquisition to the PCI or SAB buses. A large FIFO is especially important for real time acquisition over the PCI bus since data transfer is frequently held off for extended periods of time for host system activities.

Data acquired directly to the SAB is useful for real-time signal processing systems. The Signatec PMP1000 digital signal processor board is capable of controlling the PDA16 over the SAB interface without the need for host computer intervention. This allows for maximum data acquisition/processing rates in signal averaging and time critical applications. For more information on SAB Control see section 2.11.2.

The PDA16 has five external signal connections: a clock input, a trigger input, 2 analog input signals, and a user-selectable digital output 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.

2.2 Board Configuration – Master/Slave Setup

The PDA16 can be set to one of 3 possible operational configurations: Standalone Board, Master, or Slave.

2.2.1 Standalone Board Operation

If the PDA16 board is not operated as part of a Master-Slave combination then the configuration should be set to "Standalone Board", which is the power-up default. The configuration is set using the library function SetMasterSlaveConfigurationD16.

2.2.2 Master-Slave Operation

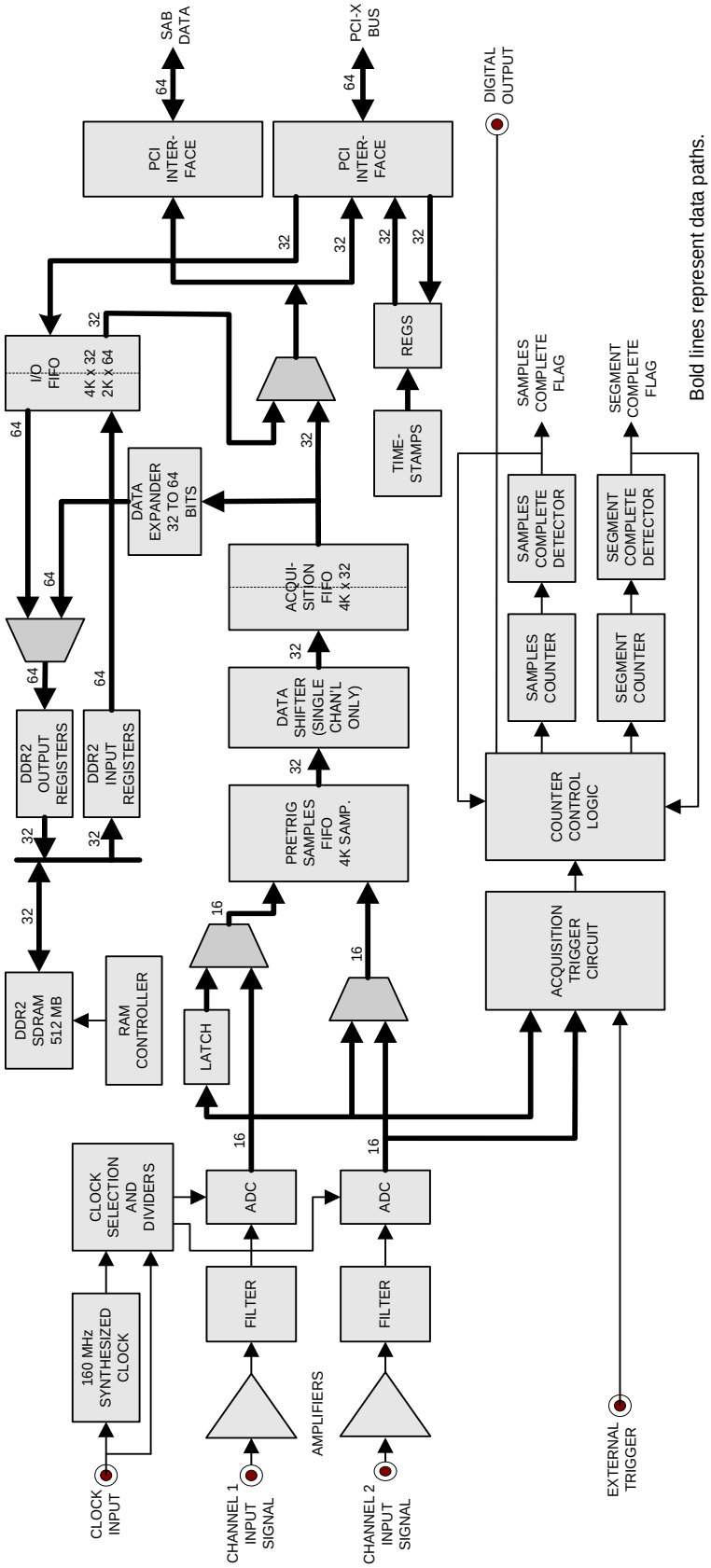
The PDA16 is designed with the capability to connect two PDA16 boards together for master-slave operation. In master-slave connection, the master board drives the clock and trigger signals for the slave board so that data on the slave board will align sample-for-sample with the data on the master board. Using a master and slave allows for the simultaneous acquisition of up to four channels at 180 MHz.

Boards are configured for Master-Slave operation by the library function SetMasterSlaveConfigurationD16.

A 12-conductor ribbon cable is used to interconnect the boards. The master-slave connector is shown pictorially in Figure 1. The electrical connections are specified in APPENDIX B - Master-Slave Connections.

2.3 Active Channels

The PDA16 may be set for operation with either one or two input channels. Use the library function SetActiveChannelsD16 to set the number of active channels.



PDA16 FUNCTIONALITY

Figure 2

2.4 Operating Modes

The PDA16 has several operating modes. The function SetOperatingModeD16 or GetOperatingModeD16 may be used to set or get the current operating mode via the PCI bus. The PDA16 library wraps the functionality of many of these operating modes in higher-level functions that include automatic management of mode changes and other details. See SetOperatingModeD16 function for details.

An SAB bus controller, such as Signatec's PMP1000 digital signal processor board, can also control the operating mode of the PDA16 by issuing direct commands across the SAB. See section 2.11.2 for more details regarding control of the PDA16 from the SAB.

The PDA16 operating modes are as follows:

Mode Number	Operating Mode
0 (0x00)	**RESERVED**
1 (0x01)	Standby
2 (0x02)	RAM Acquisition
3 (0x03)	**RESERVED**
4 (0x04)	PCI Buffered Acquisition
5 (0x05)	SAB Acquisition
6 (0x06)	SAB Buffered Acquisition
7 (0x07)	PCI Transfer
8 (0x08)	SAB Transfer
9 (0x09)	PCI Write RAM
10 - 15	Not Used

All operating mode changes on the PDA16 should either being going to, or coming from Standby mode. This means that, for example, an operating mode change from RAM Acquisition mode directly to PCI Transfer mode is not permitted. The SetOperatingModeD16 function implementation automatically ensures that all operating mode changes go to or from Standby mode. However, this behavior is not automatic when changing operating modes via SAB direct commands.

2.4.1 Standby Mode

This mode should be considered as the base mode for the PDA16. In this mode all circuitry, with the exception of the acquisition circuitry, is powered and the board is available for any command.

Unless overridden (via the SetAddressCounterAutoResetOverrideD16 function), putting the PDA16 into Standby mode will automatically reset the current RAM address to zero. This is often a faster method of resetting the current address than by manually by calling SetStartSampleD16.

2.4.2 RAM-Buffered PCI Acquisition Mode

In this mode of operation the entire on-board RAM may be utilized as a giant FIFO to prevent data from being lost due to periodic hold-off of the PCI data flow due to other system traffic. The maximum acquisition rate for a PCI acquisition is dependent upon the hardware of the host system and other PCI bus traffic. For data acquisitions larger than 256 mebi-samples, if the acquisition rate is higher than the PCI data rate then it is possible that the on-board FIFOs could overflow and data could be lost. A FIFO-full flag is available to be read at the end of the transfer to check for this FIFO overflow condition. The GetFifoFullFlagD16 library function may be used to read this flag.

Once the PDA16 is put into the PCI Buffered Acquisition mode and a trigger has been received, data from the ADC will flow through the RAM buffer to the PCI bus. While in this mode, the GetPciAcquisitionDataD16 library function is repeatedly called to obtain fresh acquisition data.

2.4.3 SAB Acquisition Mode

This mode is used to acquire data directly to the SAB bus. In this mode of operation, data bypasses the on-board memory and is passed to the SAB bus via the equivalent of a 48k byte FIFO.

This mode is typically selected to minimize the data latency at the SAB interface. This is important for applications that must make quick decisions in real time based on the signal information. Acquisition sizes of 48k samples or less are also good candidates for use of this mode since the acquisition FIFO is large enough to totally contain all of the data.

To use this mode there must be some device that can capture the data transmitted over the bus, such as a Signatec PMP1000 digital signal processing board. The SAB may be set for either 32-bit or 64 bit operation. When set for 32 bits the data may be output over the low order SAB connector. In 32-bit mode, the maximum acquisition rate is 250 MSPS while in 64-bit mode the maximum rate is 500 MSPS. See section 2.4.7 for more details on SAB bus width.

The AcquireToSabD16 library function may be used to acquire data in this mode.

2.4.4 SAB Buffered Acquisition Mode

This operating mode is for future logic releases and should not be used at this time.

In this mode of operation the entire on-board RAM may be utilized as a giant FIFO to prevent data from being lost due to periodic hold off of the SAB data flow. The maximum acquisition rate for a SAB Buffered acquisition is dependent upon the hardware of the SAB device receiving the data and/or the algorithm execution time for a processing product that is working on the SAB transferred data, as may be the case when connecting Signatec's PMP1000 board to the PDA16 via the SAB. For data acquisitions larger than 256 mebi-samples, if the acquisition rate is higher than the SAB data rate, then it is possible that the on-board FIFOs could overflow and data can be lost. A FIFO-full flag is available to be read at the end of the transfer to check for this condition. The GetFifoFullFlagD16 library function may be used to read this flag.

To use this mode there must be some device that can capture the data transmitted over the bus, such as a PMP1000 board. The SAB may be set for either 32-bit or 64 bit operation. When set for 32 bits the data may be output over the low order SAB connector. See Section 2.4.7 for more details on SAB bus width.

The AcquireToSabD16 library function may be used to acquire data in this mode.

2.4.5 RAM Acquisition Mode

This is the principal data acquisition mode. When the PDA16 is placed in this mode, waveform sampling will commence when the next trigger event is detected. (Triggering is discussed in section 2.7.) The first data sample is stored into signal RAM at the starting sample set by the SetStartSampleD16 function. The PDA16 continues to acquire data and store it to RAM until the total number of samples acquired matches the total samples count, specified by the SetSampleCountD16 function.

When the acquisition is complete and all data has been written into PDA16 RAM, the Samples Complete flag is set and a Samples Complete interrupt is sent to the system. The Samples Complete flag may be read by calling the

GetSamplesCompleteFlagD16 library function. The WaitForAcquisitionCompleteD16 function can be used to wait (sleep) for the Samples Complete interrupt with optional timeout.

The AcquireToBoardRamD16 library function can be used to acquire data in this mode. This function handles the setting up of the active memory region and all operating mode changes.

2.4.6 PCI Transfer Mode

This mode allows PC access to the waveform data in the PDA16 RAM. This mode is rarely explicitly used by user applications. The ReadSampleRamFastD16 or ReadSampleRamD16 library function may be used to transfer data from the PDA16 signal RAM to the PC.

2.4.7 SAB Transfer Mode

This mode is used to send previously acquired data from the PDA16 signal RAM to the Signatec Auxiliary Bus (SAB). Before entering this mode the region of RAM to be transferred (the "active memory" region) must be specified. This is specified via a starting RAM address (see SetStartSampleD16) and total sample count (see SetSampleCountD16). Data transfer will start immediately after the board is set to the SAB Transfer mode. This mode is used to send data to a DSP board, such as Signatec's PMP1000 or to other possible target boards. When the transfer is complete the board's Samples Complete flag is set. This flag may be read using the GetSamplesCompleteFlagD16 function. See Section 2.11.2 for control of the PDA16 for a SAB Bus Controller connected to the SAB.

The PDA16 supports data transfer widths of 64 and 32 bits. For 32-bit operation, data can be output via the low-order SAB port. The SetSabConfigurationD16 library function may be used to specify transfer width configuration.

The PDA16 implements version 4 of the Signatec Auxiliary Bus (see section 2.11). This version is capable of data strobe frequencies of up to 66.6MHz. To insure backward compatibility with earlier versions of the SAB that have a maximum strobe frequency of either 62.5MHz or 25MHz, the strobe frequency is selectable on the PDA16. The SetSabClockD16 library function may be used to select this frequency.

2.4.8 PCI Write RAM Mode

This operating mode is for future logic releases and should not be used at this time.

2.5 Analog Input Circuitry

The PDA16 incorporates two identical analog input channels. The input signals are AC coupled. Eight voltage ranges are implemented for each channel. The upper range is 2.5 volts full scale while the lower range is 267 millivolts FS. Each range is separated by a factor of approximately 1.5. See the specification section for the actual range values. Functions SetVoltRangeCh1D16 and SetVoltRangeCh2D16 can be used to set the range for channel 1 and channel 2 respectively.

The upper two ranges are transformer coupled to the ADC with no amplification and no filtering. These ranges have the highest performance level with respect to SFDR and SNR. The transformer coupled ranges have a higher minimum operating frequency and a lower maximum operating frequency than the amplifier ranges.

The lower four voltages ranges are implemented with multiple high-performance amplifiers and switches. The amplifiers have a bandwidth of greater than 1 GHz. The amplifier outputs are connected to a third order Bessel low pass filter. The purpose of this filter is to reduce the noise bandwidth of the amplifiers to no more than

required to capture the maximum input signal frequency. The PDA16 is offered with no filter, or standard filter cutoffs of 120 MHz or 200 MHz.

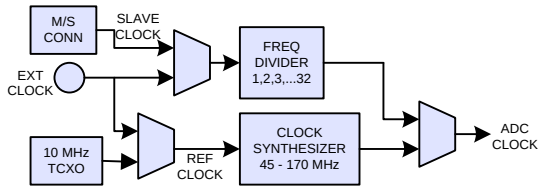
ADC data can be captured in dual channel or single channel mode. In single channel mode the entire signal memory can be used to capture data from channel 1 only.

The Pre-trigger Samples FIFO can be thought of as a programmable length shift register with a maximum length of 4k samples. It is used to capture pre-trigger samples. Before a trigger is received, the digitizers are active and data is continuously written into the shift register. After receiving a trigger, data samples start to be written into the Acquisition FIFO. See the section “Trigger Modes and Options” for trigger mode details.

Data is written into the SDRAM via the Acquisition FIFO and read from RAM via the I/O FIFO. The double-data-rate RAM operates at a clock rate of 200 MHz so it has a bandwidth capability of 1600 MB/s.

2.6 ADC Clocking

An internal synthesized clock is the primary clock source for the ADCs. The ADC clock can also be supplied from the external clock input or from the Slave clock at the Master/Slave connector. The figure below shows the functionality of the ADC clock circuitry.



The synthesizer can generate any frequency from 45 to 128 MHz and most frequencies from 128 to 170 MHz. See the specification section for the range of un-settable frequencies.

If the external clock input is the ADC clock source, it may be divided by any integer value from 1 to 32. For Master/Slave board combinations the slave board(s) derive(s) the ADC clock from the Master/Slave connector via a ribbon cable connection. For slave boards the frequency divider should be set to 1 to match the slave clock to the master clock.

When the synthesized clock is selected, ADC clock jitter is extremely low at about 200 fS RMS. The jitter is independent of the clock divider setting. Clock jitter can reduce the SNR of the captured signal at high frequencies. Due to the low synthesizer clock jitter, degradation does not occur until the signal frequency is typically greater than 100 MHz.

The synthesizer clock is locked to a 10 MHz reference clock. The reference clock may be selected from the internal reference or an externally supplied reference clock. The internal reference clock is accurate to better than 5ppm. This sets the ADC clock accuracy to also be within 5ppm.

For all clock sources the effective digitization rate can be further reduced via sample discarding of the digitized data. This second divider can be set from 2 to 32 in factors of 2.

2.7 Triggering the PDA16

2.7.1 Minimizing Jitter - Synchronized Triggering

Under normal triggering from an asynchronous external source there will be a peak-to-peak jitter of 1 clock cycle between the trigger signal and the captured waveform. The use of a synchronized trigger can eliminate this jitter. There are three methods that can be used.

2.7.1.1 External Clock and Trigger

Supplying an external clock to the PDA16 as well as a trigger that is synchronized to that clock will eliminate all trigger jitter.

2.7.1.2 Synchronizing the Trigger via Clock Out

Via the digital output connector, an output clock may be selected for use by external circuitry to synchronize an input trigger to the PDA16's internal digitizer clock. The clock output frequency will be equal to the digitizer clock frequency. The function SetDigitalOutputModeD16 may be used to set the digital output for clock out operation.

2.7.1.3 Synchronizing the Trigger via Trigger Out

A synchronized output trigger can be obtained via the digital output connector. For this type of triggering an asynchronous trigger is applied to the trigger input and an output trigger, that is synchronous to the digitizer clock, is used externally to drive a pulse source. This type of operation is useful for many types of pulsed systems. The function SetDigitalOutputModeD16 may be used to set the digital output for synchronized trigger operation.

2.7.2 Trigger Modes

Two triggering modes with two triggering options are implemented on the PDA16. The different trigger modes and options determine how and when data will be acquired. The SetTriggerModeD16 function in the PDA16 function library may be used to select the trigger mode. Operation in the various modes is described in the following sections. The PDA16 must be placed into one of the data acquisition modes for the trigger to have any effect.

2.7.2.1 Post-Trigger Mode

Post-Trigger mode is a single-shot mode in which a single trigger signal causes the PDA16 to start acquiring data. In this mode all (or most) of the data that is acquired occurs AFTER the trigger. Data acquisition will continue until the number of samples acquired equals the Total Sample Count setting (function SetSampleCountD16). This is referred to as a single-shot mode because just one trigger is required to start and complete the acquisition. Notification of acquisition complete is indicated by the Samples Complete flag (see GetSamplesCompleteFlagD16) or the Samples Complete interrupt (see WaitForAcquisitionCompleteD16).

2.7.2.2 Segmented Trigger Mode

When Segmented trigger mode is selected, each trigger event causes a fixed portion of the active PDA16 memory to be filled until the entire active memory is filled. The amount of memory that is filled for each trigger is called the segment size. To be meaningful, the segment size must always be smaller than the active memory size. Refer to the function SetSegmentSizeD16 for details concerning the segment sizes available and how they are selected.

The PDA16 is equipped with a Segment Full flag that can be monitored by the application to indicate when a segment has been acquired. This flag may be read by calling the GetSegmentFullFlagD16 function. Monitoring this flag may be useful in conditions where long time periods exist between segment triggers. This provides a means of monitoring acquisition progress.

2.7.3 Trigger Timing Options

In addition to the normal trigger modes described above, two trigger options exist that provide additional capability for capturing data. The trigger options determine the relationship of the stored data samples to the trigger event. Using the normal method as described above, data capture begins on detection of a trigger event. The following trigger options allow for starting the data capture slightly before or after the actual trigger signal.

2.7.3.1 Pre-trigger Samples

This option may be used in Post-Trigger or Segmented trigger modes but not in Pre-Trigger mode. When activated, data from the ADC is routed to a “data pipeline” of programmable length. The digitization and shifting of data through the pipeline is always active (while in the acquisition mode) so that the pipeline is full when a trigger occurs. The pipeline therefore holds pre-trigger data. When the trigger occurs, the output from the pipeline is written to RAM or output to one of the buses, depending on the current acquisition mode.

Data is stored until either the desired total samples is attained or, in the case of segmented trigger mode, the segment size limit is reached. The data written to the RAM (or output to the buses) consists of the defined number of pre-trigger samples immediately preceding the trigger, with the remaining number of samples occurring immediately after the trigger. This mode is useful for seeing "backward in time", i.e. seeing the waveform as it existed before the trigger signal occurred.

The amount of pre-trigger samples can be specified with the SetPreTriggerSamplesD16 function.

2.7.3.2 Delayed Trigger

With delayed triggering, data acquisition will begin a selectable number of digitizer clock cycles after a trigger event occurs. The trigger delay can be set with the SetTriggerDelaySamplesD16 function. Trigger delay and pre-trigger samples are mutually exclusive options. One or the other (or both) is normally set to zero.

2.7.4 Trigger Setup

The external trigger input can be used to synchronize the start of data acquisition with an external event. This is a digital input with TTL signal level. Triggering may be set to occur on either the positive or negative going edge of the signal.

Triggering may also be set to be “internal”. In this case, acquisition may be set to occur based on the amplitude level of either of the two input signals exceeding a programmed trigger level. The triggering threshold is a 16-bit digital value that is compared against the digitized signal. The detection is edge based with either positive or negative excursion being selectable.

2.7.5 Software Trigger

The PDA16 requires a trigger signal to start data acquisition. Besides internal and external triggering as described above, it is also possible to generate an acquisition trigger via software. The software trigger is functional whether internal or external triggering is selected. To generate a software-generated trigger applications call the IssueSoftwareTriggerD16 function.

2.8 Digital IO

The Digital IO SMA connector on the PDA16 bracket can supply a selection of digital signals that may be used to synchronize external events or to monitor internal processes. Among the available signals are the digitizer clock and the synchronized trigger. See the function SetDigitalOutputModeD16 for a list of available signals and how to select them.

The SetDigitalOutputEnableD16 function is used to enable/disable the digital IO port.

2.9 Active Memory Region

The Active Memory region defines the area of PDA16 RAM used for subsequent acquisition or transfers operations and are defined by two parameters: Starting Sample and Sample Count.

2.9.1 Starting Sample

The starting sample setting determines where the first data sample will be stored when acquiring data into RAM or when transferring data into RAM from one of the buses. It also sets the starting sample for reading data from RAM. The SetStartSampleD16 function can be used to set the starting sample.

The starting sample can be any integer multiple of 2048 not greater than 2147481600.

2.9.2 Sample Count

This setting determines the total number of samples that will be acquired or transferred when the board is put into an acquisition or transfer operating mode. The Sample Count is set with the SetSampleCountD16 function.

When the actual number of samples acquired/transferred matches the Sample Count setting the Samples Complete flag is set and a Samples Complete interrupt is sent to the system. The WaitForAcquisitionCompleteD16 function can be used to wait (sleep) for the Samples Complete interrupt with optional timeout.

The Sample Count can be any integer multiple of 2048 not greater than 2147483648.

When using one of the bus acquisition modes it is possible to have the PDA16 acquired data indefinitely. These are called free-run acquisitions and are specified by passing D16_FREE_RUN (0) to SetSampleCountD16.

2.9.3 Segment Size

This setting is only functional in Segmented trigger mode. Segmented trigger mode allows for partitioning the active memory into a number of fixed-size partitions, each of which requires its own trigger signal to fill the memory segment. The Segment Size setting should always be less than the current Total Samples setting to be meaningful. Data acquisition will stop when the Total Samples count is reached, even if the segment is not complete. The function SetSegmentSizeD16 may be used to set the memory segment size.

2.10 PCI Interface

The PDA16 is a 64-bit PCI board compatible with revision 2.2 of the PCI local bus specification (with the exception of support for operation in PCI 32-bit busses) and is "Plug and Play" compatible. The control and monitoring registers are accessed from the PCI bus with 32 bit I/O read/writes. The signal data can be read from or written to the PDA16 with 64-bit bus master direct memory access (DMA) transfers.

2.10.1 Plug and Play Features

The PDA16 contains a non-volatile EEPROM in which has been stored information required by the system for "Plug and Play" configuration. Upon boot-up, the system Plug and Play BIOS will poll each PCI slot in order to identify the boards installed in the system. The system then interrogates each board in order to determine what system resources are required. The system assigns the necessary I/O and memory addresses to each board. The PDA16 requires two address spaces and one memory space.

