DASCard-1000 Series Function Call Driver

USER'S GUIDE

DASCard-1000 Series Function Call Driver User’s Guide

Revision A – June 1995
Part Number: 91880

New Contact Information

Keithley Instruments, Inc.
28775 Aurora Road
Cleveland, OH 44139

Technical Support: 1-888-KEITHLEY
Monday – Friday 8:00 a.m. to 5:00 p.m (EST)
Fax: (440) 248-6168

Visit our website at http://www.keithley.com

The information contained in this manual is believed to be accurate and reliable. However, Keithley Instruments, Inc., assumes no responsibility for its use or for any infringements of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of Keithley Instruments, Inc.

KEITHLEY INSTRUMENTS, INC., SHALL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RELATED TO THE USE OF THIS PRODUCT. THIS PRODUCT IS NOT DESIGNED WITH COMPONENTS OF A LEVEL OF RELIABILITY SUITABLE FOR USE IN LIFE SUPPORT OR CRITICAL APPLICATIONS.

Refer to your Keithley Instruments license agreement for specific warranty and liability information.

MetraByte is a trademark of Keithley Instruments, Inc. All other brand and product names are trademarks or registered trademarks of their respective companies.

© Copyright Keithley Instruments, Inc., 1995.

All rights reserved. Reproduction or adaptation of any part of this documentation beyond that permitted by Section 117 of the 1976 United States Copyright Act without permission of the Copyright owner is unlawful.

Keithley MetraByte Division

Keithley Instruments, Inc.

440 Myles Standish Blvd. Taunton, MA 02780

Telephone: (508) 880-3000 • FAX: (508) 880-0179

Preface

This manual describes how to write application programs for the DASCard-1000 Series using the DASCard-1000 Series Function Call Driver. The DASCard-1000 Series Function Call Driver supports the following DOS-based languages:

- Microsoft® QuickBasic™ (Version 4.5)
- Microsoft Professional Basic (Version 7.0)
- Microsoft C/C++ (Versions 7.0 and 8.0)
- Borland® C/C++ (Versions 3.1 and 4.0)

The DASCard-1000 Series Function Call Driver also supports the following Windows™-based languages:

- Microsoft C/C++ (Versions 7.0 and 8.0)
- Borland C/C++ (Versions 3.1 and 4.0)
- Microsoft Visual Basic® for Windows (Version 3.0)
- Microsoft Visual C++™ (Version 1.5)

The manual is intended for application programmers using a DASCard-1000 Series card in a notebook or desktop computer. It is assumed that users have read the *DASCard-1000 Series User's Guide* to familiarize themselves with the card's features and that they have completed the appropriate hardware and software installation and configuration. It is also assumed that users are experienced in programming in their selected language and that they are familiar with PCMCIA and data acquisition principles.

The *DASCard-1000 Series Function Call Driver User's Guide* is organized as follows:

- Chapter 1 provides an overview of the Function Call Driver, describes how to get started using the Function Call Driver, and describes how to get help.
- Chapter 2 contains the background information needed to use the functions included in the Function Call Driver.
- Chapter 3 contains programming guidelines and language-specific information related to using the Function Call Driver.
- Chapter 4 contains detailed descriptions of the functions, arranged in alphabetical order.
- Appendix A contains a list of the error codes returned by the Function Call Driver.
- Appendix B contains instructions for converting counts to voltage and for converting voltage to counts.

An index completes this manual.

Keep the following conventions in mind as you use this manual:

- References to DASCard-1000 Series cards apply to the DASCard-1001, DASCard-1002, and DASCard-1003 cards. When a feature applies to a particular card, that card's name is used.
- References to BASIC apply to the DOS-based BASIC languages (Microsoft QuickBasic and Microsoft Professional Basic). When a feature applies to a specific language, the complete language name is used. References to Visual Basic for Windows apply to Microsoft Visual Basic for Windows.
- Keyboard keys are shown in bold typeface.

Table of Contents

List of Figures

List of Tables

1

Getting Started

The DASCard-1000 Series Function Call Driver is a library of data acquisition and control functions (referred to as the Function Call Driver or FCD functions). It is part of the following two software packages:

- **DASCard-1000 Series standard software package -** This is the software package that is shipped with DASCard-1000 Series cards; it includes the following:
  - Libraries of FCD functions for Microsoft QuickBasic and Microsoft Professional Basic.
  - Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.
  - Utility programs, running under DOS and Windows, that allow you to allocate resources for, configure, calibrate, and test the features of DASCard-1000 Series cards.
  - Language-specific example programs.
  - Support files for using the DASCard-1000 Series cards with Visual Test Extensions™ (VTX™).
- **ASO-1000 software package -** This is the advanced software option for the DASCard-1000 Series cards. It includes the following:
  - Libraries of FCD functions for Microsoft C/C++ and Borland C/C++.
  - Dynamic Link Libraries (DLLs) of FCD functions for Microsoft C/C++, Borland C/C++, Microsoft Visual Basic for Windows, and Microsoft Visual C++.
  - Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.