The serial number of each PDA16 is stored in the on-board EEPROM. This allows user software to distinguish between multiple PDA16 boards installed in a system by reading the serial number value.

Each time the system is powered-up the "Plug and Play" configuration process is carried out. Therefore it is necessary to retrieve the address space used by the board each time a program is started. For more information on Plug and Play operations contact the Signatec web site at http://www.signatec.com/ for the application titled *Plug and Play Operation of the PCI Local Bus*.

2.10.2 DMA Transfers

PDA16 data transfers over the PCI bus are implemented using Direct Memory Access (DMA) transfers. To perform DMA transfers a PCI board must be configured as a bus master. This means that the PCI board must take control of the PCI bus and command the DMA transfer to occur. A DMA transfer is then carried out using the system's DMA controller without intervention from the system processor. This is the only method available for burst transfers on the PCI bus in a standard PC. The PDA16 library, driver, and hardware handle all aspects of doing DMA transfers.

Before any DMA transfer may occur a DMA buffer needs to be allocated. A DMA buffer is a contiguous, non-paged region of memory. To a user application, using a DMA buffer is just like using normal memory, the only difference being that it is allocated and freed by calling PDA16 library functions instead of malloc/free or new/delete. DMA buffers are allocated using the AllocateDmaBufferD16 function and are freed using the FreeDmaBufferD16 function.

The PDA16 becomes the PCI bus master and data is transferred to the PC in bursts. The PDA16 is capable of transferring one 64-bit word on each PCI bus clock cycle. This means the PDA16 is capable of data transfers at a peak rate of 1066 megabytes per second. The sustainable transfer rate realized in a system will be less than this and is dependent upon the system hardware configuration and data traffic from other devices on the PCI bus.

2.11 Signatec Auxiliary Bus Interface

Many Signatec products implement a high-speed bus termed the Signatec Auxiliary Bus, or SAB. The SAB is capable of transfer rates of 500 megabytes per second using a 64 bit bus with a 62.5MHz clock. A data sheet and specification for the bus is available from the Signatec website. The SAB is located at the top of the PDA16 board on the right side. Commands can be sent to the PDA16 via the SAB to control its operation. SAB control is much faster than PCI control since there are typically no bus contention or operating system access issues.

The PDA16 supports data transfer widths of 64 and 32 bits. For 32-bit operation, data can be output via the low-order SAB port.

The SAB allows data to be moved efficiently between boards while bypassing the host computer operating system and PCI bus traffic. Total system performance (in a single computer) is also enhanced by the fact that multiple boards may be interconnected with multiple SAB buses, thus far exceeding what would be possible using only the PCI bus. These issues make the SAB attractive for the most demanding real-time applications.

In all of its normal operational modes the PDA16 is a data “transmitter” over the SAB and not a receiver.

In order to transmit its data, the PDA16 must be set as a SAB “master”. There can be only one active master board on the SAB at a time. When not set to transmit its data the board should be set as an “inactive slave”. The terms master and slave used here pertain to SAB protocols and should not be confused with the same terms used elsewhere in this manual to refer to the operation of multiple PDA16 boards for synchronized acquisition.

A maximum of eight boards can be interconnected on one SAB bus. Pin connections for the SAB on the PDA16 are given in “APPENDIX C - Signatec Auxiliary Bus Connections”.

2.11.1 SAB Data Transfer

This section details SAB data transfer characteristics in all the various modes and configurations. The primary signals involved in the transfer of data are the busy signal (BUSY-) the data strobe (STRB-) and the 32/64 data bits. Refer to the SAB Data Sheet for timing details.

2.11.1.1 SAB Configuration

SAB data can be transferred over the 32-bit half bus (SABL) or over the entire 64-bit bus (SAB). The SAB width configuration is set by the SetSabConfigurationD16 function.

In 64-bit mode the bus control signals from SABL are active and those on SABH are inactive. For 32-bit operation the maximum transfer rate will be one-half that available for 64-bit operation.

2.11.1.2 SAB Acquisition Mode

In SAB Acquisition mode (D16MODE_ACQ_SAB), the SAB WAIT- signal is not acknowledged. This means that data flow out from the PDA16 cannot be paused using this line.

If the digitization rate is less than 500 MSPS (for 64-bit SAB operation) the SAB data flow will be gated (ON/OFF) so as to that the average output data rate is equal to the digitization rate.

It is possible to set the digitization rate to 1GSPS in SAB Buffered Acquisition mode. However, if the total number of samples acquired exceeds about 536 million (again assuming 64-bit operation), the RAM buffer will overflow and data will be lost. It is up to the user to manage this situation.

2.11.1.3 SAB Transfer Mode

In SAB Transfer mode (D16MODE_RAM_READ_SAB), the SAB WAIT- signal is functional. Data flow out from the PDA16 can be paused using this wait line. In SAB Transfer mode, the data strobe frequency will usually be set to 62.5 MHz. This frequency is set using the SetSabClockD16 function.

2.11.2 SAB Commands

The PDA16 is capable of responding to commands issued over the Signatec Auxiliary Bus (SAB) by an external device, such as Signatec's PMP1000 digital signal processor board. Controlling the operation and transferring data over the SAB maximizes the overall data throughput rate in applications requiring multiple acquisition cycles or in systems with a large amount of PCI traffic.

Commands are issued over the SAB with the use of the SAB interrupt lines (INTR1-INTR5), the SAB communication lines (COM1-COM4), and in some cases the lower 32 data lines. The PDA16 is assigned a specific INTR line to which it will respond. This is accomplished by setting the SAB board number. This enables

control of the PDA16 from the SAB when the board number is set to a non-zero value using SetSabBoardNumberD16.

A combination of direct commands, which use only the INTR and COM lines, and indirect command, which also use the data lines, are used for SAB communication. The SAB protocol allows indirect commands to be used to write all PDA16 control registers accessed by the PCI bus for maximum flexibility. The programmer must be aware when using SAB commands that access from the PCI bus is still possible and care must be taken to ensure proper operation when using both interfaces to control the PDA16.

A listing of all SAB commands implemented on the PDA16 is available in the following table. When a PMP1000 is used as the SAB bus controller, the PMP1000 PEP function PepSendSabDirectCmd may be used to issue the command to the PDA16 board.

SAB Command Value	PDA16 Library Constant	Interpretation
15 (0xF)	D16SABCMD_MODE_STANDBY	Enter standby mode
11 (0xB)	D16SABCMD_MODE_ACQ_SAB	Acquire to SAB (unbuffered)
10 (0xA)	D16SABCMD_MODE_ACQ_RAM	Acquire to RAM
13 (0xD)	D16SABCMD_MODE_READ_SAB	Transfer PDA16 RAM data over SAB

2.12 PDA16 Acquisition Data Format

2.12.1 Sample Format

PDA16 data samples are little-endian, unsigned 16-bit integral values.

The PDA16 library uses the 'pda16_sample_t' data type to represent a PDA16 data sample. This data type is an alias that will always equate to an unsigned 16-bit type (usually 'unsigned short' in C/C++ parlance) for the current platform.

Since samples are unsigned, this means that sample value can go from 0 to 65535 (or FFFF in hexadecimal). A sample value of 0 is equivalent to the minimum input voltage. A sample value of 65535 is equivalent to the maximum input voltage range. A sample value of 32768 is equivalent to approximately 0V.

A general formula for converting a sample value to its corresponding input voltage is:

$$V_s = -R/2 + ((S / 65535) * R)$$

Where:

- V_s is the voltage for a particular sample
- R is the input volt range selection (e.g. 2.5, 1.67, 1, 0.667, 0.400, or 0.267) in peak-to-peak voltage
- S is the sample value.

So for a sample value of 8192 @ 1V input voltage range:

$$V_s = -1/2 + ((8192 / 65535) * 1) = -0.375V \text{ or } -375mV$$

2.12.2 Multichannel Data

When acquiring dual-channel data, the PDA16 stores and transfers data in a channel-interleaved format. This means that each channel is alternated in the resultant data:

Ch. 1, Ch. 2, Ch. 1, Ch. 2, Ch. 1, Ch. 2, …

The PDA16 library implements routines to interleave (InterleaveDataD16) or de-interleave (DeInterleaveDataD16) dual-channel data.

2.12.3 PDA16 Sample Data Files (*.rd16)

Signatec software, like the PDA16 Scope Application, as well as certain PDA16 library routines has the ability to save PDA16 acquisition data to a file. The native file format used by these entities is the RD16 file format. The RD16 moniker is derived from "Raw Data 16-bit". RD16 files are identified by the '.rd16' file extension.

RD16 files are binary files that contain only sample data. There is no file header or additional information in the file. The first two bytes of the file are the first data sample. This simple file format has two big advantages. First, it's very fast to write these files since data is written to the file exactly as it is received from the PDA16. If the underlying file system (e.g. a high-speed RAID system) can keep up with the data rate, data can be streamed from the PDA16 card to the file at the full acquisition rate. The second advantage is that this file format is very generic which makes it easy for other software to get at the data. This includes custom software, or other software environments like Matlab or LabVIEW.

2.12.4 Signatec Recorded Data Context Files (*.srdc)

The main disadvantage of the RD16 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- or dual-channel? 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 file. This auxiliary file can contain information like 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 data file, appended with a '.srdc' extension. For example, if acquisition data was being written to the file "C:\Data\Recordings\MyData.rd16" then the auxiliary data would be written to "C:\Data\Recordings\MyData.rd16.srdc". The PDA16 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.

3 PDA16 Software Development Reference

3.1 Getting Started

3.1.1 Contents

The PDA16 is shipped with a disk titled "PDA16 Product Software" that contains a Windows Installer setup used to install PDA16 drivers, libraries, documentation, components, applications, utilities, and programming examples. Some of the more important items are:

PDA16 dynamic link library. This library is the interface to the PDA16 driver. User applications will use the functions exported by this library to connect their software to the PDA16 device. The library (PDA16.dll) is installed to the product installation folder and the system library search path is updated to include this directory.

PDA16 Operator's Manual. This is the document that you are currently reading and is installed to the Documentation\ folder of the install folder.

The PDA16 Scope application. This is a full-featured application that demonstrates how to use the various features of the PDA16. This application is also a great starting point to getting familiar with the PDA16. The source code for this application is installed to the Source\Examples\PDA16 Scope Application\ folder of the install folder.

Various projects demonstrating how to interface with the PDA16 located in the Source\ folder of the install folder.

3.1.2 Installing Software

To install the PDA16 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 the "PDA16 Product Software.msi" Windows installer file found in the root directory of the CD. This installation will install all PDA16 software and documentation to the folder selected during installation.

3.1.3 Developing PDA16 Software

PC applications interface the PDA16 through the PDA16 library. This library exports a variety of functions that you can use in your own programs. These functions are documented in the C Library Functions section. The following sections describe how to set up the library and program the PDA16.

3.1.3.1 Setting up the Build Environment – Windows Platform

The PDA16 library is composed of three parts: the header file (pda16.h), the import library (PDA16.lib), and the dynamic link library (PDA16.dll). The header and import library are required to build PDA16 applications and the dynamic link library is required to run PDA16 applications.

The header file, PDA16.h, is the C interface to the library. Your PDA16 applications will "#include" this file to define PDA16 data structures, function prototypes, constants, and macros. This file is installed to the include\ folder of the PDA16 install directory. It is recommended that you add this path to your compiler's header file search path[1]. Adding the header file to the include file search path will allow you to keep a single copy of PDA16.h on your system and access it like a standard C header file. (That is, you can do an "#include <pda16.h>" instead of an "#include "*Path-to-pda16.h*".)

The import library, PDA16.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 PDA16 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

[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.

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" D16_NO_LINK_LIBRARY before including PDA16.h.)

The dynamic link library, PDA16.dll, is installed to the PDA16 software installation folder (C:\Program Files\Signatec\PDA16 by default). During software installation, the system library search path is updated to include this folder so the PDA16 library can be located when needed.

3.1.3.2 Setting up the Build Environment – Linux Platform

The PDA16 library is composed of two files: the header file (pda16.h) and library binary (libsig_pda16.so). Both files are required to build PDA16 applications. Only the library binary is required to run PDA16 applications.

The header file, pda16.h, is the C interface to the library. Your PDA16 applications will "#include" this file to define PDA16 data structures, function prototypes, constants, and macros. The main PDA16 software build process will put the pda16 library header in the "/usr/local/include" so that it is accessed like a standard C header file. (That is, you can do an "#include <pda16.h>" instead of an "#include "*Path-to-pda16.h*".)

PDA16 applications must be linked with the PDA16 library. This is done by adding the "-lsig_pda16" linker option to the gcc/g++ command line. For an example of this, refer to the Makefile for one of the PDA16 example applications.

3.1.3.3 Library Functions That Use Character Strings

For library functions that accept or generate character strings (e.g. GetErrorTextD16), the PC platform library may actually implement two versions: one version that works with ASCII strings (char*) and another version that works with UNICODE strings (wchar_t*). The ASCII version function names are suffixed with an 'AD16' and the UNICODE version function names are suffixed with 'WD16'.

Macros have been defined so that library users can write generic code that will work with either version. Consider GetErrorTextD16 as an example. The library implements two versions of this function: GetErrorTextAD16 and GetErrorTextWD16. If the _UNICODE symbol is defined, then the GetErrorTextD16 macro will expand to GetErrorTextWD16, else it will default to GetErrorTextAD16. In this manual library functions are documented using the character-size agnostic type TCHAR.

```
int GetErrorTextAD16 (int res, char** bufpp, unsigned int flags, HPDA16 hBrd);
int GetErrorTextWD16 (int res, wchar_t** bufpp, unsigned int flags, HPDA16 hBrd);

#ifdef _UNICODE
# define GetErrorTextD16        GetErrorTextWD16
#else
# define GetErrorTextD16        GetErrorTextAD16
#endif

// Effective library function prototype:
int GetErrorTextD16 (int res, TCHAR** bufpp, unsigned int flags, HPDA16 hBrd);
```

3.2 Library Utility Functions

The functions in this section are generic library utility functions.

3.2.1 GetErrorTextD16

Form

```
int GetErrorTextD16 (int res, TCHAR** bufpp, unsigned int flags, HPDA16 hBrd =
INVALID_PDA16_HANDLE);
```

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 (PDA16-specific) SIG_D16_* error value.

[out] *bufpp*

A pointer to a char pointer that will receive the address of the library allocated buffer containing the error text string. This buffer must be freed by caller by calling the FreeMemoryD16 library function

[in] *flags*

A set of flags that dictate function behavior.

Flag	Interpretation
D16ETF_IGNORE_SYSERROR (0x00000001)	Ignore system specific error information (Windows: GetLastError, Linux: errno)
D16ETF_NO_SYSERROR_TEXT (0x00000002)	Do not generate any text for system specific error
D16ETF_FORCE_SYSERROR (0x00000004)	Force inclusion of system error information even if it might not be relevant

[in] *hBrd*

A handle to the PDA16 device for which the error occured. For some types of errors, the PDA16 handle used when the error was generated may contain additional context information. This parameter may be INVALID_PDA16_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 ambiguous (and certain known) error values, information on the current (thread-specific) system error state is also provided.

Related Functions

DisconnectFromDeviceD16, ConnectToVirtualDeviceD16

3.3 PDA16 Device Enumeration and Connection Management

The functions in this section involve managing PDA16 device connections and general PDA16 device handle management.

3.3.1 ConnectToDeviceD16

Form

```
int ConnectToDeviceD16 (HPDA16* phDev, unsigned int brdNum);
```

Description

This function is used to establish a connection to a PDA16 data acquisition device, represented by a PDA16 device handle of type HPDA16.

Parameters

[in] *pBrd*

A pointer to a HPDA16 variable that will receive the PDA16 device handle. This PDA16 device handle is only valid in the current process. This handle should be treated as an opaque object; interpretation of the handle is PDA16 library specific.

[in] *brdNum*

This parameter is used to select which PDA16 in the system to connect to. If this value is in the range [1, D16_MAX_DEVICES (16)] 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 PCI(-X) slots. If this value is outside the aforementioned range then it is assumed to be the PDA16 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 PDA16 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 PDA16 device in a non-intrusive manner.

Like other handle-based mechanisms, the actual value of a PDA16 device handle should be treated as an opaque value. The exception to this is a special value, INVALID_PDA16_HANDLE, which is used to identify an invalid PDA16 device handle.

An application should close the connection to the PDA16 device by calling the DisconnectFromDeviceD16 library function. Failure to disconnect from the device can result in memory leaks in the calling process.

Related Functions

DisconnectFromDeviceD16, ConnectToVirtualDeviceD16, ConnectToRemoteDeviceD16

3.3.2 ConnectToVirtualDeviceD16

Form

```
int ConnectToVirtualDeviceD16 (HPDA16* phBrd, unsigned int serialNum, unsigned int brdNum);
```

Description

Establish a connection to a virtual (fake) PDA16 device

Parameters

[in] *pBrd*
A pointer to a HPDA16 variable that will receive the virtual PDA16 device handle.

[in] *serialNum*
The serial number to use for the virtual device. This can be any number; the PDA16 library ignores this value.

[in] *brdNum*
The board number to use for the virtual device. This can be any number; the PDA16 library ignores this value.

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 PDA16 data acquisition device. A virtual device is one that is not connected to any real PDA16 hardware. Virtual devices are mainly used for software development and debugging.

Most all PDA16 library functions will work with virtual devices. Data acquisition and transfer operations are not virtualized; attempting to do so with a virtual device results in the library just returning SIG_SUCCESS. DMA buffer allocation is virtualized; normal heap-allocated memory is allocated in this case however.

Related Functions

DisconnectFromDeviceD16, ConnectToDeviceD16

3.3.3 DuplicateHandleD16

Form

```
int DuplicateHandleD16 (HPDA16 hBrd, HPDA16* phNew);
```

Description

Duplicate a PDA16 device handle

Parameters

[in] *hBrd*
The PDA16 device handle to duplicate. This handle must have been obtained from the current process.

[out] *phNew*
A pointer to a HPDA16 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.

Related Functions

ConnectToDeviceD16

3.3.4 DisconnectFromDeviceD16

Form

```
int DisconnectFromDeviceD16 (HPDA16 hBrd);
```

Description

This function is used to release the handle for the PDA16 when it is no longer needed in the program. The function also frees any utility DMA buffers allocated by the library for the given PDA16 device.

Parameters

[in] *hBrd*
The PDA16 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 PDA16 device handle will have no effect on the underlying hardware. This allows one to connect/disconnect to a PDA16 device in an unobtrusive manner. A slight exception to this is that the PDA16 driver may interact with the PDA16 hardware when all connections to a particular PDA16 (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

ConnectToDeviceD16

3.3.5 GetDeviceCountD16

Form

```
int GetDeviceCountD16();
```

Description

Obtain a count of all PDA16 devices present in the local system

Return Value

Returns the number of physical PDA16 devices present in the local system. If this function returns zero then either no PDA16 devices are present or the PDA16 driver has not been installed.

3.3.6 IsDeviceRemoteD16

Form

```
int IsDeviceRemoteD16 (HPDA16 hBrd);
```

Description

Determines if a given PDA16 device handle is for a remote device

Parameters

[in] *hBrd*
The PDA16 device handle to check. The ConnectToRemoteDeviceD16 function is used to connect to a remote PDA16 device.

Return Value

Returns a positive value if the given handle is connected to a remote PDA16, zero if the handle is not connected to a remote PDA16, or a negative library error code on error.

Related Functions

IsDeviceVirtualD16, IsHandleValidD16

3.3.7 IsDeviceVirtualD16

Form

```
int IsDeviceVirtualD16 (HPDA16 hBrd);
```

Description

Determines if a given PDA16 device handle is for a virtual device

Parameters

[in] *hBrd*
The PDA16 device handle to check. A PDA16 device handle is obtained by calling the ConnectToDeviceD16 or ConnectToVirtualDeviceD16 function.

Return Value

Returns a positive value if the given handle is connected to a virtual device, zero if the handle is not connected to a virtual device, or a negative library error code on error.

Remarks

Use this function to determine if the given handle is associated with a virtual PDA16 device. A virtual PDA16 is not connected to real PDA16 hardware and is mainly used for software development and debugging.

Related Functions

IsHandleValidD16, IsDeviceRemoteD16

3.3.8 IsHandleValidD16

Form

```
int IsHandleValidD16 (HPDA16 hBrd);
```

Description

Determines if the given PDA16 device handle is connected to a device

Parameters

[in] *hBrd*
The PDA16 device handle to check. A PDA16 device handle is obtained by calling the ConnectToDeviceD16 or ConnectToVirtualDeviceD16 function.

Return Value

Returns a positive value if the given handle is connected to a valid device, zero if the handle is not connected to a valid device, or a negative library error code on error.

Remarks

A handle is valid if it is connected to a local, remote, or virtual PDA16 device.

Related Functions

IsDeviceVirtualD16, IsDeviceRemoteD16

3.4 PDA16 Device State and Configuration

The functions in this section are used to obtain state and configuration information on the PDA16.

3.4.1 GetActualAdcAcqRateD16

Form

```
int GetActualAdcAcqRateD16 (HPDA16 hBrd, double* pRateMHz);
```

Description

Obtain the actual ADC acquisition rate

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *pRate*
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.

Remarks

This routine is used to obtain the actual ADC acquisition rate taking the current ADC clock source and any clock dividers into consideration.

This routine does not consider post-ADC clock division. For the effective acquisition rate, which does take post-ADC division into consideration, use the GetEffectiveAcqRateD16 function.

Related Functions

GetEffectiveAcqRateD16

3.4.2 GetEffectiveAcqRateD16

Form

```
int GetEffectiveAcqRateD16 (HPDA16 hBrd, double* pRateMHz);
```

Description

Obtain effective acquisition rate considering post-ADC division

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *pRate*
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

GetActualAdcAcqRateD16, SetPostAdcClockDividerD16

3.4.3 GetFifoFullFlagD16

Form

```
int GetFifoFullFlagD16 (HPDA16 hBrd);
```

Description

Get status of the RAM FIFO full flag; use with RAM-buffered acquisitions

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

Returns 1 if an internal FIFO has overflowed at some point during the acquisition, 0 if no FIFOs have overflowed or one of the library error codes (which are all negative) on error.

Remarks

For buffered PCI acquisitions, it is not necessary to call this routine. The PDA16 library/driver automatically checks the status of this bit after each DMA transfer and notifies of FIFO overflow by returning the SIG_D16_FIFO_OVERFLOW error code when GetPciAcquisitionDataD16 is called.

The function GetFifoFullFlagD16 checks the status of the FIFO Full Flag that indicates whether either of the two on-board first-in-first-out (FIFO) data buffers has become full during an acquisition to the PCI or SAB buses or whether the RAM itself has overflowed in either of the buffered acquisition modes.

This flag is latched so that once an overflow condition is detected it will remain set until a transition into an acquisition mode from standby mode.

The possibility of data being lost is greatly reduced if the on-board RAM is used for buffering during an SAB or PCI acquisition. To do this use one of the buffered acquisition modes, D16MODE_ACQ_PCI_BUF or D16MODE_ACQ_SAB_BUF.

3.4.4 GetFirmwareVersionD16

Form

```
int GetFirmwareVersionD16 (HPDA16 hBrd, unsigned long long* verp);
```

Description

Obtains the version of the PDA16 firmware

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[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

The PDA16 firmware version is 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

Related Functions

GetHardwareRevisionD16

3.4.5 GetHardwareRevisionD16

Form

```
int GetHardwareRevisionD16 (HPDA16 hBrd, unsigned long long* verp);
```

Description

Obtains the revision of the PDA16 hardware

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[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

The PDA16 hardware revision is 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

Related Functions

GetFirmwareVersionD16

3.4.6 GetOrdinalNumberD16

Form

```
int GetOrdinalNumberD16 (HPDA16 hBrd, unsigned int* onp);
```

Description

Obtain the ordinal number of the PDA16 connected to the given handle

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[out] *onp*
A pointer to the variable that will receive the PDA16'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 PDA16's ordinal number is the number of the PDA16 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 PCI slots.

Related Functions

GetSerialNumberD16

3.4.7 GetSamplesCompleteFlagD16

Form

```
int GetSamplesCompleteFlagD16 (HPDA16 hBrd);
```

Description

Get status of the samples complete flag; set when a RAM or SAB acquisition completes

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

Returns 1 if the Samples Complete Flag is set (indicating acquisition complete), 0 if the flag is not set (indicating that acquisition has not completed), or one of the library error codes (which are all negative) on error.

Remarks

This function checks the status of the PDA16 Samples Complete flag (SCF) and returns the flag status. A value of 1 indicates that the samples count is complete and a value of 0 indicates that the transfer/acquisition is not yet complete. The Samples Complete flag is meaningful both for detecting the end of data acquisition and also for detecting the end of data transfer.

This function is only necessary for "manual" data acquisitions in which the code explicitly handles all mode changes.

3.4.8 GetSerialNumberD16

Form

```
int GetSerialNumberD16 (HPDA16 hBrd, unsigned int* snp);
```

Description

Obtain the PDA16 board's serial number

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[out] *snp*
A pointer to the variable that will receive the PDA16's serial number

Return Value

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

Related Functions

GetOrdinalNumberD16

3.4.9 ReadConfigEepromD16

Form

```
int ReadConfigEepromD16 (HPDA16 hBrd, unsigned int eeprom_addr, unsigned short* eeprom_datap);
```

Description

Read an element from the PDA16 configuration EEPROM

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] eeprom_addr
The EEPROM address that to be read. Valid addresses that user applications may use are within the range [0x80, 0xFF]. All values below address 0x80 are reserved for internal use and are read and write protected by the library.

[out] *eeprom_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 PDA16 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.

If the caller attempts to read outside of the range set aside for user-defined data the function will return SIG_D16_CFG_EEPROM_ACCESS_DENIED.

Related Functions

WriteConfigEepromD16

3.4.10 WriteConfigEepromD16

Form

```
int WriteConfigEepromD16 (HPDA16 hBrd, unsigned int eeprom_addr, unsigned short eeprom_data);
```

Description

Write an element from the PDA16 configuration EEPROM

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *eeprom_addr*
The EEPROM address to be written. Valid addresses that user applications may use are within the range [0x80, 0xFF]. All values below address 0x80 are reserved for internal use and are read and write protected by the library.

[in] *eeprom_datal*
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.

Remarks

If the caller attempts to write outside of the range set aside for user-defined data the function will return SIG_D16_CFG_EEPROM_ACCESS_DENIED.

Related Functions

ReadConfigEepromD16

3.5 PDA16 Hardware Settings

The functions in this section control the getting and setting of the various PDA16 hardware settings. These setting include the numerous data acquisition, clock, and trigger parameters.

3.5.1 IssueSoftwareTriggerD16

Form

```
int IssueSoftwareTriggerD16 (HPDA16 hBrd);
```

Description

Issue a software-generated trigger event to a PDA16

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 IssueSoftwareTriggerD16 allows the user to issue a software trigger to the PDA16 board. After a software trigger has been issued, the next (or current) data acquisition will begin immediately and will not wait for a physical trigger event to occur.

Related Functions

SetTriggerLevelAD16, SetTriggerModeD16, SetTriggerSourceD16

3.5.2 SetActiveChannelsD16

Form

int SetActiveChannelsD16 (HPDA16 hBrd, unsigned int val);
int GetActiveChannelsD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the active channel selection; defines which channels are digitized

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Selects which channels will be active for the next data acquisition. This parameter may be any of the following:

PDA16 Library Constant	Interpretation
D16CHANNEL_DUAL (0)	Channel 1 and Channel 2
D16CHANNEL_ONE (1)	Single Channel (Channel 1 only)

Return Value

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

On success, GetActiveChannelsD16 will return the current active channel selection.

3.5.3 SetAdcClockReferenceD16

Form

int SetAdcClockReferenceD16 (HPDA16 hBrd, unsigned int val);
int GetAdcClockReferenceD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the internal ADC clock reference

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The identifier of the ADC clock reference to set. Can be any of the following:

PDA16 Library Constant	Interpretation
D16CLKREF_INT_10MHZ (0)	Internal 10MHz clock reference (Power-up default)
D16CLKREF_EXT (1)	External clock reference

Return Value

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

On success, GetAdcClockReferenceD16 will return the current ADC clock reference selection.

Related Functions

SetAdcClockSourceD16

3.5.4 SetAdcClockSourceD16

Form

int SetAdcClockSourceD16 (HPDA16 hBrd, unsigned int val);
int GetAdcClockSourceD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the PDA16's source ADC clock setting

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The identifier of the ADC clock source to set. Can be any of the following:

PDA16 Library Constant	Interpretation
D16CLKSRC_INT_VCO (0)	Internal 1.5GHz voltage-controlled oscillator (VCO)
D16CLKSRC_EXTERNAL (1)	External clock supplied on the PDA16's external clock input

Return Value

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

On success, GetAdcClockSourceD16 will return the current ADC clock source selection.

Remarks

Do not change the clock source or frequency while an acquisition is in progress.

Internal 1.5GHz Voltage-Controlled Oscillator

When this clock source is selected, the acquisition rate is defined by the SetInternalClockRateD16 function. When this clock source is selected, user-specified clock divider values are ignored. (The clock dividers are used internally to setup acquisition rate.)

External clock

When this clock source is used, the SetExternalClockRateD16 function should be used to specify the external clock rate so the software can properly track things like effective acquisition rate, calculate FFT statistics, etc.

Related Functions

SetPostAdcClockDividerD16, SetInternalClockRateD16, SetClockDividersD16, SetExternalClockRateD16

3.5.5 SetAdcDitheringEnableD16

Form

int SetAdcDitheringEnableD16 (HPDA16 hBrd, unsigned int bEnable);
int GetAdcDitheringEnableD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Enable/disable ADC dithering; may reduce low signal amp frequency spurs

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *bEnable*
If this parameter is zero then ADC dithering will be enabled, else it 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, GetAdcDitheringEnableD16 returns the current ADC dithering enable setting.

3.5.6 SetAdcRandomizedOutputDataEnableD16

Form

int SetAdcRandomizedOutputDataEnableD16 (HPDA16 hBrd, unsigned int bEnable);
int GetAdcRandomizedOutputDataEnableD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Enable/disable ADC randomized output data; may reduce amplitude of frequency spurs

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *bEnable*
If this parameter is zero then ADC dithering will be enabled, else it 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, GetAdcRandomizedOutputDataEnableD16 returns the current ADC randomized output data setting.

3.5.7 SetAddressCounterAutoResetOverrideD16

Form

int SetAddressCounterAutoResetOverrideD16 (HPDA16 hBrd, int bEnable);
int GetAddressCounterAutoResetOverrideD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Enable/disable the auto address counter reset override feature

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *bEnable*

If this parameter is zero then the address counter auto-reset override will be enabled, else it 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, GetAddressCounterAutoResetOverrideD16 returns the current address counter auto-reset override setting.

Remarks

Normally, whenever the PDA16 is placed into Standby mode the RAM address counter is reset to 0. When enabled, this feature could be used to start a new acquisition at the next address after the previous acquisition. Normally the address counter value is set by the function SetStartSampleD16.

Related Functions

SetStartSampleD16

3.5.8 SetBoardProcessingEnableD16

Form

int SetBoardProcessingEnableD16 (HPDA16 hBrd, unsigned int bEnable);
int GetBoardProcessingEnableD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Enable/disable PDA16 FPGA processing

Parameters

[in] *hBrd*

A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *bEnable*

If this parameter is non-zero then PDA16 FPGA processing will be enabled. If this parameter is zero then PDA16 FPGA processing will be disabled. This setting only has effect if the PDA16 supports custom FPGA processing.

Return Value

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

On success, GetBoardProcessingEnableD16 returns the current board processing enable setting.

Remarks

SetBoardProcessingEnableD16 is used to enable PDA16 FPGA processing. Not all PDA16 devices support this feature. FPGA processing a generic term applied to custom or user-defined logic running on one the PDA16's FPGAs. Some examples of FFT board processing include: FFT operations or digital down conversion.

The actual processing that is enabled is dependent on the underlying firmware, which may be user-defined.

Depending on the underlying firmware, additional configuration may be required to properly configure board processing. Consult specific FPGA processing firmware project documentation for additional details.

Related Functions

SetBoardProcessingParamD16

3.5.9 SetBoardProcessingParamD16

Form

```
int SetBoardProcessingParamD16 (HPDA16 hBrd, unsigned short idx, unsigned short value);
```

Description

Sets a PDA16 FPGA processing parameter

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *idx*
The index of the PDA16 FPGA processing parameter to write. This can be any value in the range [0, 65535].

[in] *value*
The FPGA processing value to write. This can be any value in the range [0, 65535]. Interpretation of this value is dependent on the underlying firmware.

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 specify PDA16 FPGA processing parameter data. This is a generic interface that allows the host PC to specify arbitrary arguments to custom PDA16 FPGA processing firmware. These parameters might include things like FFT sizes, FFT window data, or frequency specification for example.

Actual location and interpretation of the parameter data is dependent on the actual underlying firmware. This firmware will either be custom firmware generated by Signatec or by an end-user. Custom FPGA processing firmware created by Signatec will include documentation that defines the details of any processing parameters.

End-users developing their own custom firmware will know the details of any parameters because they are the ones defining them.

Related Functions

SetBoardProcessingEnableD16

3.5.10 SetClockDivider*D16

Form

int SetClockDividersD16 (HPDA16 hBrd, unsigned int div1, unsigned int div2);
int SetClockDivider1D16 (HPDA16 hBrd, unsigned int div1);
int GetClockDivider1D16 (HPDA16 hBrd, int bFromCache = 1);
int SetClockDivider2D16 (HPDA16 hBrd, unsigned int div1);
int GetClockDivider2D16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set clock divider values; applies to internal VCXO and external clock

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *div1*
The divider to use for clock divider #1. This can be any value from 1 to 32.

[in] *div2*
The divider to use for clock divider #1. This can be any value from 1 to 32.

Return Value

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

On success, GetClockDivider1D16 and returns the current clock divider #1 setting.
On success, GetClockDivider2D16 and returns the current clock divider #2 setting.

Remarks

The PDA16 has two clock dividers that operate in series. Each clock divider can take a value between 1 and 32 for up to 354 unique clock divisions ranging from 1 to 1024.

These clock divider values are ignored when the internal VCO is selected as the current clock source.

Related Functions

SetAdcClockSourceD16

3.5.11 SetDigitalOutputEnableD16

Form

int SetDigitalOutputEnableD16 (HPDA16 hBrd, int bEnable);
int GetDigitalOutputEnableD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Enable/disable the PDA16's digital output

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *bEnabled*
If this parameter is nonzero then the PDA16 digital output will be enabled. If this parameter is zero then digital output will be disabled. The actual digital output data is determined by the SetDigitalOutputModeD16 function.

Return Value

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

On success, GetDigitalOutputEnableD16 returns the current digital output enable setting.

Related Functions

SetDigitalOutputModeD16

3.5.12 SetDigitalOutputModeD16

Form

int SetDigitalOutputModeD16 (HPDA16 hBrd, unsigned int val);
int GetDigitalOutputModeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the digital output mode; determines what is driven on PDA16 digital output

Parameters

[in] *hBrd*

A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Selects one of eight possible signals according to the following table:

PDA16 Library Constant	Digital Input/Output	Interpretation
D16DIGOUT_TTL_LOW (0)	Output	TTL low level (0V)
D16DIGOUT_SYNC_TRIG (1)	Output	Synchronized Trigger
D16DIGOUT_ADC_CLOCK (2)	Output	ADC Clock
D16DIGOUT_3_3V (3)	Output	3.3V
D16DIGOUT_INPUT_TS_GEN (4)	Input	Digital input pulse requests a timestamp
D16DIGOUT_RESERVED_5 (5)	Assumed output	Reserved
D16DIGOUT_RESERVED_6 (6)	Assumed output	Reserved
D16DIGOUT_RESERVED_7 (7)	Assumed output	Reserved

Return Value

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

On success, GetDigitalOutputModeD16 returns the current digital output mode setting.

Remarks

This setting is used to define what gets driven on the PDA16 digital output. This setting is only applicable when the digital output is enabled, which is done by calling the SetDigitalOutputEnableD16 function.

Related Functions

SetDigitalOutputEnableD16

3.5.13 SetExternalClockRateD16

Form

```
int SetExternalClockRateD16 (HPDA16 hBrd, double dRateMHz);
int GetExternalClockRateD16 (HPDA16 hBrd, double* ratep, int bFromCache = 1);
```

Get or set the assumed external clock rate in MHz

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *dRateMHz*
The assumed external clock rate in MHz. An error will be returned if this is not a well-formed finite value. The is value should be in the range: [1, 160].

[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

The assumed external clock rate has no direct effect on the underlying PDA16 hardware. It exists solely as a convenience for the applications programming. The updating of this setting is up to the end user since the software cannot know the external clock rate.

Though there is no direct hardware effect of this setting, it is important to keep the PDA16 software up-to-date with the external clock rate. When a change in the acquisition clock frequency is made, the PDA16 firmware needs to be resynchronized with this new frequency. Ensuring this setting guarantees that this synchronization takes place.

The GetEffectiveClockRateD16 function implementation will use this assumed rate when calculating the effective clock rate when the external clock is selected.

Related Functions

SetAdcClockSourceD16

3.5.14 SetInternalClockRateD16

Form

int SetInternalClockRateD16 (HPDA16 hBrd, double dRateMHz);
int GetInternalClockRateD16 (HPDA16 hBrd, double* ratep, int bFromCache = 1);

Get or set the ADC clock rate; only applies to internal 1.5GHz VCO

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *dRateMHz*

The clock rate (i.e. sampling rate), in MHz, to use when the 1.5GHz VCO is selected as the source clock. This value must be in the range of [45, 160]. There are some "hole" frequencies that cannot be reached, see Remarks. An error will be returned if this is not a well-formed finite value.

[out] *ratep*

A pointer to the variable that will receive the current internal clock rate when the 1.5GHz VCO is selected as the ADC clock source.

Return Value

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

Remarks

The 1.5GHz VCO ADC clock source is a voltage controlled oscillator. This means that by varying a voltage input the oscillator can change its frequency. This, in conjunction with onboard clock dividers and a phase lock loop (PLL), allow for a wide range of possible acquisition rates.

There are some "hole" frequencies that cannot be reached:

- 154.70 through 158.66 MHz
- 140.64 through 142.80 MHz
- 128.92 through 129.82 MHz

Related Functions

SetAdcClockSourceD16

3.5.15 SetMasterSlaveConfigurationD16

Form

int SetMasterSlaveConfigurationD16 (HPDA16 hBrd, unsigned int val);
int GetMasterSlaveConfigurationD16 (HPDA16 hBrd, int bFromCache = 1);

Get or set master/slave configuration; slaves are clock/trigger synchronized with master

Parameters

[in] *hBrd*

A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

Selects the PDA16's master/slave configuration. Can be any of the following values:

PDA16 Library Constant	Interpretation
D16MSCFG_NORMAL (0)	Normal, standalone PDA16 (Power-up default)
D16MSCFG_MASTER (1)	Master PDA16; will provide clock and trigger for

	slave PDA16 boards
D16MSCFG_SLAVE (2)	Slave PDA16; clock and trigger will be synchronized to master PDA16

Return Value

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

On success, GetMasterSlaveConfigurationD16 returns the current master/slave configuration setting.

Remarks

It is up to the end user to ensure that two PDA16 boards that are connected via master/slave cable are not both configured as masters. In this situation, both boards will be driving a common net which can damage the board.

Slave PDA16 devices should always use the external clock source and no clock division. This library function will automatically ensure these setting when configuring a board as a slave.

3.5.16 SetOperatingModeD16

Form

int SetOperatingModeD16 (HPDA16 hBrd, unsigned int val);
int GetOperatingModeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the PDA16's operating mode

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The operating mode to switch to. This can be any of the following values:

PDA16 Library Constant	Interpretation
0	Reserved for future use
D16MODE_STANDBY (1)	Standby (ready) mode
D16MODE_ACQ_RAM (2)	RAM acquisition mode.
D16MODE_ACQ_PCI_SMALL_FIFO (3)	This is an older, deprecated legacy operating mode that has been superseded with the RAM-buffer PCI acquisition mode. Signatec recommends that D16MODE_ACQ_PCI_BUF be used instead of this operating mode.
D16MODE_ACQ_PCI_BUF (4)	RAM-buffered PCI acquisition mode; data buffered

	through PDA16 RAM. Never set this mode directly; instead use BeginBufferedPciAcquisitionD16 library function.
D16MODE_ACQ_SAB (5)	Signatec Auxiliary Bus (SAB) acquisition; acquire directly to SAB bus
D16MODE_ACQ_SAB_BUF (6)	Buffered SAB acquisition; acquire to SAB, buffering through PDA16 RAM
D16MODE_RAM_READ_PCI (7)	PDA16 RAM read to PCI; transfer data from PDA16 RAM to host PC. Never set this mode directly; instead use ReadSampleRamFastD16 or ReadSampleRamD16 library function.
D16MODE_RAM_READ_SAB (8)	PDA16 RAM read to SAB; transfer data from PDA16 RAM to SAB bus
D16MODE_RAM_WRITE_PCI (9)	PDA16 RAM write from PCI; transfer data from host PC to PDA16 RAM. Never set this mode directly; instead use WriteSampleRamFastD16or WriteSampleRamD16 library function.

Return Value

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

On success, GetOperatingModeD16 returns the current operating mode setting.

Remarks

For most all active operating modes the PDA16 library implements higher level routines that automatically manage the operating mode. These higher-level routines are shown in the following table.

Underlying Operating Mode	Wrapper Function(s)
D16MODE_ACQ_RAM	AcquireToBoardRamD16
D16MODE_ACQ_PCI_SMALL_FIFO	
D16MODE_ACQ_PCI_BUF	BeginBufferedPciAcquisitionD16
D16MODE_ACQ_SAB	AcquireToSabD16
D16MODE_ACQ_SAB_BUF	AcquireToSabD16
D16MODE_RAM_READ_PCI	ReadSampleRamFastD16 ReadSampleRamD16
D16MODE_RAM_READ_SAB	TransferSampleRamToSabD16
D16MODE_RAM_WRITE_PCI	WriteSampleRamFastD16 WriteSampleRamD16

3.5.17 SetPostAdcClockDividerD16

Form

int SetPostAdcClockDividerD16 (HPDA16 hBrd, unsigned int div);
int GetPostAdcClockDividerD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Set the PDA16's post-ADC clock divider; effective clock division by dropping samples

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *div*
The post-ADC clock divider value to set. Can be any of the following value:

PDA16 Library Constant	Effective Clock Divider
D16POSTADCCLKDIV_01 (0)	1
D16POSTADCCLKDIV_02 (1)	2
D16POSTADCCLKDIV_04 (2)	4
D16POSTADCCLKDIV_08 (3)	8
D16POSTADCCLKDIV_16 (4)	16
D16POSTADCCLKDIV_32 (5)	32

Return Value

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

On success, GetPostAdcClockDividerD16 returns the current post-ADC clock divider setting.

Related Functions

SetAdcClockSourceD16, SetInternalClockRateD16, SetExternalClockRateD16

3.5.18 SetPreTriggerSamplesD16

Form

int SetPreTriggerSamplesD16 (HPDA16 hBrd, unsigned int val);
int GetPreTriggerSamplesD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set pre-trigger sample count; count of samples to keep prior to trigger event

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

The number of pre-trigger samples to set. For dual channel acquisitions, this can be any even number not greater than 8000. For single channel acquisitions, this can be any even number not greater than 4000. The function will clip this value at the maximum allowed value if necessary. If this value is 0 then no pre-trigger samples are collected.

Return Value

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

On success, GetPreTriggerSamplesD16 returns the current pre-trigger sample count setting.

Remarks

Pre-trigger samples can be collected in both Post Trigger and Segmented triggering modes.

The pre-trigger samples setting and delay trigger samples setting are mutually exclusive. Using both at the same time will result in undefined behavior.

Related Functions

SetTriggerDelaySamplesD16

3.5.19 SetSabBoardNumberD16

Form

int SetSabBoardNumberD16 (HPDA16 hBrd, unsigned int val);
int GetSabBoardNumberD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set SAB board number; uniquely identifies a board on Signatec Auxiliary Bus (SAB)

Parameters

[in] *hBrd*

A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

Specifies the board's SAB board number. This can be any number from 0 to 8 (inclusive). Setting this parameter to 0 will disable all SAB control. Each board on the SAB must have a unique SAB board number.

Return Value

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

On success, GetSabBoardNumberD16 returns the current SAB board number setting.

Remarks

This function allows the user to select the value of the SAB interrupt selection. This defines which SAB interrupt will be used by the PDA16 board for control and signaling on the SAB. Each board on the same SAB must have a unique board number.

When doing a SAB acquisition or transfer the PDA16 will signal the SAB Controller that an acquisition is complete by issuing an interrupt on the INTR line corresponding to the board number. This function must be called with a board number other than 0 before all other SAB functions are called.

Related Functions

SetSabClockD16, SetSabConfigurationD16

3.5.20 SetSabClockD16

Form

int SetSabClockD16 (HPDA16 hBrd, unsigned int val);
int GetSabClockD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set SAB clock; governs rate when writing to SAB bus

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The SAB clock selection to set. May be one of the following values:

PDA16 Library Constant	Interpretation
D16SABCLK_125MHZ (0)	125 MHz clock (Power-up default)
D16SABCLK_62_5MHZ (1)	62.5 MHz clock
D16SABCLK_ACQ_CLOCK (2)	Use acquisition clock

Return Value

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

On success, GetSabClockD16 returns the current SAB clock setting.

Remarks

This function is used to set the source clock frequency for data transfers (or acquisitions) across the SAB.

For SAB data transfers (transferring data in PDA16 sample RAM to another device on the SAB), use either the 125 or 62.5 MHz clock. Check with the receiving device documentation to see if it is capable of receiving data at the desired rate.

For SAB acquisitions, use the D16SABCLK_ACQ_CLOCK selection.

Related Functions

SetSabBoardNumberD16, SetSabConfigurationD16

3.5.21 SetSabConfigurationD16

Form

int SetSabConfigurationD16 (HPDA16 hBrd, unsigned int val);
int GetSabConfigurationD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set SAB configuration; defines which bus lines are used

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The SAB acquisition/transfer configuration to use. This parameter can be one of the following.

PDA16 Library Constant	Interpretation
D16SABCFG_64 (0)	64-bit mode
D16SABCFG_32 (1)	32-bit mode using low order data port SABL
D16SABCFG_32L (2)	32-bit mode using high order data port SABH

Return Value

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

On success, GetSabConfigurationD16 returns the current SAB configuration setting.

Remarks

This function is used to define the configuration of SAB transfers. The width of the bus may be selected as 64-bit or 32-bit. Additionally, if 32-bit is selected, either the low data port or high data port may be selected.

Related Functions

SetSabBoardNumberD16, SetSabClockD16

3.5.22 SetSegmentSizeD16

Form

int SetSegmentSizeD16 (HPDA16 hBrd, unsigned int segSize);
int GetSegmentSizeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the PDA16's segment size for use with Segmented triggering mode

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] segSize
The segment size, in samples, to set. This should be an even value.

Return Value

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

On success, GetSegmentSizeD16 returns the current segment size setting.

Remarks

This setting has no effect when the PDA16 is not configured to use Segmented trigger mode.

This function selects the amount of signal memory that will be assigned to each data segment when the board is acquiring in segmented trigger mode. For 2-channel acquisitions the segment size represents the total number of samples acquired for both channels combined. When set to single channel mode the segment size represents the number of samples acquired on channel 1 only.

When the PDA16 board is set to the data acquisition mode and a trigger occurs, the amount of data specified by the *segSize* parameter will be acquired. Acquisition will then stop until another trigger is received, at which point another segment of *segSize* samples will be acquired. This series of events will continue until the PDA16 reaches the specified total samples count (via SetSampleCountD16). To be meaningful, the segment size should always be set smaller than the active memory size.

Related Functions

SetTriggerModeD16

3.5.23 SetSampleCountD16

Form

int SetSampleCountD16 (HPDA16 hBrd, unsigned int val);
int GetSampleCountD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the PDA16's total sample count; defines length of subsequent acquisitions/transfers

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The new total sample count. This value can be any integer multiple of 2048 not greater than 2147483648. The function will clip this value to the maximum or align down if necessary. For dual channel data acquisitions, this value represents the combined total of samples to acquire for both channels. This value can be D16_FREE_RUN (0) to specify a free-run (infinite) 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.

On success, GetSampleCountD16 returns the current sample count setting.

Remarks

This function sets the total number of samples to be acquired or transferred. When this function is called, all subsequent acquisitions or data transfers will end when the specified number of samples is reached.

In the SAB or PCI acquisition modes it is possible to acquire more than D16_MAX_SAMPLE_COUNT samples. This is done by specifying the special value D16_FREE_RUN (0). Free-run acquisitions acquire data indefinitely. To terminate a free-run acquisition, return the board to Standby mode.

Related Functions

SetStartSampleD16

3.5.24 SetStartSampleD16

Form

int SetStartSampleD16 (HPDA16 hBrd, unsigned int val);

int GetStartSampleD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the PDA16's start sample; defines starting PDA16 RAM address of subsequent acquisitions/transfers

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The new starting sample. This value can be any integer multiple of 2048 not greater than 2147481600. The function will clip this value to the maximum or align down if necessary.

Return Value

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

On success, GetStartSampleD16 returns the current starting sample setting.

Remarks

This function sets the PDA16 RAM address counter to the given sample number. This value determines the starting sample at which data acquisitions or data transfers begin when put into an active operating mode.

The starting sample used by most applications is zero. This means that data acquisitions and transfers will start at the beginning of the PDA16 RAM.

Related Functions

SetAddressCounterAutoResetOverrideD16, SetSampleCountD16

3.5.25 SetTimestampCounterModeD16

Form

int SetTimestampCounterModeD16 (HPDA16 hBrd, unsigned int val);
int GetTimestampCounterModeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set timestamp counter mode; determines how timestamp counter increments

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

The timestamp mode to set. Currently defined timestamp modes are listed in the following table.

PDA16 Library Constant	Value	Interpretation
D16TSCNTMODE_DEFAULT	0	Counter unconditionally increments during acquisition mode.
D16TSCNTMODE_PAUSE_WHEN_ARMED	1	Counter only increments when digitizing; pauses when waiting for trigger.

Return Value

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

On success, GetTimestampCounterModeD16 returns the current timestamp counter mode.

Remarks

The PDA16 firmware automatically resets the timestamp counter to zero when entering acquisition mode. The timestamp counter mode affects how the PDA16 firmware manages the timestamp counter.

In the default mode, D16TSCNTMODE_DEFAULT, the timestamp counter is reset to zero when entering acquisition mode. The counter is then unconditionally incremented once per acquisition clock cycle, regardless of whether the card is digitizing or waiting for a trigger.

In the D16TSCNTMODE_PAUSE_WHEN_ARMED timestamp counter mode, the timestamp counter is reset to zero when entering acquisition mode. The counter is then incremented once per acquisition clock cycle, but only while the card is digitizing. While the card is armed and waiting for a trigger, the timestamp counter is not incremented.

Related Functions

SetTimestampModeD16

3.5.26 SetTimestampModeD16

Form

int SetTimestampModeD16 (HPDA16 hBrd, unsigned int val);
int GetTimestampModeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set timestamp mode; determines how timestamps are generated

Parameters

[in] *hBrd*

A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*

If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

The timestamp mode to set. Currently defined timestamp modes are listed in the following table.

PDA16 Library Constant	Value	Interpretation
D16TSMODE_NO_TIMESTAMPS	0	No timestamps are generated. This is the default setting.
D16TSMODE_SEGMENTS	1	A timestamp is generated for each segment in a segmented acquisition. Intended to work with Segmented trigger mode.
D16TSMODE_TS_ON_EXT_TRIGGER	2	A timestamp is generated for each trigger received on external trigger, regardless if that trigger started an acquisition or segment. This timestamp mode works with any trigger mode.
D16TSMODE_TS_ON_DIGITAL_INPUT	3	(This feature requires a minimum firmware version of 2.1) A timestamp is generated for each rising edge of digital input. In order to use this mode, the digital IO mode must be D16DIGOUT_INPUT_TS_GEN and the digital IO port must be enabled. See SetDigitalOutputModeD16 and SetDigitalOutputEnableD16.

Return Value

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

On success, GetTimestampModeD16 returns the current timestamp mode.

Remarks

The PDA16 can be configured to generate timestamps during data acquisitions.

Upon entering an acquisition operating mode, the PDA16 firmware will automatically reset the timestamp counter to zero. This counter will then be incremented by 1 with each tick of the PDA16 acquisition clock.

When the PDA16 determines that a timestamp should be generated (by the timestamp mode), the current timestamp counter value will be inserted into the PDA16 timestamp FIFO. The PDA16 timestamp FIFO can hold 2048 64-bit timestamps. (The GetTimestampFifoDepthD16 function can be used to programmatically determine the timestamp FIFO depth in the event that the FIFO size is changed in a future firmware update.)

The ReadTimestampDataD16 library function is used to read timestamp values from the timestamp FIFO. The library can also be configured to automatically read and save timestamp data in an external file during a recording operation or a ReadSampleRamFileFastD16/ReadSampleRamFileD16 call. See D16S_FILE_WRITE_PARAMS structure documentation for details.

Related Functions

ReadTimestampDataD16, SetTimestampCounterModeD16

3.5.27 SetTriggerDelaySamplesD16

Form

int SetTriggerDelaySamplesD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerDelaySamplesD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger delay samples; count of samples to skip after trigger

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The number of samples to ignore after a trigger. This can be any even number not greater than D16_MAX_PRE_TRIGGER_SAMPLES (131070). The function will clip this value at the maximum allowed value if necessary. If this value is 0 then no samples will be dropped.

Return Value

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

On success, GetTriggerDelaySamplesD16 returns the current trigger delay samples setting.

Remarks

When using trigger delay samples, the acquisition of data will be delayed from the trigger event. The length of the delay is the number of specified samples.

Trigger delay samples can be used in both Post Trigger and Segmented triggering modes.

The pre-trigger samples setting and delay trigger samples setting are mutually exclusive. Using both at the same time will result in undefined behavior.

Related Functions

SetPreTriggerSamplesD16

3.5.28 SetTriggerDirectionAD16

Form

int SetTriggerDirectionAD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerDirectionAD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger A direction; defines the direction that defines a trigger

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the trigger slope to use and may be one of the following:

PDA16 Library Constant	Interpretation
D16TRIGDIR_POS (0)	Trigger occurs on positive going signal (Power-up default)
D16TRIGDIR_NEG (1)	Trigger occurs on negative going signal

Return Value

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

On success, GetTriggerDirectionAD16 returns the current trigger A direction setting.

Remarks

The PDA16 implements two independent triggers, A and B. Each trigger has its own level and direction. A trigger can be disabled by setting its trigger level to 0 and direction to negative.

Related Functions

SetTriggerLevelAD16, SetTriggerSourceD16

3.5.29 SetTriggerDirectionBD16

Form

int SetTriggerDirectionBD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerDirectionBD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger B direction; defines the direction that defines a trigger

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the trigger slope to use and may be one of the following:

PDA16 Library Constant	Interpretation
D16TRIGDIR_POS (0)	Trigger occurs on positive going signal (Power-up default)
D16TRIGDIR_NEG (1)	Trigger occurs on negative going signal

Return Value

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

On success, GetTriggerDirectionBD16 returns the current trigger B direction setting.

Remarks

The PDA16 implements two independent triggers, A and B. Each trigger has its own level and direction. A trigger can be disabled by setting its trigger level to 0 and direction to negative.

Related Functions

SetTriggerLevelBD16, SetTriggerSourceD16

3.5.30 SetTriggerDirectionExtD16

Form

int SetTriggerDirectionExtD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerDirectionExtD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set external trigger direction

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*

Specifies the trigger slope to use for the external trigger and may be one of the following:

PDA16 Library Constant	Interpretation
D16TRIGDIR_POS (0)	Trigger occurs on positive-going edge of TTL signal (Power-up default)
D16TRIGDIR_NEG (1)	Trigger occurs on negative-going edge of TTL signal

Return Value

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

On success, GetTriggerDirectionExtD16 returns the current trigger B direction setting.

Remarks

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

Related Functions

SetTriggerSourceD16

3.5.31 SetTriggerLevelAD16

Form

int SetTriggerLevelAD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerLevelAD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger A level; defines threshold for an internal trigger event

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The digital ADC value that defines the trigger level threshold for trigger A. 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, GetTriggerLevelAD16 returns the current trigger A level setting.

Remarks

The PDA16 implements two independent triggers, A and B. Each trigger has its own level and direction. A trigger can be disabled by setting its trigger level to 0 and direction to negative.

The trigger level defines the threshold at which a trigger event is detected. A trigger event is defined by the current ADC value crossing the value specified by this function in the direction specified by the Trigger A Direction (SetTriggerDirectionAD16).

Previous generation data acquisition cards (like the PDA14, PDA1000, or PDA12) used analog triggering. The PDA16 uses digital triggering so the trigger level unit is in ADC counts, versus relative voltage.

Related Functions

SetTriggerDirectionAD16, SetTriggerSourceD16

3.5.32 SetTriggerLevelBD16

Form

int SetTriggerLevelBD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerLevelBD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger B level; defines threshold for an internal trigger event

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
The digital ADC value that defines the trigger level threshold for trigger B. 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, GetTriggerLevelBD16 returns the current trigger B level setting.

Remarks

The PDA16 implements two independent triggers, A and B. Each trigger has its own level and direction. A trigger can be disabled by setting its trigger level to 0 and direction to negative.

The trigger level defines the threshold at which a trigger event is detected. A trigger event is defined by the current ADC value crossing the value specified by this function in the direction specified by the Trigger B Direction (SetTriggerDirectionBD16).

Previous generation data acquisition cards (like the PDA14, PDA1000, or PDA12) used analog triggering. The PDA16 uses digital triggering so the trigger level unit is in ADC counts, versus relative voltage.

Related Functions

SetTriggerDirectionBD16, SetTriggerSourceD16

3.5.33 SetTriggerModeD16

Form

int SetTriggerModeD16 (HPDA16 hBrd, unsigned int val);
int GetTriggerModeD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set triggering mode; relates trigger events to how digitized data is saved

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the trigger mode to set and may be one of the following:

PDA16 Library Constant	Interpretation
D16TRIGMODE_POST_TRIGGER (0)	Trigger event starts a single data acquisition (Power-up default)
D16TRIGMODE_SEGMENTED (1)	Each trigger event begins a new statically sized data acquisition

Return Value

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

On success, GetTriggerModeD16 returns the current trigger mode setting.

Related Functions

SetTriggerSourceD16

3.5.34 SetTriggerSourceD16

Form

int SetTriggerSourceD16 (HPDA16 hBrd, unsigned int val);

int GetTriggerSourceD16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set trigger source; defines where trigger events originate

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the trigger source to use and may be one of the following:

PDA16 Library Constant	Interpretation
D16TRIGSRC_INT_CH1 (0)	Internal trigger, channel 1 (Power-up default)
D16TRIGSRC_INT_CH2 (1)	Internal trigger, channel 2
D16TRIGSRC_EXT (2)	External trigger

Return Value

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

On success, GetTriggerSourceD16 returns the current trigger source setting.

Remarks

This function allows the user to select the PDA16 trigger source. If the trigger source is set to internal, the ADC input signal from either channel 1 or channel 2 becomes the trigger signal. If the trigger source is set to external, the signal from the external trigger input connector is used for the trigger source.

Related Functions

SetTriggerLevelAD16, SetTriggerDirectionAD16, SetTriggerModeD16

3.5.35 SetVoltRangeCh1D16

Form

int SetVoltRangeCh1D16 (HPDA16 hBrd, unsigned int val);
int GetVoltRangeCh1D16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the channel 1 input voltage range

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the channel 1 volt range value to use and may be one of the following:

PDA16 Library Constant	Value	Interpretation
D16VOLTRNG_2_50V	0	2.50 volts peak-to-peak
D16VOLTRNG_1_67V	1	1.67 volts peak-to-peak
D16VOLTRNG_1_00V	2	1.00 volts peak-to-peak
D16VOLTRNG_0_667V	3	667 millivolts peak_to_peak
D16VOLTRNG_0_400V	4	400 millivolts peak_to_peak
D16VOLTRNG_0_267V	5	267 millivolts peak_to_peak

Return Value

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

On success, GetVoltRangeCh1D16 returns the current channel 1 input voltage setting.

Related Functions

SetVoltRangeCh2D16

3.5.36 SetVoltRangeCh2D16

Form

int SetVoltRangeCh2D16 (HPDA16 hBrd, unsigned int val);
int GetVoltRangeCh2D16 (HPDA16 hBrd, int bFromCache = 1);

Description

Get or set the channel 1 input voltage range

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromCache*
If non-zero, the setting will be read from the local device register cache associated with the given PDA16 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 PDA16 device register read.

[in] *val*
Specifies the channel 1 volt range value to use and may be one of the following:

PDA16 Library Constant	Value	Interpretation
D16VOLTRNG_2_50V	0	2.50 volts peak-to-peak
D16VOLTRNG_1_67V	1	1.67 volts peak-to-peak
D16VOLTRNG_1_00V	2	1.00 volts peak-to-peak
D16VOLTRNG_0_667V	3	667 millivolts peak_to_peak
D16VOLTRNG_0_400V	4	400 millivolts peak_to_peak
D16VOLTRNG_0_267V	5	267 millivolts peak_to_peak

Return Value

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

On success, GetVoltRangeCh1D16 returns the current channel 1 input voltage setting.

Related Functions

SetVoltRangeCh1D16

3.6 Device Register State Functions

The functions in this section involve manipulation of PDA16 device registers

3.6.1 CopyHardwareSettingsD16

Form

```
int CopyHardwareSettingsD16 (HPDA16 hBrdDst, HPDA16 hBrdSrc);
```

Description

Copy hardware settings from another PDA16 device

Parameters

[in] *hBrdDst*
A handle to the PDA16 device to which the copied hardware settings will be applied.

[in] *hBrdSrc*
A handle to the PDA16 device from which the hardware settings are copied from.

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 PDA16 associated with the source device to the PDA16 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.

Related Functions

RewriteHardwareSettingsD16, RefreshLocalRegisterCacheD16

3.6.2 LoadSettingsFromBufferXmlD16

Form

```
int LoadSettingsFromBufferXmlD16 (HPDA16 hBrd, unsigned int flags, const TCHAR* pBuffer);
```

Description

Load hardware settings from an XML string

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *flags*
Flags that affect how settings are loaded:

Library Constant	Value	Interpretation
D16XMLSET_NO_PRELOAD_DEFAULTS	0x00000004	Do not set default hardware settings prior to loading settings

[in] *bufp*
A pointer to a NULL-terminated string containing the XML data containing the settings data. The SaveSettingsToBufferXmlD16 function is used to generate this data.

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 refresh all internal device register caches with current hardware register values.

Related Functions

SaveSettingsToBufferXmlD16, LoadSettingsFromFileXmlD16

3.6.3 LoadSettingsFromFileXmlD16

Form

```
int LoadSettingsFromFileXmlD16 (HPDA16 hBrd, unsigned int flags, const TCHAR* bufp);
```

Description

Load hardware settings from an XML file

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *flags*
Flags that affect how settings are loaded:

Library Constant	Value	Interpretation
D16XMLSET_NO_PRELOAD_DEFAULTS	0x00000004	Do not set default hardware settings prior to loading settings

[in] *bufp*
A pointer to a NULL-terminated string containing the pathname of the XML file containing the settings data. The SaveSettingsToFileXmlD16 function is used to generate this data.

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 apply hardware settings saved by a previous call to SaveSettingsToFileXmlD16.

Related Functions

SaveSettingsToFileXmlD16

3.6.4 ReadAllDeviceRegistersD16

Form

```
int ReadAllDeviceRegistersD16 (HPDA16 hBrd);
```

Description

Read all device registers from hardware; updates all register caches

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 refresh all internal device register caches with current hardware register values.

Related Functions

RewriteHardwareSettingsD16, RefreshLocalRegisterCacheD16

3.6.5 RefreshLocalRegisterCacheD16

Form

```
int RefreshLocalRegisterCacheD16 (HPDA16 hBrd, int bFromHardware = 0);
```

Description

Refresh local device register cache from driver's cache; no hardware read

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *bFromHardware*
If this parameter is non-zero then register content will be updated from the PDA16 hardware. If zero, the driver's local register cache will be consulted. Since all device writes go through the driver, the driver's cache will contain an updated cache of hardware register content. (Status registers are the exception to this.)

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 update the local PDA16 device register cache associated with the given PDA16 device handle. The register values are obtained from the kernel-level register cache maintained by the PDA16 device driver. Calling this function has no effect on any underlying PDA16 hardware.

This function is automatically invoked by the ConnectToDeviceD16 function as part of the device connection procedure.

Related Functions

RewriteHardwareSettingsD16

3.6.6 RewriteHardwareSettingsD16

Form

```
int RewriteHardwareSettingsD16 (HPDA16 hBrd);
```

Description

Bring hardware settings up to date with current cache settings

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 rewrite all hardware settings with the values contained in the local register cache that is associated with the given PDA16 handle.

Calling this function will place the board in the Standby operating mode regardless of the operating mode in the register cache.

Related Functions

RefreshLocalRegisterCacheD16

3.6.7 SaveSettingsToBufferXmlD16

Form

```
int SaveSettingsToBufferXmlD16 (HPDA16 hBrd, unsigned int flags, TCHAR** bufpp, int* buflenp);
```

Description

Save board settings to XML format in a library allocated-buffer

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *flags*
Flags that affect how settings are saved:

Library Constant	Value	Interpretation
D16XMLSET_NODE_ONLY	0x00000001	Serialize to a node only; do not include XML header info
D16XMLSET_FORMAT_OUTPUT	0x00000002	Pretty-print XML output; add newlines and indentation. This will generate nicer, human-readable output.

[out] *bufp*
A pointer to a TCHAR* variable that will receive the address of a library-allocated buffer containing the XML data. It is the caller's responsibility to free this memory with FreeMemoryD16 function.

[out] *buflenp*

A pointer to an integer variable that will receive the length of the library-allocated buffer, in characters.

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 save hardware settings to an XML buffer. These settings can later be reloaded by calling the LoadSettingsFromBufferXmlD16 function.

Related Functions

LoadSettingsFromBufferXmlD16, SaveSettingsToFileXmlD16

3.6.8 SaveSettingsToFileXmlD16

Form

```
int SaveSettingsToFileXmlD16 (HPDA16 hBrd, unsigned int flags, TCHAR* pathnamep, TCHAR* encodingp = "UTF-8");
```

Description

Save board settings to an XML file

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *flags*
Flags that affect how settings are saved:

Library Constant	Value	Interpretation
D16XMLSET_NODE_ONLY	0x00000001	Serialize to a node only; do not include XML header info
D16XMLSET_FORMAT_OUTPUT	0x00000002	Pretty-print XML output; add newlines and indentation. This will generate nicer, human-readable output.

[out] *pathnamep*
A pointer to a NULL-terminated string containing the pathname of the XML file.

[out] *encodingp*
A pointer to a NULL-terminated string containing the encoding to use for the generated XML file. This can be any encoding supported by the iconv library: ASCII, UTF-8, UTF16, char, wchar_t, etc.

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 save hardware settings to an XML file. These settings can later be reloaded by calling the LoadSettingsFromFileXmlD16 function.

Related Functions

LoadSettingsFromFileXmlD16

3.6.9 SetPowerupDefaultsD16

Form

```
int SetPowerupDefaultsD16 (HPDA16 hBrd);
```

Description

Restores all PDA16 settings to power-up default values

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

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

Remarks

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. For most all PDA16 settings, these equate to the '0' values. A few exceptions to this:

- Channel 1 and 2 DC offsets are set to midscale, ~0V (D16_DC_OFFSET_NULL)
- Trigger levels are set to midscale (D16_TRIGGER_LEVEL_MIDSCALE)
- Internal acquisition rate is set to 160MHz
- The board is kept in Standby mode

3.7 Memory/DMA Buffer Allocation Routines

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

3.7.1 AllocateDmaBufferD16

Form

```
int AllocateDmaBufferD16 (HPDA16 hBrd, unsigned int samples, pda16_sample_t** bufpp);
```

Description

Allocate a DMA buffer for use with DMA transfers

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *samples*
The number of samples 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.

[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 calling process' address space. 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 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 FreeDmaBufferD16 function when it is no longer needed.

The DMA buffer may only be used with the PDA16 device that it was allocated for. A DMA buffer can be used with another PDA16 device handle (HPDA16) of the same process as long as both device handles refer to the same underlying PDA16 device.

All 'Fast' PDA16 library functions (e.g. ReadSampleRamFastD16, WriteSampleRamFastD16) require a DMA buffer as a parameter. These functions are faster than their counterparts because the driver can perform the DMA transfer directly to the DMA buffer which is mapped in the user process. The "slower" functions work by using a smaller driver-allocated DMA buffer to perform the data transfers which is less efficient.

The maximum size of a DMA buffer is system-dependent.

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

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 PDA16 driver will ensure that all un-freed DMA buffers for a given process will be freed when that process' last handle is closed. That is to say, like normal, heap-allocated memory, the underlying PDA16 software will ensure that all DMA buffers are properly cleaned up when a process exits, gracefully or not.

Related Functions

FreeDmaBufferD16, ReadSampleRamFastD16, WriteSampleRamFastD16

3.7.2 FreeDmaBufferD16

Form

```
int FreeDmaBufferD16 (HPDA16 hBrd, pda16_sample_t* bufp);
```

Description

Free a DMA buffer previously allocated by the AllocateDmaBufferD16 function

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 PDA16 driver will automatically free any DMA buffers that have not been freed by a process when it exits.

Related Functions

AllocateDmaBufferD16

3.7.3 FreeMemoryD16

Form

```
int FreeMemoryD16 (void* p);
```

Description

Free memory allocated by the PDA16 library

Parameters

[in] *p*
The address of the PDA16 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 PDA16 library functions allocate memory on behalf of the caller and it is the caller's responsibility to free this memory when it is no longer needed. Use this function to free the memory. Do not use *free* or *delete* because

the PDA16 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 FreeDmaBufferD16

3.8 Data Acquisition Routines

The functions in this section are used to perform data acquisitions

3.8.1 AcquireToBoardRamD16

Form

```
int AcquireToBoardRamD16 (HPDA16 hBrd, unsigned int samp_start, unsigned int samp_count, unsigned int timeout_ms = 0, int bAsynchronous = 0);
```

Description

Acquire data to PDA16 RAM

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *samp_start*
The PDA16 RAM sample at which to put the first acquired sample. This parameter has the same restrictions defined in the SetStartSampleD16 function.

[in] *samp_count*
The total number of samples to acquire. This is independent of active channel count and segment size. This parameter has the same restrictions defined in the SetSampleCountD16 function.

[in] *timeout_ms*
An optional timeout value in milliseconds. If this value is 0 then the function will not timeout. This parameter is ignored if parameter *bAsynchronous* is nonzero.

[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.

This function will return SIG_CANCELLED (-10) if the RAM acquisition is cancelled before it finishes.

Remarks

This function will perform a synchronous RAM acquisition using current data acquisition settings to the specified region of PDA16 onboard RAM. This function will not return until the acquisition has completed or the optional timeout has elapsed. The invoking thread will sleep while the acquisition takes place. The Samples Complete interrupt will be used for end of acquisition notification.

Once the data acquisition has completed, data may be transferred from the PDA16 sample RAM to the host PC by calling ReadSampleRamFastD16 or ReadSampleRamD16.

An acquisition may also be cancelled by putting the board into Standby operating mode from a secondary thread or process. A separate thread is required because the thread that invoked this function will be suspended waiting for the acquisition to finish (or timeout).

This function handles all necessary operating mode changes as well as setting up the active memory region.

This function can also start an asynchronous acquisition. By default, this function will not return until the data acquisition completes. If the *bAsynchronous* parameter is nonzero, then the function will start the data acquisition and return immediately without waiting for it to finish allowing the calling thread to continue working. The caller can use the IsAcquisitionInProgressD16 function to determine if the acquisition is still in progress. To wait (sleep) for the acquisition to complete, call the WaitForAcquisitionCompleteD16 function.

Related Functions

AcquireToSabD16, IsAcquisitionInProgressD16, WaitForAcquisitionCompleteD16

3.8.2 AcquireToSabD16

Form

```
int AcquireToSabD16 (HPDA16 hBrd , unsigned int samp_count, int bRamBuffered = 0, unsigned int timeout_ms = 0, int bAsynchronous = 0);
```

Description

Synchronously acquire data to Signatec Auxiliary Bus (SAB)

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *samp_count*
The total number of samples to acquire. This is independent of active channel count and segment size. If this parameter is D16_FREE_RUN, then a free-run (infinite) SAB acquisition will take place. In this case, the timeout is ignored and the function will return immediately after placing the board into acquisition mode. To stop a free-run acquisition place the board in Standby mode.

[in] *bRamBuffered*
If this parameter is nonzero then the RAM-buffered SAB acquisition mode (D16MODE_ACQ_SAB_BUF) will be used. If this parameter is zero then the normal SAB acquisition mode (D16MODE_ACQ_SAB) will be used.

[in] *timeout_ms*
An optional timeout value in milliseconds. If this value is 0 then the function will not timeout.

[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.

This function will return SIG_CANCELLED (-10) if the RAM acquisition is cancelled before it finishes.

Remarks

This function will perform a synchronous SAB acquisition using current data acquisition settings to the specified region of PDA16 onboard RAM. This function will not return until the acquisition has completed or the optional timeout has elapsed. The invoking thread will sleep while the acquisition takes place. The Samples Complete interrupt will be used for end of acquisition notification.

An acquisition may also be cancelled by putting the board into Standby operating mode from a secondary thread or process. A separate thread is required because the thread that invoked this function will be suspended waiting for the acquisition to finish (or timeout).

This function handles all necessary operating mode changes as well as setting up the active memory region.

This function can also start an asynchronous acquisition. By default, this function will not return until the data acquisition completes. If the *bAsynchronous* parameter is nonzero, then the function will start the data acquisition and return immediately without waiting for it to finish allowing the calling thread to continue working. The caller can use the IsAcquisitionInProgressD16 function to determine if the acquisition is still in progress. To wait (sleep) for the acquisition to complete, call the WaitForAcquisitionCompleteD16 function.

Related Functions

AcquireToBoardRamD16

3.8.3 BeginBufferedPciAcquisitionD16

Form

```
int BeginBufferedPciAcquisitionD16 (HPDA16 hBrd , unsigned int samp_count = D16_FREE_RUN);
```

Description

Begin a PDA16 RAM buffered PCI acquisition

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *samp_count*
The total number of samples to acquire. For most applications the default value will be used to indicate an infinite recording length. When the desired amount of data has been obtained the recording is then stopped.

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 begin a PDA16 RAM buffered PCI data acquisition. In this type of data acquisition, the PDA16 sample RAM (256 mebi-samples) is used as a large FIFO to buffer the data as it is consumed via DMA transfer over the PCI/PCI-X bus.

Once the acquisition has been started, the GetPciAcquisitionDataD16 function is repeatedly called to transfer acquisition data to the host PC. When the desired amount of data has been obtained, the acquisition is ended by calling the EndBufferedPciAcquisitionD16 function.

This function takes care of setting up the active memory region and all mode changes.

Related Functions

GetPciAcquisitionDataD16, EndBufferedPciAcquisitionD16

3.8.4 EndBufferedPciAcquisitionD16

Form

```
int EndBufferedPciAcquisitionD16 (HPDA16 hBrd);
```

Description

End a buffered PCI acquisition

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 PDA16 RAM buffered PCI acquisition started by a previous call to BeginBufferedPciAcquisitionD16.

Related Functions

BeginBufferedPciAcquisitionD16, GetPciAcquisitionDataD16

3.8.5 IsAcquisitionInProgressD16

Form

```
int IsAcquisitionInProgressD16 (HPDA16 hBrd);
```

Description

Determine if an asynchronous RAM acquisition, SAB acquisition, or SAB transfer is currently in progress

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

Returns 1 if an acquisition is currently in progress, 0 if an acquisition is not in progress, or one of the library error codes (which are all negative) on error. See remarks.

Remarks

This function is not used for PCI-based acquisitions. Instead, the IsTransferInProgressD16 is used to determine if a DMA transfer is currently in progress.

This function is used to determine if any of the following operations are currently in progress:

- RAM acquisition (via AcquireToBoardRamD16)
- SAB acquisition (via AcquireToSabD16)
- SAB transfer (via TransferSampleRamToSabD16)

Data acquisition operations are considered in progress even if the board has not yet received a trigger event.

A return value 1 should be interpreted as: A RAM acquisition, SAB acquisition, or SAB transfer operation has been started, but we haven't been notified by the hardware that the operation has completed.

A return value 0 should be interpreted as: The PDA16 driver is not currently expecting a notification for any of the above operations. This may be because the operation has already finished or because it was never attempted.

This function will always return 0 for virtual devices.

This function is useful when doing asynchronous data acquisition operations.

Related Functions

WaitForAcquisitionCompleteD16

3.8.6 WaitForAcquisitionCompleteD16

Form

```
int WaitForAcquisitionCompleteD16 (HPDA16 hBrd, unsigned int timeout_ms = 0);
```

Description

Wait for a RAM acquisition, SAB acquisition, or SAB transfer operation to complete with optional timeout

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *timeout_ms*
An optional timeout value in milliseconds. If this value is 0 then the function will not timeout

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_D16_TIMED_OUT (-541) if the timeout elapses before the acquisition operation completes; see remarks.

This function will return SIG_CANCELLED (-10) if the operation is cancelled before it finishes; see remarks.

Remarks

This function is used when doing asynchronous RAM acquisition operations. This function is not used for PCI-based acquisitions. Instead, the WaitForTransferCompleteD16 is used to wait for a DMA transfer to finish.

This function can be used to wait for any of the following asynchronous operations to complete:

- RAM acquisition (via AcquireToBoardRamD16)
- SAB acquisition (via AcquireToSabD16)
- SAB transfer (via TransferSampleRamToSabD16)

Calling WaitForAcquisitionCompleteD16 while any of the above operations are in progress will result in the calling thread being blocked until one of the following events occur:

- The acquisition completes; function returns SIG_SUCCESS
- The acquisition is cancelled; function returns SIG_CANCELLED
- The wait times out; function returns SIG_D16_TIMED_OUT
- An error occurs; function returns error code

In the first two cases, the board will automatically be put into Standby mode. In all other cases the operating mode is not changed.

An acquisition is cancelled by putting the board into Standby mode, which can be done from any other thread or process.

Related Functions

IsAcquisitionInProgressD16

3.9 Data Transfer Routines

The functions in this section are used to perform data transfers

3.9.1 GetPciAcquisitionDataD16

Form

```
int GetPciAcquisitionDataD16 (HPDA16 hBrd, unsigned int samples, pda16_sample_t* dma_bufp, int bAsynchronous = 0);
```

Description

Obtain fresh acquisition data during a PCI data acquisition

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *samp_count*
The number of data samples to transfer. This value must be a multiple of 1024 samples.

[out] *dma_bufp*
A pointer to a DMA buffer that will be used for the DMA transfer performed to obtain the acquisition data. This buffer must be at least *samp_count* samples. If a non-DMA buffer is used the function will return an error.

[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.

This function will return SIG_CANCELLED (-10) if the data transfer is cancelled before it finishes.

Remarks

This function is used to obtain data from the acquisition started by a previous call to BeginBufferedPciAcquisitionD16.

This function will not return until all of the requested samples have been transferred. It is safe to call this function before the desired number of samples has been acquired.

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 IsTransferInProgressD16 function to determine if the transfer is still in progress. To wait (sleep) for the transfer to complete, call the WaitForTransferCompleteD16 function.

Synchronous transfers: When the underlying transfer completes the software will check the status of the PCI FIFO Full flag. If this flag is set then the PDA16 RAM-FIFO went full at some point during the transfer. This means that an indeterminate amount of data may have been lost during the transfer. This will happen if data cannot be pulled from the card faster than it is acquiring data. If the flag is set, the function will return SIG_D16_FIFO_OVERFLOW.

Related Functions

BeginBufferedPciAcquisitionD16, EndBufferedPciAcquisitionD16, IsTransferInProgressD16, WaitForTransferCompleteD16

3.9.2 IsTransferInProgressD16

Form

```
int IsTransferInProgressD16 (HPDA16 hBrd);
```

Description

Determine if an asynchronous data transfer is currently in progress

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 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 PDA16 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 we haven't been notified by the hardware that the operation has completed.

A return value 0 should be interpreted as: The PDA16 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

WaitForTransferCompleteD16

3.9.3 ReadSampleRamFastD16

Form

```
int ReadSampleRamFastD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
pda16_sample_t* dma_bufp, int bAsynchronous);
```

Description

Transfer data in PDA16 RAM to a DMA buffer on the host PC

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *sample_start*
The index of the first sample to copy. This parameter has the same restrictions defined in the SetStartSampleD16 function.

[in] *sample_count*
The total number of samples to copy. This parameter has the same restrictions defined in the SetSampleCountD16 function and must be an integer multiple of 1024 samples.

[out] *dma_bufp*
The address of the host PC buffer at which the sample data will be copied to. This buffer must be large enough to hold *sample_count* samples and must have been allocated for this device by the AllocateDmaBufferD16 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.

Remarks

This function copies a section of PDA16 signal RAM to a buffer in the host PC memory using Direct Memory Access (DMA) transfers. Before this function may be used, a DMA buffer must be allocated for the target device using the AllocateDmaBufferD16 function.

This function handles all necessary operating mode changes and active memory settings.

The difference between this function and the ReadSampleRamD16 function is that ReadSampleRamD16 does not require a DMA buffer for input. Rather, an internal driver-managed DMA buffer is used for transfer. This requires an extra buffering of the data that can hinder high-performance applications.

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 IsTransferInProgressD16 function to determine if the transfer has completed. To wait (sleep) for the transfer to complete, call the WaitForTransferCompleteD16 function.

Related Functions

AllocateDmaBufferD16, ReadSampleRamD16, IsTransferInProgressD16, WaitForTransferCompleteD16

3.9.4 ReadSampleRamD16

Form

```
int ReadSampleRamD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
pda16_sample_t* bufp);
```

Description

Transfer data in PDA16 RAM to a buffer on the PC; any boundary or alignment at the expense of speed.

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *start_sample*

The first sample in the PDA16 RAM in which you would like to copy. This can be any valid sample number and need not be on any particular boundary.

[in] *sample_count*

The number of samples to copy from the PDA16 RAM. This value need not be on any particular boundary.

[out] *bufp*

A pointer to the buffer that will receive the PDA16 data. This buffer does not have to be a DMA buffer. (If you do have a DMA buffer you should use ReadSampleRamFastD16 since it's much faster due to not having to use intermediate buffering.)

Return Value

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

Remarks

This function transfers the data from the given region of the given board to the given buffer. Note that this buffer need not be a DMA buffer. This function exists as a convenience in situations where getting data off of the board is not speed-critical. This function does not run as fast as the ReadSampleRamFastD16 function but has a few advantages:

- No DMA buffer needs to be allocated by the caller
- Starting sample and transfer length needn't be on any particular boundary
- The transfer size is not bound by the minimum or maximum DMA transfer size

The difference between this function and the ReadSampleRamFastD16 is that the ReadSampleRamFastD16 function requires a DMA buffer in order to do the data transfer. Also, start and length of transfer are also restricted to aligned values when using ReadSampleRamFastD16. The advantage of this is that ReadSampleRamFastD16 can run faster since no intermediate buffering is required.

To transfer and automatically de-interleave dual-channel data use the ReadSampleRamDualChannelD16 function.

Related Functions

ReadSampleRamFastD16, ReadSampleRamDualChannelD16

3.9.5 ReadSampleRamDualChannelD16

Form

```
int ReadSampleRamDualChannelD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
pda16_sample_t* buf_ch1p, pda16_sample_t* buf_ch2p);
```

Description

Transfer and de-interleave dual-channel data in board RAM to host PC; any boundary or alignment at the expense of speed

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *start_sample*
The first sample in the PDA16 RAM in which you would like to copy. This can be any valid sample number and need not be on any particular boundary.

[in] *sample_count*
The number of samples to copy from the PDA16 RAM. This value need not be on any particular boundary.

[out] *buf_ch1p*
A pointer to the buffer that will receive the channel 1 data. This buffer must be large enough to hold *sample_count* / 2 samples. This parameter may be NULL if channel 1 data is not needed.

[out] *buf_ch2p*
A pointer to the buffer that will receive the channel 2 data. This buffer must be large enough to hold *sample_count* / 2 samples. This parameter may be NULL if channel 2 data 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

This function transfers the data from the given region of the given board to the given buffer. Note that this buffer need not be a DMA buffer. This function exists as a convenience in situations where getting data off of the board is not speed-critical. Speed critical applications should use the ReadSampleRamFastD16 function to obtain single- or dual-channel data.

This function will also automatically de-interleave the data into separate buffers. If you do not want to de-interleave data you should use the ReadSampleRamD16 function.

Related Functions

ReadSampleRamD16

3.9.6 ReadSampleRamFileFastD16

Form

```
int ReadSampleRamFileFastD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
pda16_sample_t* dma_bufp, unsigned int dma_buf_samples, D16S_FILE_WRITE_PARAMS* paramsp);
```

Description

Transfer data in PDA16 RAM to a file on the host PC

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *sample_start*
The index of the first sample to copy. This parameter has the same restrictions defined in the SetStartSampleD16 function.

[in] *sample_count*
The total number of samples to copy. This parameter has the same restrictions defined in the SetSampleCountD16 function and must be an integer multiple of 1024 samples.

[in] *dma_bufp*
The address of a DMA buffer previously allocated by the AllocateDmaBufferD16 function. This DMA buffer is used for the data transfer between the PDA16 and host PC. This buffer need not be as large as the total amount of data to transfer.

[in] *dma_buf_samples*
The size, in samples, of the DMA buffer pointed to by dma_bufp

[in] *paramsp*
A pointer to a D16S_FILE_WRITE_PARAMS structure that defines how and where data will be saved.

Return Value

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

Remarks

This function copies a section of PDA16 signal RAM to a file on the host PC using Direct Memory Access (DMA) transfers. Before this function may be used, a DMA buffer must be allocated for the target device using the AllocateDmaBufferD16 function.

This function handles all necessary operating mode changes and active memory settings.

Related Functions

AllocateDmaBufferD16, ReadSampleRamD16

3.9.7 ReadSampleRamFileD16

Form

```
int ReadSampleRamFileD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
D16S_FILE_WRITE_PARAMS* paramsp);
```

Description

Transfer data in PDA16 RAM to a file on the host PC; any boundary or alignment at the expense of speed.

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *sample_start*

The index of the first sample to copy. This can be any valid sample number and need not be on any particular boundary.

[in] *sample_count*
The total number of samples to copy. This can be any valid sample number and need not be on any particular boundary.

[in] *paramsp*
A pointer to a D16S_FILE_WRITE_PARAMS structure that defines how and where data will be saved.

Return Value

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

Remarks

This function handles all necessary operating mode changes and active memory settings.

Related Functions

ReadSampleRamD16, ReadSampleRamFileFastD16

3.9.8 TransferSampleRamToSabD16

Form

```
int TransferSampleRamToSabD16 (HPDA16 hBrd, unsigned int sample_start, unsigned int sample_count,
unsigned int timeout_ms = 0);
```

Description

Transfer data in PDA16 RAM to the SAB bus

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *start_sample*
The first sample in the PDA16 RAM in which you would like to copy. This can be any valid sample number and need not be on any particular boundary.

[in] *sample_count*
The number of samples to copy from the PDA16 RAM. This parameter has the same restrictions defined in the SetSampleCountD16 function

[in] *timeout_ms*
An optional timeout value in milliseconds. If this value is 0 then the function will not timeout

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 perform a synchronous RAM acquisition using current data acquisition settings to the specified region of PDA16 onboard RAM. This function will not return until the acquisition has completed or the optional timeout has elapsed. The invoking thread will sleep while the acquisition takes place. The Samples Complete interrupt will be used for end of acquisition notification.

An acquisition may also be cancelled by putting the board into Standby operating mode from a secondary thread or process. A separate thread is required because the thread that invoked this function will be suspended waiting for the acquisition to finish (or timeout).

This function handles all necessary operating mode changes as well as setting up the active memory region.

3.9.9 WaitForTransferCompleteD16

Form

```
int WaitForTransferCompleteD16 (HPDA16 hBrd, unsigned int timeout_ms = 0);
```

Description

Wait for an asynchronous DMA transfer operation to complete

Parameters

[in] *pBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[in] *timeout_ms*
An optional timeout value in milliseconds. If this value is 0 then the function will not timeout

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_D16_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 PDA16 into the Standby operating mode.

Remarks

This function is used when doing asynchronous DMA transfer operations while in RAM-Buffered PCI Acquisition mode or one of the PCI transfer operating modes.

When calling this function in the RAM-Buffered PCI Acquisition mode, it is safe to call this function even if the board has not acquired enough data to satisfy the requested transfer length. In this case, the function will wait until the requested amount of data has been acquired and transferred.

The most common cause of this function not returning is that the board is not acquiring any data because it's waiting for a trigger event to start the acquisition (or acquisition segment if using Segmented trigger mode).

Calling this function while a DMA transfer is in progress will result in the current thread being blocked until one of the following events occur:

- The transfer completes; function returns SIG_SUCCESS
- The transfer is cancelled; function returns SIG_CANCELLED
- The specified timeout elapses; function returns SIG_D16_TIMED_OUT
- An error occurs; function returns error code

If the transfer is cancelled the board will be put into Standby mode. In all other cases, the operating mode is not affected.

A transfer is cancelled by putting the board into Standby mode, which can be done from any other thread or process.

Buffered PCI acquisition mode: When the underlying transfer completes the software will check the status of the PCI FIFO Full flag. If this flag is set then the PDA16 RAM-FIFO went full at some point during the transfer. This means that an indeterminate amount of data may have been lost during the transfer. This will happen if data cannot be pulled from the card faster than it is acquiring data. If the flag is set, the function will return SIG_D16_FIFO_OVERFLOW.

Related Functions

IsTransferInProgressD16

3.10 Data Manipulation Routines

3.10.1 DeInterleaveDataD16

Form

```
int DeInterleaveDataD16 (const pda16_sample_t* srcp,
        unsigned int samples_in, pda16_sample_t* dst_ch1p, pda16_sample_t* dst_ch2p)
```

Description

This function is used to de-interleave dual channel 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.

Return Value

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

Remarks

When dual channel data is acquired on the PDA16, sample data is 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 both channels of data into separate buffers.

Related Functions

InterleaveDataD16

3.10.2 InterleaveDataD16

Form

int InterleaveDataD16 (const pda16_sample_t* src_ch1p, const pda16_sample_t* src_ch2p, unsigned int samps_per_chan, pda16_sample_t* dstp)

Description

Interleave dual channel data into a single 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] *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 dual channel data

Related Functions

DeInterleaveDataD16

3.11 PDA16 Recording Session Management Routines

The PDA16 library implements a high-level recording interface that simplifies coding for recording PDA16 acquisition data to permanent storage. Using this interface, all aspects of PDA16 hardware interaction and output file management is managed internally by the library.

The library currently supports two types of recordings: PCI Acquisition recordings and RAM Acquisition/Transfer recordings.

3.11.1 AbortRecordingSessionD16

Form

int AbortRecordingSessionD16 (HD16RECORDING hRec)

Description

Abort the current recording session; stops all recording and closes all files

Parameters

[in] *hRec*
A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 function.

Return Value

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

Related Functions

CreateRecordingSessionD16

3.11.2 ArmRecordingSessionD16

Form

int ArmRecordingSessionD16 (HD16RECORDING hRec)

Description

Arm device for recording

Parameters

[in] *hRec*

A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 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 arm the PDA16 for recording. This function is only needed if the D16RECSESF_DO_NOT_ARM recording flag is specified during creation of the recording session.

Related Functions

CreateRecordingSessionD16

3.11.3 CreateRecordingSessionD16

Form

int CreateRecordingSessionD16 (HPDA16 hBrd, D16S_REC_SESSION_PARAMS* rec_paramsp, HD16RECORDING* handlep)

Description

Create a PDA16 acquisition recording session

Parameters

[in] *hBrd*
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] *rec_paramsp*
A pointer to a D16S_REC_SESSION_PARAMS structure that define the parameters of the data recording session. This structure and its members are detailed in the Structure D16S_REC_SESSION_PARAMS section.

[out] *handlep*
A pointer to a HD16RECORDING variable that will receive a handle that identifies the recording 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 PDA16 recording session. A recording session is a PDA16 library interface used to fully manage the recording of PDA16 acquisition data to permanent storage.

Related Functions

DeleteRecordingSessionD16

3.11.4 DeleteRecordingSessionD16

Form

int DeleteRecordingSessionD16 (HD16RECORDING hRec)

Description

Delete a PDA16 recording session

Parameters

[in] *hRec*
A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 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 recording is currently in progress when this function is called, it will automatically be aborted via a call to AbortRecordingSessionD16.

Related Functions

CreateRecordingSessionD16

3.11.5 GetRecordingSessionOutFlagsD16

Form

int GetRecordingSessionOutFlagsD16 (HD16RECORDING hRec, unsigned int* flagsp)

Description

Obtain recording session output flags; only valid after recording stopped

Parameters

[in] *hRec*
A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 function.

[out] *flagsp*
A pointer to an unsigned integer variable that will receive some additional recording status information that is not available during recording runtime. The currently defined flags are listed in the following table.

Flag	Interpretation
D16FILWOUTF_TIMESTAMP_FIFO_OVERFLOW (0x00000001)	Timestamp FIFO overflowed during write/record process
D16FILWOUTF_NO_TIMESTAMP_DATA (0x00000002)	No timestamp data was available

Return Value

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

Remarks

If this function is to be used, it must be called after AbortRecordingSessionD16 and before DeleteRecordingSessionD16.

3.11.6 GetRecordingSessionProgressD16

Form

int GetRecordingSessionProgressD16 (HD16RECORDING hRec, D16S_REC_SESSION_PROG* progp, unsigned flags)

Description

Obtain progress/status for current recording session

Parameters

[in] *hRec*

A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 function.

[out] *progp*

A pointer to a D16S_REC_SESSION_PROG structure that will receive the current progress/status of the PDA16 recording session. The caller should initialize the struct_ver field of this structure before calling this function.

[in] *flags*

A set of flags (D16RECPROGF_*) that control function behavior. Currently, only a single flag is defined: D16RECPROGF_NO_ERROR_TEXT (1). If this flag is set then the function will not generate an error text string if an error has occurred during recording.

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 current status/progress of the given recording session. This is the primary way of checking on the status of a recording.

Related Functions

CreateRecordingSessionD16, GetRecordingSnapshotD16

3.11.7 GetRecordingSnapshotD16

Form

int GetRecordingSnapshotD16 (HD16RECORDING hRec, pda16_sample_t* bufp, unsigned int samples, unsigned int* samples_gotp, unsigned int* ss_countp)

Description

Obtain data snapshot from current recording

Parameters

[in] *hRec*
A handle to a PDA16 recording session. This handle is obtained by calling the CreateRecordingSessionD16 function.

[out] *bufp*
A pointer to a buffer that will receive the recording snapshot data

[in] *samples*
The size, in samples, of the buffer pointed to by the bufp parameter

[out] *samples_gotp*
The address of an unsigned int variable that will receive the number of samples copied into the snapshot buffer. Pass NULL if this information is not needed

[out] *ss_countp*
The address of an unsigned int variable that will receive the snapshot counter value. Each data snapshot taken by the recording is given a unique counter value. 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

Related Functions

CreateRecordingSessionD16, GetRecordingSessionProgressD16

3.12 Remote PDA16 Operation

The PDA16 library supports remote PDA16 operation. A remote PDA16 is a PDA16 device that is located on a remote host system (server) connected to a client system (client) by a standard TCP/IP network connection.

Setting up a PDA16 Server

In order to run a PDA16 server, the Signatec Service Manager ("Service Manager") software must be installed. At the time of this writing, this software is not part of the standard PDA16 software installation. It should be included in the 'SigSvcMgr' directory of the installation CD. This software should also be available on the Signatec website.

The server software will need to be installed on the remote machine that will host the PDA16 board(s). Also note that the normal PDA16 software must also be installed on the remote machine.

The Service Manager is designed to host arbitrary devices/services. As such, it implements service requests by deferring the calls to other service-specific libraries. Before the Service Manager can host a PDA16 service, it must be properly connected to the PDA16-specific library. The Service Manager obtains this information from a small XML file that resides in the application installation directory (C:\Program Files\Signatec\SigServerMgr by default). This configuration file is always named SigSvcCfg.xml. Here's an example configuration file that defines a PDA16 service:

```
<?xml version="1.0" encoding="utf-8"?>
<SigServiceCfg>
  <Service Index="0" Port="0">C:\Program Files\Signatec\PDA16\PDA16.dll</Service>
</SigServiceCfg>
```

If the PDA16 service is not already installed on your server system you will need to add the bold sections to your own SigSvcCfg.xml file. Note that the paths above are the paths used for default installations; if you've installed to a non-default folder you may need to alter the paths accordingly.

Once the configuration data has been specified, services are started by starting the Service Manager and selecting "Startup Services" on the main form. Service Manager has options to automatically start when the system starts up.

The Service Manager will report if there were any problems starting up any services. Also, all currently running services are displayed on the main form.

Programmatically Interfacing with a PDA16 Server

The following section documents the PDA16 library functions that are used for interacting with remote PDA16 devices. Once connected to a remote PDA16 device, the normal PDA16 library functions described above may be used. The library will take care of all remote requests and the marshalling of data.

For most users, the only function from this section that will be used is the ConnectToRemoteDeviceD16 function. The other functions are used by the underlying client/server implementation.

3.12.1 ConnectToRemoteDeviceD16

Form

```
int ConnectToRemoteDeviceD16 (HPDA16 hBrd, unsigned int brdNum, D16S_REMOTE_CONNECT_CTX* ctxp)
```

Description

Establish a connection to a PDA16 device residing on a remote computer

Parameters

[out] *hBrd*
On success, the handle pointed to by this argument will be set to a valid PDA16 device handle.

[in] *brdNum*
This parameter defines the PDA16 device in which to connect. If this number is greater than D16_MAX_DEVICES (32) then this number is assumed to be the serial number of the PDA16 in which to connect. If this parameter is less than or equal to D16_MAX_DEVICES then it is assumed to be the ordinal number of the PDA16 in the system. To connect to the first PDA16 in the system, pass 1.

[in] *ctxp*
A pointer to a D16S_REMOTE_CONNECT_CTX structure that defines connection parameters, including the PDA16 server address and port.

Return Value

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

Remarks

Once a connection to a remote PDA16 has been established, the returned handle can be used with almost any of the other PDA16 library functions. The library will automatically handle the remote requests and marshalling results; this is all totally transparent to the library user. This means that writing code for local or remote PDA16 devices is essentially the same; the primary difference being the function used to connect to the device: ConnectToDeviceD16 for local PDA16 devices and ConnectToRemoteDeviceD16 for remote PDA16 devices.

Just like using a local PDA16 device, the DisconnectFromDeviceD16 library function should be used to disconnect from a PDA16 when it is no longer in use. Calling this function will ensure proper cleanup of PDA16/network resources.

If your application is not already initializing platform-specific sockets implementation, you should call the SocketsInitD16 function as part of your application initialization to ensure that the underlying sockets implementation is initialized.

Related Functions

DisconnectFromDeviceD16, GetRemoteDeviceCountD16, SocketsInitD16

3.12.2 FreeServiceReponseD16

Form

```
int FreeServiceResponseD16(void* bufp)
```

Description

Free a response from a previous remote service request

Parameters

[in] *bufp*
A pointer to the remote service request in which to free. This is obtained by calling the SendServiceRequestD16 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 free library-allocated memory used to contain a remote service request response, which is obtained by calling SendServiceRequestD16.

Do not call FreeMemoryD16 to free this memory.

Related Functions

SendServiceRequestD16

3.12.3 GetHostServerInfoD16

Form

```
int GetHostServerInfoD16 (HPDA16 hBrd, TCHAR** server_addrpp, unsigned short* portp)
```

Description

Obtain remote server connection information.

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToRemoteDeviceD16 function.

[out] *server_addrpp*
A pointer to a TCHAR pointer that will receive the address of a library-allocated buffer containing the remote server address used to establish the remote connection. The user must free this memory when no longer in use by calling FreeMemoryD16.

[out] *portp*
A pointer to an unsigned short variable that will receive the port number on which the remote PDA16 server is running.

Return Value

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

Related Functions

GetServiceSocketD16

3.12.4 GetRemoteDeviceCountD16

Form

```
int GetRemoteDeviceCountD16 (const TCHAR* server_addrp, unsigned short port = 3490, unsigned int**
```

sn_bufpp = NULL)

Description

Obtain count (and optionally serial numbers) of PDA16 devices on a remote PDA16 server.

Parameters

[in] *server_addrp*
A pointer to a NULL-terminated string that identifies the address of the PDA16 server. This can be in URL (server.com) or dotted IP (123.45.67.89) form.

[in] *port*
The port number on the server in which to check for PDA16 services. The default port used for PDA16 servers is 3490.

[out] *sn_bufpp*
If this parameter is non-NULL and the function found PDA16 devices on the specified server, then this parameter will receive the address of a library-allocated buffer containing the serial numbers of the found PDA16 devices. Use the return value of the function to determine the number of serial numbers in the buffer. It is up to the caller to free this memory with the FreeMemoryD16 function.

Return Value

Returns the number of PDA16 devices found on the specified server or one of the library error codes (which are all negative) on error.

Related Functions

ConnectToRemoteDeviceD16

3.12.5 GetServiceSocketD16

Form

int GetServiceSocketD16 (HPDA16 hBrd, d16_socket_t* sockp)

Description

Obtain the socket handle for the underlying connection to a remote PDA16.

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToRemoteDeviceD16 function.

[out] *sockp*
A pointer to the variable that will receive the socket handle.

Return Value

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

Related Functions

GetHostServerInfoD16

3.12.6 SendServiceRequestD16

Form

int SendServiceRequestD16 (HPDA16 hBrd, const void* svc_reqp, int req_bytes, void** responsepp, unsigned int timeoutMs = 3000, unsigned int flags = 0)

Description

Send a client request to a remote PDA16 server

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToRemoteDeviceD16 function.

[in] *svc_reqp*
A pointer to a buffer containing the request. The request is XML data and may use any valid encoding (as deemed by iconv: ASCII, UTF-8, UTF16, char, wchar_t, etc)

[in] *req_bytes*
The size of the request in bytes

[out] *responsepp*
A pointer to a void* variable that will receive the address of the library allocated response buffer. Depending on the flags passed in the flags parameter, the response data will be one of the following:

If the D16SRF_AUTO_HANDLE_RESPONSE flag is specified then the response data will be service specific response value passed in the XML response data. The format of this data is UTF-8 encoded data but is safe to convert to ASCII. (Signatec authored services will only return data safe for implicit ASCII conversion.) The FreeMemoryD16 function should be used to free this memory.

If the D16SRF_AUTO_HANDLE_RESPONSE flag is not specified then the response data will be a pointer to an xmlDocPtr that points to an xmlDoc object that contains the parsed response information. This document may be interrogated via libxml2 routines. The FreeServiceResponseD16 function should be used to free this memory.

[in] *timeoutMs*
An optional timeout value. It's specified in terms of milliseconds but will most likely operate on granularity of a second.

[in] *flags*
Optional flags (D16SRF_*) that dictate behavior. Currently defined flags are:

Flag	Interpretation
D16SRF_AUTO_HANDLE_RESPONSE (1)	Auto-handle response; useful if you're only receive pass/fail info

D16SRF_NO_VALIDATION (2)	Do not validate response at all; default is a quick, cursory check
D16SRF_AUTO_FREE_REQUEST (4)	SendServiceRequestD16 will free incoming request data (not to be confused with outgoing response data).

Return Value

Returns SIG_SUCCESS (0) on success or one of the library error codes (which are all negative) on error. Note that if the D16SRF_AUTO_HANDLE_RESPONSE flag is used then this function can return an error that is generated by the remote server in response to the service request itself.

Remarks

The format of a request is:

```
<?xml version="1.0" encoding="utf-8"?>
<SigServiceRequest>
        <!-- Insert service specific request here -->
</SigServiceRequest>
```

The format of a conventional response is:

```
<?xml version="1.0" encoding="utf-8"?>
<SigServiceResponse error="false">
        <ErrorText>Error text on error</ErrorText>
        <ReturnData>Service-specific return data</ReturnData>
</SigServiceResponse>
```

Related Functions

FreeServiceResponseD16

3.12.7 SocketsCleanupD16

Form

```
int SocketsCleanupD16 (void)
```

Description

Sockets implementation cleanup; call if your application called SocketsInitD16

Return Value

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

Related Functions

SocketsInitD16

3.12.8 SocketsInitD16

Form

int SocketsInitD16 (void)

Description

Initializes platform-specific sockets implementation for the calling process.

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 once at application startup to ensure that your platform's sockets implementation is ready for use. If you already do this in your application then there is no need to call this function.

Call SocketsCleanupD16 to perform the inverse of this operation.

Related Functions

SocketsCleanupD16

3.13 Signatec Recorded Data Context (SRDC) Information

Information in this section assumes a minimum PDA16 software release version of 1.4.

The PDA16 library supports the saving of PDA16 acquisition data to external files. This is usually done by running a recording (see PDA16 Recording Session Management Routines) or by copying data from PDA16 RAM (see ReadSampleRamFastD16). In both cases, by default, only sample data is written to disk; no additional header or context information is written to the file. There are two primary reasons for this; first and foremost, it allows for fast access to the underlying data. This is important if the data is to be used for a corresponding playback and needs to be pulled off disk quickly. The other reason is that it keeps things generic; introducing extra context information may incompatible with other software systems that will read the sample data.

The disadvantage to acquisition-data-only files is that it can get confusing trying to keep up with what kind of data is in recorded data files (*.rd16) since things like channel count, segment size, etc are not stored anywhere. This is where Signatec Recorded Data Context (SRDC) information comes in. This information is additional context information that describes the data in a corresponding data file. This context data is stored in XML format and can be extended to include any number of user defined items.

The PDA16 library can generate and store this SRDC data in one of two places: an auxiliary file or in an alternate file stream of the data file. In either case, the same data is generated; it's just the location of that data that varies.

Auxiliary File

This is the most portable method of SRDC data storage. Using this method, the SRDC data is stored in a secondary file located in the same directory as the data file that it describes. By convention, the name of the file will be the name of the data file appended with '.srdc'. As an example, if your output data file was "C:\Data\MyRecording.rd16" then the corresponding SRDC data file would be named

“C:\Data\MyRecording.rd16.srdc”. This naming convention allows software to quickly check to see if SRDC information is available for a given .rd16 file.

Alternate File Stream

This method of SRDC data storage is only available on the Windows platform. Further, this method is only available when saving recorded data to a volume that uses the NTFS file system. This is the default file system for Windows-based systems.

This method of SRDC data storage takes advantage of one of the features of the NTFS file system; in particular, the ability to create alternate file streams. In a nutshell, an alternate file stream is a file “within” another file such that the original file’s data is unaffected. It allows a file to also act as a “directory” containing one or more alternate file streams. When this method is used, SRDC data is stored in a stream named “SRDC”. As an example, if your output data file was something like “C:\Data\MyRecording.rd16” then the SRDC data would be stored in “C:\Data\MyRecording.rd16:SRDC”. For more information about alternate file streams, see Microsoft website or MSDN Library Documentation.

The advantage to this method is that it keeps everything in a single file. This option is useful when recording/saving data to the same system that will be analyzing the data.

The disadvantage is that the SRD data will be lost if the file is moved to a file system that does not support alternate file streams (such as a portable flash drive) or if it is sent across a network.

Signatec Recorded Data Context Format

SDRC data is stored in XML format. The data consists of a number of named item-value pairs. Here is an example SDRC data set:

```
<?xml version="1.0" encoding="UTF-8"?>
<SignatecRecordedData>
<!--Source board information-->
  <SourceBoard>PDA16</SourceBoard>
  <SourceBoardSerialNum>100147</SourceBoardSerialNum>
<!--Data sample information-->
  <ChannelCount>1</ChannelCount>
  <ChannelId>1</ChannelId>
  <SampleSizeBytes>2</SampleSizeBytes>
  <SampleSizeBits>16</SampleSizeBits>
  <SampleFormat>Unsigned</SampleFormat>
<!--Data acquisition information-->
  <SamplingRateMHz>160</SamplingRateMHz>
  <PeakToPeakInputVoltRange>2.5</PeakToPeakInputVoltRange>
  <SegmentSize>0</SegmentSize>
  <PreTriggerSampleCount>0</PreTriggerSampleCount>
  <TriggerDelaySamples>0</TriggerDelaySamples>
<!--User-defined items-->
  <OperatorNotes>Sample operator notes....</OperatorNotes>
  <SomeUserDefinedItem>User defined data abc123</SomeUserDefinedItem>
</SignatecRecordedData>
```

A SRDC context item is defined by an item name, represented by an element, and that item’s value, represented by the element’s data. So in the data above, we can see there’s an item “ChannelCount” with a value of “1”. There are a number of predefined items that are used to define the data that this SDRC data is associated with. Users may specify any number of user defined items for their own applications.

Some notes on the XML used to represent SRDC data:

- The root element must be named SignatecRecordedData.
- Only immediate child elements of the root node are considered as context items.
- The PDA16 library considers item names to be case sensitive.
- Item names should be unique; if duplicate item names are used, only the last one will be used.

Predefined SRDC Items

Item Name	Value Interpretation	Default Value
SourceBoard	Name of the board that generated the described data. For data generated by a PDA16, this value will be "PDA16"	<None>
SourceBoardSerialNum	The serial number of the board that generated the described data.	<None>
ChannelCount	The number of channels of data in the described data. Multichannel data is assumed to sample interleaved.	1
ChannelId	The board specific channel number that generated the described data. This will be 1 for channel 1 data, 2 for channel 2 data, and so on. 0 indicates multichannel data. Some older software used -1 (or 4,294,967,295) to indicate multichannel data.	1
SampleSizeBytes	The size of a data sample in bytes. This will be 2 for all data generated by a PDA16	2
SampleSizeBits	The size of a data sample in bits. This will be 16 for all data generated by a PDA16	16
SampleFormat	The format of a data sample. May be either "Signed" or "Unsigned". PDA16 boards currently only acquire unsigned data, but software options allow for conversion to signed format.	Unsigned
SamplingRateMHz	The sampling rate, in MHz, used when acquiring the described data	0
PeakToPeakInputVoltRange	The peak-to-peak input voltage, in volts, used when acquiring the described data.	0
SegmentSize	The size of a data segment used when acquiring with the Segmented triggering mode. If data is non-segmented this value should be 0.	0
PreTriggerSampleCount	The number of pre-trigger samples in the described data. This indicates the number of samples kept prior to the trigger event that started the acquisition. This value is independent of channel count; to get the number of pre-trigger samples per-channel, divide this value by the channel count.	0
TriggerDelaySamples	The number of trigger delay samples in the described data. This is the number of samples after a trigger event that are ignored. This value is independent of channel count; to get the number of trigger delay samples per-channel, divide this value by the channel count.	0
FileFormat	The format of the data file containing the described data. May be either "Binary" or "Text". If this element is not present, it is assumed that the format is binary.	Binary
SampleRadix	The radix used to save data in text format. This value only has relevance when the *FileFormat* item is "Text'. If this element is not present, the assumed radix is 10 (decimal).	
HeaderBytes	The number of initial bytes set aside for an application defined header in the described data.	0
OperatorNotes	User-defined operator notes describing the described data set.	
RecArmTimeStr	A full string representation of date and time of when recording was armed (i.e. placed into an acquisition operating mode). This item will only be present for files generated when using the PDA16 Recording	

	Session API.	
RecArmTimeSec	Time when recording was armed, in seconds since midnight 1/1/1970. This item will only be present for files generated when using the PDA16 Recording Session API.	
RecEndTimeStr	A full string representation of date and time of when recording stopped This item will only be present for files generated when using the PDA16 Recording Session API.	
RecEndTimeSec	Time when recording was stopped, in seconds since midnight 1/1/1970. This item will only be present for files generated when using the PDA16 Recording Session API.	

The following library function are used to programmatically interact with SRDC data files

3.13.1 CloseSrdcFileD16

Form

int CloseSrdcFileD16 (HD16SRDC hFile)

Description

Close given SRDC file without updating contents

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 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 close the in-memory representation of the SRDC data. After calling this function, the given SRDC file handle is invalid.

Note that closing the SRDC file handle does not automatically flush modified content to disk. Call the SaveSrdcFileD16 function to ensure all modifications are written to disk.

Related Functions

OpenSrdcFileD16, SaveSrdcFileD16

3.13.2 EnumSrdcItemsD16

Form

int EnumSrdcItemsAD16 (HD16SRDC hFile, TCHAR** itemspp, unsigned int flags)

Description

Obtain enumeration of all SRDC items with given constraints

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

[out] *itemspp*
A pointer to a TCHAR pointer that will receive the address of a library allocated buffer containing a space-delimited list of all SRDC context items with constraints defined by the flags parameter. If there are no items to enumerate, an empty string will be returned.

[in] *flags*
A set of flags that determine which items to include in the enumeration

Flag	Value	Interpretation
D16SRDCENUM_SKIP_STD	0x00000001	Skip standard SRDC items; use to obtain only user-defined items
D16SRDCENUM_SKIP_USER_DEFINED	0x00000002	Skip user-defined SRDC items; use to obtain only standard items
D16SRDCENUM_MODIFIED_ONLY	0x00000004	Only include modified items in enumeration

Return Value

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

Remarks

This library function will return an enumeration of currently defined SRDC items in the form of a space-delimited string in case-sensitive, alphabetical-ascending order.

3.13.3 GetRecordedDataInfoD16

Form

```
int GetRecordedDataInfoD16 (const TCHAR* pathnamep, D16S_RECORDED_DATA_INFO* infop,
TCHAR** operator_notespp)
```

Description

Obtain basic SRDC information on acquisition data in given file

Parameters

[in] *pathnamep*
A pointer to a NULL-terminated string containing the pathname of the acquisition data (*.rd16).

[in] *infop*
A pointer to a D16S_RECORDED_DATA_INFO structure that will receive some most basic SRDC information for the acquired data.

[out] *operator_notespp* = NULL

The address of a TCHAR* that will receive the address of a library-allocated buffer containing the operator notes for the acquired data. If no operator notes have been specified, the library will write NULL. The caller should free the memory for this string when no longer needed by calling the FreeMemoryD16 function. Pass NULL if operator notes are not required.

Return Value

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

Remarks

The caller should zero out the D16S_RECORDED_DATA_INFO and initialize the struct_ver member with the current structure version (D16SVER_RECORDED_DATA_INFO).

The library will attempt to find the SRDC information for the given acquisition data file. It will first check to see if an external file is present (pathname + ".srdc"), then check for an alternate file stream (pathname + ":SRDC"), and lastly, check the given file itself.

3.13.4 GetSrdcItemD16

Form

```
int GetSrdcItemD16 (HD16SRDC hFile, const TCHAR* namep, TCHAR** valuepp)
```

Description

Look up SRDC item with given name; name is case-sensitive

Parameters

[in] *hFile*

A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

[in] *namep*

A pointer to a NULL-terminated string containing the case-sensitive name of the context item to find the value for.

[out] *valuepp*

A pointer to a TCHAR pointer that will receive the address of the library allocated buffer containing the value for the given named SRDC item. If no value has been specified for this name the library will return a NULL address. It is the caller's responsibility to free the storage for the item value when no longer needed; use the FreeMemoryD16 function. NULL may be passed for this parameter if the item value is not needed.

Return Value

Returns SIG_SUCCESS (0) on success or one of the library error codes (which are all negative) on error. If the named item does not exist in the SRDC data, SIG_D16_NAMED_ITEM_NOT_FOUND will be returned.

Related Functions

SetSrdcItemD16

3.13.5 IsSrdcFileModifiedD16

Form

int IsSrdcFileModifiedD16 (HD16SRDC hFile)

Description

Determine if given SRDC file data has been modified

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

Return Value

Returns > 0 if given SRDC data has been modified, 0 if SRDC data has not been modified, or one of the library error codes (which are all negative) on error.

Remarks

The SetSrdcItemD16 function is used to modify the value of a specific SRDC data item. The RefreshSrdcParametersD16 function can also change one or more standard SRDC data item values.

Saving SRDC data via the SaveSrdcFileD16 function will reset the modified state of all SRDC items.

3.13.6 OpenSrdcFileD16

Form

int OpenSrdcFileD16 (HPDA16 hBrd, HD16SRDC* handlep, const TCHAR* pathnamep, unsigned flags)

Description

Open a new or existing .srd file

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function. This parameter may be D16_INVALID_HANDLE to not associate a PDA16 device with the SRDC file. A valid board handle is only required if RefreshSrdcParametersD16 will be used on the SRDC file.

[out] *handlep*
A pointer to a HD16SRDC variable that will receive a handle that identifies the SRDC file.

[in] *pathnamep*
A pointer to a NULL-terminated string that contains the pathname of the SRDC context file. This can be a path to a .srdc file (e.g. C:\Path\To\RecordingData.rd16.srdc) or an alternate file stream containing SRDC data (e.g. C:\Path\To\RecordingData.rd16:SRDC).

[in] *flags*

A set of flags (D16SRDOF_*) that define function behavior. Currently defined flags are:

Flag	Value	Interpretation
D16SRDCOF_QUICK_SET	0x00000001	Opens/create file, refresh and write settings, then close file. See Remarks.
D16SRDCOF_OPEN_EXISTING	0x00000002	Open existing SRDC file; function will fail if file does not exist
D16SDRCOF_CREATE_NEW	0x00000004	Create a new SDRC file; will ignore and overwrite previously existing data
D16SRDCOF_PATHNAME_IS_REC_DATA	0x00000008	Given pathname is recorded data file; have library search for SRDC data. Using this flag implies D16SRDCOF_OPEN_EXISTING. This flag cannot be used with D16SDRCOF_CREATE_NEW

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 open or create a new Signatec Recorded Data Context file. Once the file has been opened, context item-value pairs can be read or written.

Flags D16SRDCOF_OPEN_EXISTING and D16SDRCOF_CREATE_NEW are mutually exclusive. The library will return an error (SIG_D16_INVALID_ARG_4) if both are set. Also note that use of D16SRDCOF_PATHNAME_IS_REC_DATA implies D16SRDCOF_OPEN_EXISTING.

If the D16SRDCOF_QUICK_SET flag is set, then the function will open the file (if it exists), refresh standard SRDC context items (by calling RefreshSrdcParametersD16), save changes, and close the file. In this case, no SRDC file handle is returned.

Related Functions

SaveSrdcFileD16, CloseSrdcFileD16, SetSrdcItemD16, GetSrdcItemD16, RefreshSrdcParametersD16

3.13.7 RefreshSrdcParametersD16

Form

```
int RefreshSrdcParametersD16 (HD16SRDC hFile, unsigned flags)
```

Description

Refresh SRDC data with current board settings; not written to file

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

[in] *flags*
No flags have been defined yet; pass 0.

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 refresh current SRDC data with current hardware settings. This includes the following SDRC items: SourceBoard, SourceBoardSerialNum, ChannelCount, ChannelId, SampleSizeBytes, SampleSizeBits, SampleFormat, SamplingRateMHz, PeakToPeakInputVoltRange, SegmentSize, PreTriggerSampleCount, and TriggerDelaySamples.

Note that these changes are only written to the in-memory representation of the SDRC data. Changes will not be saved to disk until the SaveSrdcFileD16 function is called.

Related Functions

OpenSrdcFileD16, SaveSrdcFileD16

3.13.8 SaveSrdcFileD16

Form

```
int SaveSrdcFileD16 (HD16SRDC hFile, const TCHAR* pathnamep)
```

Description

Write SRDC data to file

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

[in] *pathnamep*
A pointer to a NULL-terminated string containing the pathname of the output file. This can also be the pathname of an alternate file stream in which to save the SDRC data. This parameter may be NULL if the SRDC file was opened with an explicit pathname, or if the data has already been saved to a file with an earlier call to this 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 flush all SRDC data to a file (or alternate file stream). After calling this function, all 'modified' flags will be reset.

Related Functions

OpenSrdcFileD16

3.13.9 SetSrdcItemD16

Form

int SetSrdcItemD16 (HD16SRDC hFile, const TCHAR* namep, const TCHAR* valuep)

Description

Add/modify SRDC item with given name; not written to file

Parameters

[in] *hFile*
A handle to a SRDC file previously opened by calling the OpenSrdcFileD16 function.

[in] *namep*
A pointer to a NULL-terminated string containing the case sensitive name of the context item to modify.

[in] *valuep*
A pointer to a NULL-terminated string containing the value to be associated with the given name. If this parameter is NULL, then the given named item will be removed from the SRDC data.

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 only changes the in-memory representation of the SDRC data. Changes will not be written to disk until the SaveSrdcFileD16 function is called.

Modifying the value a SRDC item will result in that particular item's internal "modified" bit to be set.

Related Functions

OpenSrdcFileD16, GetSrdcItemD16, SaveSrdcFileD16

3.14 PDA16 Timestamp Management Routines

Information in this section assumes a minimum PDA16 firmware version of 1.4 and a minimum PDA16 software release version of 1.4.

The PDA16 has the ability to generate timestamps during the data acquisition process.

3.14.1 GetTimestampAvailabilityD16

Form

int GetTimestampAvailabilityD16 (HPDA16 hBrd)

Description

Determine if any timestamps are available in the PDA16 timestamp FIFO

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

Returns 0 if the timestamp FIFO is currently empty (indicating that no timestamps are available to be read), a positive value if the timestamp FIFO is not empty, or one of the negative library error codes on error.

Remarks

This function is used to determine if any timestamps are available to be read.

Related Functions

GetTimestampOverflowFlagD16, ReadTimestampDataD16

3.14.2 GetTimestampFifoDepthD16

Form

int GetTimestampFifoDepthD16 (HPDA16 hBrd, unsigned int* ts_elementsp)

Description

Obtain the size of the PDA16 timestamp FIFO, in timestamp elements

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[out] *ts_elementsp*
A pointer to the unsigned int variable that will receive the size of the PDA16 timestamp FIFO

Return Value

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

Remarks

The standard PDA16 timestamp FIFO size is 2048 timestamps. Each timestamp is an unsigned 64-bit integer.

3.14.3 GetTimestampOverflowFlagD16

Form

int GetTimestampOverflowFlagD16 (HPDA16 hBrd)

Description

Read Timestamp Overflow Flag from PDA16 hardware

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

Return Value

Returns 0 if the PDA16 timestamp FIFO is not full, a positive value of the PDA16 timestamp FIFO is full, or one of the library error codes (which are all negative) on error.

Remarks

If the timestamp FIFO goes full, then one or more timestamps may have been lost.

Related Functions

GetTimestampAvailabilityD16, ReadTimestampDataD16

3.14.4 ReadTimestampDataD16

Form

int ReadTimestampDataD16 (HPDA16 hBrd, pda16_timestamp_t* bufp, unsigned int ts_count, unsigned int* ts_readp, unsigned int flags = 0, unsigned int timeout_ms = 0, unsigned int* flags_outp = NULL)

Description

Read timestamp data from PDA16 timestamp FIFO

Parameters

[in] *hBrd*
A handle to the PDA16 board. This handle is obtained by calling the ConnectToDeviceD16 function.

[out] *bufp*
A pointer to the buffer that will receive the timestamp values

[in] *ts_count*
The size of the buffer pointed to by the bufp parameter, in timestamp elements.

[out] *ts_readp*

A pointer to an unsigned int variable that will receive the number of timestamps read from the timestamp FIFO. This parameter may not be NULL.

[in] *flags*

A set of flags (D16TSREAD_*) that define function behavior. Currently defined flags are in the following table.

Library Constant	Value	Interpretation
D16TSREAD_READ_FROM_FULL_FIFO	0x00000001	Set to read from a known-full timestamp FIFO

[in] *timeout_ms*

This value is currently ignored, but may be used in a future to specify a timeout for a read operation.

[out] *flags_outp*

A pointer to an unsigned int variable that will receive output flags describing the state of the timestamp FIFO after the read operation. These flags can be used to determine if another timestamp read operation can take place immediately. Currently defined flags are in the following table.

Library Constant	Value	Interpretation
D16TSREAD_MORE_AVAILABLE	0x01000000	Timestamp FIFO is not empty after read; more timestamps are available
D16TSREAD_FIFO_OVERFLOW	0x02000000	Timestamp FIFO was full prior to the read operation.

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 may not be necessary. The PDA16 library can be setup to automatically download and save timestamps during a recording session, or when using ReadSampleRamFileFastD16/ ReadSampleRamFileD16. See documentation for D16S_FILE_WRITE_PARAMS structure for details on this.

This function will read up to *ts_count* timestamps from the PDA16 timestamp FIFO. If the timestamp FIFO goes empty before reading *ts_count* timestamps, the function will return; it will not wait for more timestamps to accumulate.

It is safe to call this function while an acquisition or recording is in progress.

Timestamps will be generated based on the rules of the current timestamp mode, which is set by the SetTimestampModeD16 function.

If the D16TSREAD_READ_FROM_FULL_FIFO flag is not specified and the timestamp FIFO is full, then this function will return success (0) and the variable pointed to by the ts_readp parameter will be set to zero. The D16TSREAD_FIFO_OVERFLOW output flag will also be set. To read from a known full FIFO, set the D16TSREAD_READ_FROM_FULL_FIFO flag.

Related Functions

SetTimestampModeD16

3.15 PDA16 Library Data Types

The following sections define some of the data types used by the PDA16 library

3.15.1 Data Type: HPDA16

This data type is used to represent a handle to a PDA16 device.

A special library constant INVALID_PDA16_HANDLE (NULL) represents an invalid PDA16 device handle. Any other value should be considered opaque; this value only has relevance to the PDA16 library implementation.

PDA16 handles are only valid in the process for which they are obtained.

The DisconnectFromDeviceD16 library function should be called to release a PDA16 device handle when it is no longer needed. This will free handle-specific resources allocated by the library.

3.15.2 Data Type: HD16RECORDING

This data type is used to represent a handle to a PDA16 Recording Session.

A special library constant INVALID_HD16RECORDING_HANDLE (NULL) represents an invalid PDA16 Recording Session handle. Any other value should be considered opaque; this value only has relevance to the PDA16 library implementation.

PDA16 Recording Session Handles are only valid in the process for which they are obtained.

3.15.3 Data Type: HD16SRDC

This data type is used to represent a handle to a Signatec Recorded Data Context (SRDC) file.

A special library constant D16_INVALID_HD16SRDC (NULL) represents an invalid SRDC file handle. Any other value should be considered opaque; this value is only has relevance to the PDA16 library implementation.

3.15.4 Data Type: pda16_sample_t

This data type is used to represent a PDA16 data sample. It is typically defined as an ‘unsigned short’ (unsigned 16-bit integral value).

Currently, the PDA16 will only generate little-endian, unsigned data.

3.15.5 Data Type: pda16_timestamp_t

This data type is used to represent a PDA16 timestamp value. It is typically defined as an ‘unsigned long long’ (unsigned 64-bit integral value) and is little-endian.

Timestamp values are taken from a counter that is incremented at each tick of the acquisition clock.

3.15.6 Structure: D16S_FILE_WRITE_PARAMS

This structure is used by a few of the PDA16 library functions that save PDA16 board data (either from PDA16 RAM or from a PDA16 recording session) to one or more files on the host system.

Structure definition

```
typedef struct _D16S_FILE_WRITE_PARAMS_tag
{
        unsigned int      struct_ver;              // Current struct version is 2

        const char*       pathname;                ///< Destination file pathname
        const char*       pathname2;               ///< For dual channel
        unsigned int      flags;                   ///< D16FILWF_*
        unsigned int      init_bytes_skip;         ///< Initial bytes to skip in file
        unsigned long long max_file_seg;           ///< Max file size in samples; binary only

        D16_FILEIO_CALLBACK     pfnCallback;     ///< Optional progress callback
        uintptr_t               callbackCtx;     ///< User-defined callback context

        // Members below this only considered when struct_ver > 1

        unsigned int*           flags_out;       ///< Output flags; D16FILWOUTF_*
        const char*             operator_notes;  ///< User-defined notes
        const char*             ts_filenamep;    ///< Timestamp filename override

} D16S_FILE_WRITE_PARAMS;
```

Fields

struct_ver

This field will be used to discriminate against possible future versions of this structure. Portable code should zero out the contents of the entire structure and set this member to D16SVER_FILE_WRITE_PARAMS as an initialization step.

pathname

A pointer to a NULL-terminated char string that defines the pathname to use for output data.

pathname2

A pointer to a NULL-terminated char string that defines the pathname to use for channel 2 data. This parameter may be NULL.

flags

A set of flags that define how data will be saved. Currently defined flags are:

Library constant	Value	Interpretation
D16FILWF_APPEND	0x00000001	Append if file already exists; default is to overwrite existing file.
D16FILWF_ALWAYS_SKIPBYTES	0x00000002	Use *bytes_skip* even if appending to a file. By default, if appending to an existing file then the *bytes_skip* field is ignored.
D16FILWF_SAVE_AS_TEXT	0x00000004	Save data as text; sample values will be whitespace-delimited. This is not a fast operation, so it's not recommended for use with recording PCI acquisition data.
D16FILWF_NO_UNBUFFERED_IO	0x00000008	*Windows*: Don't try to use fast,

		unbuffered IO. If this flag is not set then the library may try to use the FILE_FLAG_NO_BUFFERING flag when creating the file. Using this flag can significantly improve file IO performance but requires file access lengths/offsets to be aligned to the underlying volume's sector size. If the library cannot guarantee these requirements, then normal, buffered file IO is used. See Win32 CreateFile function for details on this.
D16FILWF_ASSUME_DUAL_CHANNEL	0x00000010	Assume data is dual channel; applies to single-file text saves. If this flag is set and data is being saved to a single text file then channel 1 data will be listed in one column and channel 2 data will be in another column.
D16FILWF_HEX_OUTPUT	0x00000020	Use hexadecimal output; applies to text saves
D16FILWF_DEINTERLEAVE	0x00000040	De-interleave data into separate files. If this flag is set then the library will split channel data into separate files. Channel 1 data will be written to the file specified by the *pathname* field. Channel 2 data will be written to the file specified by the *pathname2* field. Either one of these parameters can be NULL if that channel is not needed.
Fields below this row require PDA16 library version 1.3.3 or higher.		
D16FILWF_CONVERT_TO_SIGNED	0x00000080	Convert data to signed format before saving. When this flag is set, data will be converted to equivalent signed value before being written. For signed data, the minimum ADC value is -32768, the maximum ADC value is 32767, and the middle ADC value is 0. Note that this conversion is done in the software, not the hardware so it may affect IO throughput. If this flag is not specified then native, unsigned data will be saved.
D16FILWF_GENERATE_SRDC_FILE	0x00000100	Generate a Signatec Recorded Data Context (.srdc) file for each output file.
D16FILWF_EMBED_SRDC_AS_AFS	0x00000200	Embed Signatec Recorded Data Context information in data file as NTFS alternate file stream (:SRDC)
D16FILWF_SAVE_TIMESTAMPS	0x00000400	Save PDA16 timestamp data in external file
D16FILWF_TIMESTAMPS_AS_TEXT	0x00000800	Save timestamps as newline-

		delimited text; use with D16FILWF_SAVE_TIMESTAMPS flag
D16FILWF_USE_TS_FIFO_OVFL_MARKER	0x00001000	Use timestamp FIFO overflow marker: F1F0F1F0F1F0F1F0 F1F0F1F0F1F0F1F0
D16FILWF_ABORT_OP_ON_TS_OVFL	0x00002000	Abort and fail operation if timestamp FIFO overflows; meant to be used with recording sessions

init_byte_skip

Specifies the number of bytes to skip in the file before writing the first chunk of data. This can be used to have the library leave an application-defined blank area of the file than can be used for application-specific header data. The data in the skipped region will be all zeroes. Pass zero if you do not want to skip any bytes in the output file(s).

max_file_seg

In some applications, it is preferable to write to multiple, statically-sized files rather than one large file. This member is used to specify this behavior. If this member is zero, then all data is written to a single file. (Or, in the case of dual channel data, two files.)

If this member is non-zero, then it is the maximum size, in samples, that the library will write before creating a new file. In this case, the pathname(s) that are specified are used as a base for the resultant files that are created by the library routine. The library uses the given pathname appended with an underscore ('_') and an increasing number, beginning with 0.

This parameter is only considered when writing binary files. When writing a text file, this parameter is ignored.

pfnCallback

If non-NULL, this member is the address of a user-defined callback function that the library will call prior to writing each block of data to file. This callback function is intended to give applications a chance to handle progress updates. This callback function is documented in the Callback Function: D16_FILEIO_CALLBACK section.

callbackCtx

An application-defined value that is passed to user-defined callback function.

Members below this point are only considered when struct_ver member > 1 and requires a minimum PDA16 library version of 1.3.3

flags_out

The library will write output flags to this member to convey additional information about the operation. NOTE: If this structure is being used for a recording session, use the GetRecordingSessionOutFlagsD16 library function to obtain this information after stopping the recording and before deleting the recording session. Currently defined flags are listed in the following table.

PDA16 Library Constant	Value	Interpretation
D16FILWOUTF_TIMESTAMP_FIFO_OVERFLOW	0x00000001	Timestamp FIFO overflowed during write/record process
D16FILWOUTF_NO_TIMESTAMP_DATA	0x00000002	No timestamp data was available during the operation

operator_notes

A pointer to a NULL-terminated string that defines a user-defined operator note that will be saved when generating Signatec Recorded Data Context (SRDC) for the acquisition data being saved. SDRC data is generated when either D16FILWF_GENERATE_SRDC_FILE or D16FILWF_EMBED_SRDC_AS_AFS is specified in the flags field.

ts_filenamep

An optional override for the timestamp data file. This member is only considered when the D16FILWF_SAVE_TIMESTAMPS flag is set in the flags member. See remarks below for details on how the timestamp data file is named.

Remarks

PDA16 Timestamps

If the D16FILWF_SAVE_TIMESTAMPS flag is set in the flags member then the library will automatically read and save timestamp data from the PDA16 timestamp FIFO during the data recording. Timestamp data will be inserted into the timestamp FIFO by the PDA16 based on the rules defined by the current timestamp mode, which is specified via the SetTimestampModeD16 library function.

The *ts_filenamep* member may be used to specify the timestamp data file pathname. Or, if *ts_filenamep* is NULL, the library will name the timestamp data file as follows:

- If the *pathname* member is not NULL, *pathname* will be used as the base pathname.
- If the *pathname* member is NULL and the *pathname2* member is not NULL, *pathname2* will be used as the base pathname.
- If both *pathname* and *pathname2* are NULL, an error will be returned.
- Once the base pathname has been determined, the library will be appended with a '.pda16ts' extension and this pathname will be used for the timestamp data file.
- Further, if the D16FILWF_TIMESTAMPS_AS_TEXT flag is specified then a '.txt' extension will be appended. (The net effect of this is *base pathname* + ".pda16ts.txt")

During the recording session, the library will create and manage a secondary thread that will be responsible for pulling and saving timestamp data.

PDA16 Timestamp Data File Format

If the D16FILWF_TIMESTAMPS_AS_TEXT flag is set then the library will write all timestamp data to a text file with a new-line ('\n') between each timestamp value. The timestamp values will be written in decimal format.

If the D16FILWF_TIMESTAMPS_AS_TEXT flag is not set then the library will write all timestamp data to a binary file. There is no header or auxiliary data written to the file; the first 64-bits of the file is the first timestamp value.

PDA16 Timestamp FIFO Overflow

By default, if the Timestamp FIFO overflows at any point, the operation will continue. There are two flags that can be specified to alter this behavior.

If the D16FILWF_ABORT_OP_ON_TS_OVFL flag is specified during a recording session and a Timestamp FIFO overflow condition occurs, the recording will be abort and end with an error code of SIG_D16_TIMESTAMP_FIFO_OVERFLOW (-573). Timestamps will continue being recorded in this case until stopped by the main internal recording thread.

If the D16FILWF_USE_TS_FIFO_OVFL_MARKER flag is specified and a Timestamp FIFO overflow condition occurs then the library will do the following:

- Read and save all timestamp values in the FIFO.
- Write a "timestamp overflow marker" to the timestamp data file. The timestamp overflow marker is two sequential timestamp values with the following value: 0xF1F0F1F0F1F0F1F0 (or 17,433,700,174,704,734,704 in decimal).

By inserting a timestamp overflow marker, third-party software can programmatically detect and handle FIFO overflow conditions.

It should be noted that if the Timestamp FIFO goes full, this does not guarantee that any timestamp data has been lost. If any timestamps are generated while the FIFO is full, they are lost.

Related Functions

ReadSampleRamFileFastD16, ReadSampleRamFileD16, CreateRecordingSessionD16

3.15.7 Structure: D16S_REC_SESSION_PARAMS

This structure is used by the CreateRecordingSessionD16 function to specify the various parameters that define how a PDA16 recording session will run.

Structure definition

```
typedef struct _D16S_REC_SESSION_PARAMS_tag
{
    unsigned int            struct_ver;         // Must be 1 for now

    int                     rec_flags;          ///< D16RECSESF_*
    unsigned long long      rec_samples;        ///< # samples or 0 for infinite
    unsigned int            acq_samples;        ///< For RAM acq recordings
    unsigned int            xfer_samples;       ///< Transfer size of 0 for autoset

    D16S_FILE_WRITE_PARAMS*     filwp;          ///< Defines output file(s)

    pda16_sample_t*         dma_bufp;           ///< Optional DMA buffer address
    unsigned                dma_buf_samples;    ///< Size of DMA buffer in samps

    // Data snapshot parameters; valid when D16RECSESF_DO_SNAPSHOTS set

    unsigned int            ss_len_samples;     ///< Data snapshot length in samples
    unsigned int            ss_period_xfer;     ///< Snapshot period in DMA xfers
    unsigned int            ss_period_ms;       ///<  or in milliseconds

    D16_REC_CALLBACK        pfnCallback;        ///< Optional callback
    uintptr_t               callbackData;       ///< Context data for callback

} D16S_REC_SESSION_PARAMS;
```

Fields

struct_ver

This field will be used to discriminate against possible future versions of this structure. This field should be initialized to 1 by the caller.

rec_flags

A set of flags that define PDA16 recording session behavior. Currently defined flags are:

Library Constant	Value	Interpretation
D16RECSESF_REC_PCI_ACQ	0x00000001	Do a PDA16 RAM-buffered PCI acquisition recording
D16RECSESF_REC_RAM_ACQ	0x00000002	Do a PDA16 RAM acquisition/transfer recording
D16RECSESF_REC__MASK	0x0000000F	Mask for session recording type
D16RECSESF_DO_NOT_ARM	0x00000010	Do not auto arm recording; will be done with ArmRecordingSessionD16
D16RECSESF_DO_SNAPSHOTS	0x00000020	Periodically obtain snapshots of recording data

See remarks below for more details on these flags

rec_samples

Defines the total amount of samples to record. For dual channel recordings, this count is the total samples over both channels. If zero is specified for this parameter then the recording will run indefinitely. In this case, the AbortRecordingSessionD16 can be used to stop the recording.

acq_samples

For RAM acquisition/transfer recordings, this parameter defines the total RAM acquisition size in samples. For dual channel acquisitions, this count is the total samples to acquire over both channels. This parameter has the same constraints as the samp_count parameter of the AcquireToBoardRamD16 function. This parameter is ignored for PCI-based recordings.

xfer_samples

Defines the DMA transfer length that should be used for PCI-based acquisition recordings. Pass zero to have the library determine what the transfer size should be.

filwp

A pointer to a D16S_FILE_WRITE_PARAMS structure that defines how data will be saved to persistent storage. See the Structure D16S_FILE_WRITE_PARAMS section for details on this structure. Note that the PDA16 library supports many different ways of saving data and some of these (saving as text for instance) can be quite slow. For PCI-based recordings, if data cannot be written fast enough, FIFOs will overflow and data will be lost.

dma_bufp

A pointer to a previously allocated DMA buffer to use for transfer of acquisition data. If NULL is passed for this parameter, the library will allocate a DMA buffer to be used for the duration of the recording session.

dma_buf_samples

The size, in samples, of the DMA buffer pointed to by the *dma_bufp* field.

ss_len_samples

Defines the length, in samples, of the recording data snapshot. This field is only considered when the D16RECSESF_DO_SNAPSHOTS flag is specified in the *rec_flags* field.

ss_period_xfer

The recording data snapshot update period, in DMA transfers. For long, non-segmented PCI acquisitions, a snapshot period in DMA transfers can be used to obtain synchronized recording snapshots over multiple

independently running recording sessions. This field is only considered when the D16RECSESF_DO_SNAPSHOTS flag is specified in the *rec_flags* field.

ss_period_ms
The recording data snapshot update period, in milliseconds. If the ss_period_xfer field is zero, the PDA16 library will default back to a simple, approximate-timed approach for recording data snapshot updates. This field is only considered when the D16RECSESF_DO_SNAPSHOTS flag is specified in the *rec_flags* field.

pfnCallback
A pointer to an optional callback function that the library will periodically call during the recording session. This callback is intended to provide recording progress indications.

callbackData
An application specific value that is passed to the callback function specified by the *pfnCallback* field.

Remarks

The PDA16 library currently supports two types of recording sessions: PCI Acquisition and RAM Acquisition/Transfer.

PCI Acquisition Recording Sessions

A PCI Acquisition recording session uses the PDA16's RAM-Buffered PCI Acquisition operating mode to acquire data directly to the PCI bus, using the onboard RAM as a large FIFO to help buffer the data against software latencies introduced by non real-time environments like Windows and Linux.

A PCI Acquisition recording session is specified by setting the D16RECSESF_REC_PCI_ACQ flag in the *rec_flags* field.

RAM Acquisition/Transfer Recording Sessions

A RAM Acquisition/Transfer recording session uses the PDA16's RAM Acquisition operating mode to acquire data into PDA16 RAM. Once the desired number of samples has been acquired, the acquisition is stopped and the data is transferred to the host system via DMA transfer. This process is repeated until the total number of samples has been recorded.

It is important to note that with this type of recording, multiple, independent acquisitions are performed so if a continuous, uninterrupted stream of data is needed the PCI Acquisition recording session should be used.

A RAM Acquisition/Transfer recording session is specified by setting the D16RECSESF_REC_RAM_ACQ flag in the *rec_flags* field.

3.15.8 Structure: D16S_REMOTE_CONNECT_CTX

This structure is used in the ConnectToRemoteDeviceD16 function and is used to specify server information for the remote connection.

Structure definition

```
typedef struct _D16S_REMOTE_CONNECT_CTX_tag
{
    unsigned int        struct_ver;         // D16SVER_REMOTE_CONNECT_CTX
```

```
    unsigned int        flags;              // Currently undefined, use 0
    unsigned short      port;
    const TCHAR*        pServerAddress;
    const TCHAR*        pApplicationName;   // Optional
    const TCHAR*        pSubServices;       // Optional

} D16S_REMOTE_CONNECT_CTX;
```

Fields

struct_ver
: The version of this structure. Portable code should zero-out the structure and initialize this field with D16SVER_REMOTE_CONNECT_CTX before calling ConnectToRemoteDeviceD16.

flags
: At the time of this writing, no flags are currently defined. Pass zero.

port
: The port number on which the PDA16 server is running. The preferred PDA16 server port number is defined as D16_SERVER_PREFERRED_PORT (3490) but the server may run on any valid port.

pServerAddress
: A pointer to a NULL-terminated string containing the server's network address. This can be in dotted (123.45.67.89) or URL (server.com) form.

pApplicationName
: A pointer to a NULL-terminated string containing the client application's name. This is purely for informational purposes; the server may choose to use this name when displaying diagnostic or debugging information. This parameter may be NULL.

pSubServices
: A pointer to a NULL-terminated string containing a semi-colon delimited string of additional services to connect to. Certain servers implement sub-services for things like file-sharing. This parameter may be NULL.

Remarks

The structure definition above is an effective definition; D16S_REMOTE_CONNECT_CTXA is used for ASCII builds and D16S_REMOTE_CONNECT_CTXW is used for UNICODE builds. The structures are identical with the exception that the former uses 'const char*' strings and the latter uses 'const wchar_t*' strings. See library header for details.

3.15.9 Callback Function: D16_FILEIO_CALLBACK

This item defines the signature of a callback function that is used when saving PDA16 data to file.

Prototype

int SomeCallbackFunction (HPDA16 hBrd, uintptr_t callbackCtx, pda16_sample_t* sample_datap, int sample_count, const char* pathname, unsigned long long samps_in_file, unsigned long long samps_moved, unsigned long long samps_to_move);

Arguments

hBrd

A handle to the PDA16 device for which data is being saved

callbackCtx

A user-defined value that is ignored by the library.

sample_datap

A pointer to the chunk of data that is about to be written to file.

sample_count

The number of samples in the buffer pointed to by the *sample_datap* argument.

pathname

The pathname of the file currently being written to.

samps_in_file

The total number of samples written to the current file so far

samps_moved

The total number of samples written/recorded so far. In cases where only a single file is being written, this will be the same value as the *samps_in_file* argument. In cases where board data is written to multiple files, this argument contains the total amount of data written over all files up to this point.

samps_to_move

The total number of samples that are to be written over the entire operation. If this value is zero, then the library is writing an infinite (e.g. manually stopped) amount of data.

Return Value

This function should return SIG_SUCCESS on success. Any other return value will result in the invoking library function to stop the current save operation (cleaning up as necessary) and returning that error code to the top-level caller.

4 APPENDIX A - PDA16 Specifications & Ordering Information

External Signal Connections

Analog Input, Channel 1	: SMA
Analog Input, Channel 2	: SMA
Clock Input	: SMA
Trigger Input	: SMA
Digital Output	: SMA
Ethernet (option)	: RJ45

Analog Inputs

Full Scale Volt. Ranges	: 2.50V, 1.67V, 1.00V, 667mV, : 400mV, 267mV
Impedance	: 50 ohms
Bandwidth	: 700 MHz (with no LP filter)
Coupling	: AC

External Trigger

Signal Type	: LVTTL or LVCMOS (0 to 3.3V)
Impedance	: >1k ohms
Bandwidth	: 50 MHz

Internal Synthesized Clock

Frequency range	: 45.0 - 170 MHz
Resolution	: better than 5 PPM
Accuracy	: better than 5 PPM
Unsettable ranges	: 128.9-129.8, 140.6-142.8, 154.7-158.7 MHz

External Clock

Signal Type	: sine wave or square wave
Coupling	: AC
Impedance	: 50 ohms
Frequency	: 10 MHz to 160 MHz
Amplitude	: 200 mV p-p to 2.0 V p-p

Post ADC Clock Divider

Divider Settings	: 1, 2, 4, 8, 16, 32

Reference Clock

Internal	: 10.0 MHz, ± 5 ppm max.
External	: 10.0 MHz, ± 50 ppm max (required for lock)

Digital Output

Type	: TTL Logic Level
Max. Frequency	: 160 MHz
Suggested Load	: 1k ohms

Digitizer

Resolution	: 16 bits
Aperture Jitter	: 0.07 pS typical
Clock Rate	: 1.0 to 160 MHz

Trigger Modes

Post Trigger	: single start trigger fills active memory
Segmented	: start trigger for each memory segment

Trigger Options

Pretrigger Samples	: samples prior to trigger are stored; Single Channel: 8k max.; Dual Channel: 4k max per channel
Delayed Trigger	: delay from trigger to data storage; Up to 64k digitizer clock cycles

Memory

Active Size	: Up to 256 MegaSamples
Segment Size	: Up to 128 Megasamples
Segment re-arm time[1]	: 150 nanoseconds
Addressing	: DMA transfer from starting address

I/O Addressing

PCI Controller Address	: 64 bytes, Plug and Play selected

Signatec Auxiliary Bus

Data Transfer Modes	: Block
Data Transfer Rates	: up to 640 MB/s max @ 64 bits
Data Direction	: output only

Power Requirements

+12V	: 400 mAmps max.
+5V	: 1.5 Amps max.
+3.3V	: 2.3 Amps max.

Absolute Maximum Ratings

Analog Inputs	: ±5 volts
Trigger Input	: -0.2 to +4.0 volts DC
Clock Input	: 5 volts peak to peak
Ambient Temperature	: 0 to 50 C

PDA16 Part Numbers

PDA16-[*FX*]-[*LPF*]-[*INTERFACE*]-[*MS*]

[*FX*]: **20** = Virtex-4 FX20 device.
60 = Virtex-4 FX60 device.

[*LPF*] = Cutoff frequency for the low pass filter in MHz. Standard options are **120** and **210**. Specify other values for custom filters. For no upper cutoff, this [LPF] should be **0**.

[*INTERFACE*]: **P33** = PCI 64-bit 33MHz.
X66 = PCI-X 64-bit 66MHz.
X100 = PCI-X 64-bit 100MHz.

[*MS*] = Add **MS** for Master/Slave configuration requirement.

Example Part Order Numbers (**-MS** may be added to any of these numbers):

PDA16-20-0-P33, PDA16-20-0-X66, PDA16-20-0-X100
PDA16-20-120-P33, PDA16-20-120-X66, PDA16-20-120-X100
PDA16-20-210-P33, PDA16-20-210-X66, PDA16-20-210-X100
PDA16-60-210-P33, PDA16-60-210-X66, PDA16-60-210-X100

SAB Cables

Refer to the "SAB Cable Assembly Ordering Guide" to select and order the appropriate cable assemblies.

Master-Slave Cables

The PDA16 may be software configured to operate as a Master or a Slave in a multiple board system. For Master/Slave operation a 16-pin ribbon cable is required to connect the boards. To specify master/slave configuration, order part number with ending of **–MS**. Master/Slave boards must occupy adjacent slots. The maximum number of boards to be connected is one master and three slaves.

Documentation & Accessories

The PDA16 is supplied with a comprehensive operator's manual, which thoroughly describes the operation of both the hardware and the software. Also supplied are two four-foot coaxial cables with SMA to BNC connectors. Additional cables may be purchased. Supplied software disks contain a function library for Microsoft Visual C/C++, example programs, and all source code to libraries and examples.

Product Warranty

All Signatec 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 customer misuse or abuse of the products or physical damage not reported within 15 days of the time of shipment by Signatec.

Notes:

1. In segmented mode, time from the end of a segment until a trigger will be accepted to begin another segment acquisition.

Signatec reserves the right to make changes in this specification at any time without notice. The information furnished herein is believed to be accurate, however no responsibility is assumed for its use.

5 APPENDIX B - Master-Slave Connections

Pin No.	Connection
1	No connection
3	SCLK+
5	SCLK1-
7	No Connection
9	No Connection
11	No Connection
13	No Connection
15	SYNC TRIGGER

Notes:

SCLK+ and SCLK- are the differential ADC clock signals
SYNC TRIGGER is the external trigger after synchronization with the ADC clock.
Even numbered pins 2 through 16 are grounded.

6 APPENDIX C – Signatec Auxiliary Bus Connections

All odd numbered pins on J1 and J2 are grounded.

SABL Connector		SABH Connector	
Pin No.	Connection	Pin No.	Connection
100	SABL0	100	SABH0
98	SABL1	98	SABH1
96	SABL2	96	SABH2
94	SABL3	94	SABH3
92	SABL4	92	SABH4
90	SABL5	90	SABH5
88	SABL6	88	SABH6
86	SABL7	86	SABH7
84	SABLINTR#6	84	SABHNTR#6
82	SABLWAIT-	82	SABHWAIT-
80	SABLBUSY-	80	SABHBUSY-
78	SABLINTR#7	78	SABHINTR#7
76	SABL8	76	SABHL8
74	SABL9	74	SABH9
72	SABL10	72	SABH10
70	SABL11	70	SABH11
68	SABL12	68	SABH12
66	SABL13	66	SABH13
64	SABL14	64	SABH14
62	SABL15	62	SABH15
60	SABL16	60	SABH16
58	SABL17	58	SABH17
56	SABL18	56	SABH18
54	SABL19	54	SABH19
52	SABL20	52	SABH20
50	SABL21	50	SABH21
48	SABL22	48	SABH22
46	SABL23	46	SABH23
44	SABLSTRB+	44	SABHSTRB+
42	SABLSTRB-	42	SABHSTRB-
40	SABLCOMVAL-	40	SABHCOMVAL-
38	SABLINTR#8	38	SABHINTR#8
36	SABL24	36	SABH24
34	SABL25	34	SABH25
32	SABL26	32	SABH26
30	SABL27	30	SABH27
28	SABL28	28	SABH28
26	SABL29	26	SABH29
24	SABL30	24	SABH30
22	SABL31	22	SABH31
20	SABLCOM1	20	SABHCOM1
18	SABLCOM2	18	SABHCOM2
16	SABLCOM3	16	SABHCOM3
14	SABLCOM4	14	SABHCOM4

12	SABLCOM5	12	SABHCOM5
10	SABLINTR#1	10	SABHINTR#1
8	SABLINTR#2	8	SABHINTR#2
6	SABLINTR#3	6	SABHINTR#3
4	SABLINTR#4	4	SABHINTR#4
2	SABLINTR#5	2	SABHINTR#5

7 Appendix F – PDA16 Library Error Codes

The table below lists all of the currently defined PDA16 library error codes.

The GetErrorTextD16 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_D16_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_D16__FIRST	-512	First PDA16-specific error code value
SIG_D16_NOT_IMPLEMENTED	-512	This operation is not currently implemented
SIG_D16_INVALID_HANDLE	-513	An invalid PDA16 device handle (HPDA16) was specified
SIG_D16_BUFFER_TOO_SMALL	-514	A specified buffer is too small
SIG_D16_INVALID_ARG_1	-515	Argument 1 is invalid
SIG_D16_INVALID_ARG_2	-516	Argument 2 is invalid
SIG_D16_INVALID_ARG_3	-517	Argument 3 is invalid
SIG_D16_INVALID_ARG_4	-518	Argument 4 is invalid
SIG_D16_INVALID_ARG_5	-519	Argument 5 is invalid
SIG_D16_INVALID_ARG_6	-520	Argument 6 is invalid
SIG_D16_INVALID_ARG_7	-521	Argument 7 is invalid
SIG_D16_INVALID_ARG_8	-522	Argument 8 is invalid
SIG_D16_XFER_SIZE_TOO_SMALL	-523	Requested transfer size is too small
SIG_D16_XFER_SIZE_TOO_LARGE	-524	Requested transfer size is too large
SIG_D16_INVALID_DMA_ADDR	-525	Given address is not part of a DMA buffer
SIG_D16_WOULD_OVERRUN_BUFFER	-526	Operation would overrun given buffer
SIG_D16_BUSY	-527	Device is busy; try again later
SIG_D16_INVALID_CHAN_IMP	-528	Incorrect function for board's channel implementation
SIG_D16_XML_MALFORMED	-529	Invalid XML data was encountered
SIG_D16_XML_INVALID	-530	XML data was well formed, but not valid
SIG_D16_XML_GENERIC	-531	Generic XML releated error
SIG_D16_RATE_TOO_FAST	-532	The specified rate is too fast
SIG_D16_RATE_TOO_SLOW	-533	The specified rate is too slow
SIG_D16_RATE_NOT_AVAILABLE	-534	The specified frequency is not available; see

		operator's manual
SIG_D16_UNEXPECTED	-535	An unexpected error occurred; debug builds will have failed assertion
SIG_D16_SOCKET_ERROR	-536	A socket error occurred
SIG_D16_NETWORK_NOT_READY	-537	Network subsystem is not ready for network communication
SIG_D16_SOCKETS_TOO_MANY_TASKS	-538	Limit on number of tasks/processes using sockets has been reached
SIG_D16_SOCKETS_INIT_ERROR	-539	Generic sockets implementation start up failure
SIG_D16_NOT_REMOTE	-540	Not connected to a remote PDA16 device
SIG_D16_TIMED_OUT	-541	Operation timed out
SIG_D16_CONNECTION_REFUSED	-542	Connection refused by service; service may not be running
SIG_D16_INVALID_CLIENT_REQUEST	-543	Received an invalid client request
SIG_D16_INVALID_SERVER_RESPONSE	-544	Received an invalid server response
SIG_D16_REMOTE_CALL_RETURNED_ERROR	-545	Remote service call returned with an error
SIG_D16_UNKNOWN_REMOTE_METHOD	-546	Undefined method invoked on remote server
SIG_D16_SERVER_DISCONNECTED	-547	Server closed the connection
SIG_D16_REMOTE_CALL_NOT_AVAILABLE	-548	Remote call for this operation is not implemented or available
SIG_D16_UNKNOWN_FW_FILE	-549	Unknown firmware file type
SIG_D16_FIRMWARE_UPLOAD_FAILED	-550	Firmware upload failed
SIG_D16_INVALID_FW_FILE	-551	Invalid firmware upload file
SIG_D16_DEST_FILE_OPEN_FAILED	-552	Failed to open destination file
SIG_D16_SOURCE_FILE_OPEN_FAILED	-553	Failed to open source file
SIG_D16_FILE_IO_ERROR	-554	File IO error
SIG_D16_INCOMPATIBLE_FIRMWARE	-555	Firmware is incompatible with PDA16
SIG_D16_UNKNOWN_STRUCT_VER	-556	Unknown structure version specified to library function (X::struct_ver)
SIG_D16_INVALID_REGISTER	-557	An invalid hardware register read/write was attempted
SIG_D16_FIFO_OVERFLOW	-558	An internal FIFO overflowed during acquisition; couldn't keep up with data rate
SIG_D16_DCM_SYNC_FAILED	-559	PDA16 firmware could not synchronize to acquisition clock
SIG_D16_DISK_FULL	-560	Could not write all data; disk is full
SIG_D16_INVALID_OBJECT_HANDLE	-561	An invalid object handle was used
SIG_D16_THREAD_CREATE_FAILURE	-562	Failed to create a thread
SIG_D16_PLL_LOCK_FAILED	-563	Phase lock loop (PLL) failed to lock; clock may be bad
SIG_D16_THREAD_NOT_RESPONDING	-564	Recording thread is not responding
SIG_D16_REC_SESSION_ERROR	-565	A recording sesssion error occurred
SIG_D16_REC_SESSION_CANNOT_ARM	-566	Cannot arm recording sesssion; already armed or stopped
SIG_D16_SNAPSHOTS_NOT_ENABLED	-567	Snapshots not enabled for given recording session
SIG_D16_SNAPSHOT_NOT_AVAILABLE	-568	No data snapshot is available
SIG_D16_SRD_FILE_ERROR	-569	An error occurred while processing .SRD file or stream-embedded SRD data
SIG_D16_NAMED_ITEM_NOT_FOUND	-570	Named item could not be found

SIG_D16_CANNOT_FIND_SRDC_DATA	-571	Could not find Signatec Recorded Data Context info
SIG_D16_NOT_IMPLEMENTED_IN_FIRMWARE	-572	Feature is not implemented in current firmware version; upgrade firmware
SIG_D16_TIMESTAMP_FIFO_OVERFLOW	-573	Timestamp FIFO overflowed during recording
SIG_D16_CANNOT_DETERMINE_FW_REQ	-574	Cannot determine which firmware needs to be uploaded; update software
SIG_D16_REQUIRED_FW_NOT_FOUND	-575	Required firmware not found in firmware update file
SIG_D16_FIRMWARE_IS_UP_TO_DATE	-576	Loaded firmware is up to date with firmware update file
SIG_D16_NO_VIRTUAL_IMPLEMENTATION	-577	Operation not implemented for virtual devices
SIG_D16_NOT_IMPLEMENTED_IN_DRIVER	-578	Feature is not implemented in current driver version
SIG_D16_MS_CFG_CONFLICT	-579	Cannot set specified master/slave configuration; conflicts with hardware configuration
SIG_D16_INVALID_SRDC_HANDLE	-580	Invalid SRDC file handle specified
SIG_D16_INVALIDARG_NULL_POINTER	-581	An invalid pointer was specified
SIG_D16_CFG_EEPROM_ACCESS_DENIED	-582	Access to protected area of configuration EEPROM denied

8 Appendix G – Revision History

Revision 1.0 (Initial release)

Revision 1.1
- Added documentation for D16FILWF_CONVERT_TO_SIGNED bit.
- Added documentation for D16S_FILE_WRITE_PARAMS:: max_file_seg
- Updated PDA16 library error codes in Appendix F
- Added Signatec Recorded Data Context (SRDC) documentation
- Added documentation for Timestamp management routines.

Revision 1.2
- Corrected misspelling in an embedded diagram

Revision 1.3
- Updated digital IO modes in SetDigitalOutputModeD16 function documentation
- Relabeled Digital Output connecter as Digital I/O connector now that it is implemented as bidirectional IO digital IO port.
- Added function documentation: CopyHardwareSettingsD16

Revision 1.4
- Updated PCI requirements
- Updated SAB implementation details
- Updated Appendix A for PDA16 Specs

Revision 1.5
- Added documentation for interfacing with remote PDA16 devices
- Adjusted section outlining for section 3
- Adjusted section outlining for Trigger options
- Added PDA16 Acquisition Data Format section
- Updated PDA16 library error codes in Appendix F
- Removed a few functions that require a future firmware update: WriteSampleRamFastD16, WriteSampleRamD16, WriteSampleRamDualChannelD16
- Updated digital IO mode symbol names. (D16DIGOUT_TTL_HIGH is a misnomer since it's actually a 3.3V output and TTL high-level is 5V. We've replaced the symbol name with D16DIGOUT_3_3V, but the old symbol is still good so we don't break any code.)

Revision 1.6
- Added documentation for functions: SetBoardProcessingEnableD16, SetBoardProcessingParamD16

Revision 1.7
- Replaced a few 'PDAC4000' labels with the correct 'PDA16'.
- Removed incorrect information from the Segmented Trigger Mode section, referring to injected timestamps.
- Renamed 'digital output' to 'digital IO' in top-level documentation.
- Added input gain information to SRDC data documentation. (PDA16 doesn't apply gain, but other Signatec data acquisition devices do.)
- Removed documentation for deprecated PCI Acquisition mode (the non-buffered one).

Revision 1.8

- Added ‘Library Functions That Use Character Strings’ section
- Update Predefined SRDC Items table to include some newer items.
- Updated library error codes in Appendix F
- Documented library functions: SaveSettingsToBufferXmlD16, LoadSettingsFromBufferXmlD16, SaveSettingsToFileXmlD16, LoadSettingsFromFileXmlD16, and SetTimestampCounterModeD16
- Marked older, non-buffered PCI acquisition mode as deprecated.

Revision 1.9

- Removed reference to undefined functions: SetFreeRunModeP16 and EnableSampleCompP16 and updated documentation accordingly.
- Replaced a few incorrect library symbolic constants: (P16MODE* -> D16MODE*)
- Updated the Active Memory Region section contents

Revision 1.10

- WaitForAcquisitionCompleteD16 remarks updated
- WaitForTransferCompleteD16 remarks updated
- Corrected external trigger specifications in Appendix A
- Corrected heading level for Remote PDA16 functions

Revision 1.11

- Corrected maximum pre-trigger samples value; for single channel it’s 4000 and for dual channel it’s 8000.

Revision 1.12

- Corrected minimum alignment for starting address; it is 2048, not 4.
- Corrected references to PDA16 RAM size. The PDA16 has 256 MiS (268435456 samples) of RAM.
- Corrected minimum alignment for sample count; it is 2048 not 16. The firmware may be updated in the future to handle partial burst lengths (i.e. a multiple of 16).
- Added FIFO full check notification to WaitForTransferCompleteD16 and GetPciAcquisitionDataD16 function remarks.

Revision 1.13

- Updated manual cover page for DynamicSignals contact information