- Utility programs, running under DOS and Windows, that allow you to allocate resources for, configure, calibrate, and test the features of the DASCard-1000 Series cards.
- Language-specific example programs.
- Support files for using the DASCard-1000 Series cards with VTX.

Before you use the Function Call Driver, make sure that you have installed the software and your DASCard-1000 Series cards using the procedures described in Chapter 3 of the *DASCard-1000 Series User's Guide*.

If you need help installing or using the DASCard-1000 Series Function Call Driver, call your local sales office or the Keithley MetraByte Applications Engineering Department at:

(508) 880-3000

Monday - Friday, 8:00 A.M. - 6:00 P.M., Eastern Time

An applications engineer will help you diagnose and resolve your problem over the telephone.

Please make sure that you have the following information available before you call:

DASCard-1000 Series card	Model	____________
	Serial #	____________
	Revision code	____________
	Input configuration	Single-ended Differential
	Input range type	Unipolar Bipolar
Computer	Manufacturer	____________
	CPU type	386 486 Pentium
	Clock speed (MHz)	20 25 33 50 66 75 100
	Math coprocessor	Yes No
	Amount of RAM	____________
	Video system	EGA VGA SVGA
	BIOS type	____________
	PCMCIA controller	____________
	Memory manager	____________
Operating system	DOS version	____________
	Windows version	3.0 3.1
	Windows mode	Standard Enhanced
Card and socket services	Type	____________
	Version	____________
Software package	Name	____________
	Serial #	____________
	Version	____________
	Invoice/Order #	____________
Compiler (if applicable)	Language	____________
	Manufacturer	____________
	Version	____________
Accessories	Type/Number	____________
	Type/Number	____________
	Type/Number	____________
	Type/Number	____________
	Type/Number	____________
	Type/Number	____________
	Type/Number	____________
	Type/Number	____________

2

Available Operations

This chapter contains the background information you need to use the FCD functions to perform operations on DASCard-1000 Series cards. The supported operations are listed in Table 2-1.

Table 2-1. Supported Operations

Operation	**Page Reference**
System	page 2-1
Analog input	page 2-5
Digital input and output (I/O)	page 2-24

System Operations

This section describes the miscellaneous and general maintenance operations that apply to DASCard-1000 Series cards and to the DASCard-1000 Series Function Call Driver. It includes information on the following operations:

- Initializing the driver
- Initializing a card
- Retrieving card information
- Retrieving revision levels
- Handling errors

Initializing the Driver

You must initialize the DASCard-1000 Series Function Call Driver and any other Keithley DAS Function Call Drivers you are using in your application program. To initialize the drivers, use the **K_OpenDriver** function. You specify the driver you are using and the configuration file that defines the use of the driver. The driver returns a unique identifier for the driver; this identifier is called the driver handle.

You can specify a maximum of 30 driver handles for all the Keithley MetraByte drivers initialized from all your application programs. If you no longer require a driver and you want to free some memory or if you have used all 30 driver handles, you can use the **K_CloseDriver** function to free a driver handle and close the associated driver.

If the driver handle you free is the last driver handle specified for a Function Call Driver, the driver is shut down. (For Windows-based languages only, the DLLs associated with the Function Call Driver are shut down and unloaded from memory.)

Note: If you are programming in BASIC, **K_OpenDriver** and **K_CloseDriver** are not available. You must use the **DAS1000_DevOpen** function instead. **DAS1000_DevOpen** initializes the DASCard-1000 Series Function Call Driver according to the configuration file you specify. Refer to page 4-6 for more information. In BASIC, closing the DASCard-1000 Series Function Call Driver is not required.

Initializing a Card

The DASCard-1000 Series Function Call Driver supports up to two DASCard-1000 Series cards. You must use the **K_GetDevHandle** function to specify the cards you want to use. The driver returns a unique identifier for each card; this identifier is called the device handle.

Device handles allow you to communicate with more than one Keithley MetraByte DAS card or board. You use the device handle returned by **K_GetDevHandle** in subsequent function calls related to the card or board.

You can specify a maximum of 30 device handles for all the Keithley MetraByte DAS cards or boards accessed from all your application programs. If a card or board is no longer being used and you want to free some memory or if you have used all 30 device handles, you can use the **K_FreeDevHandle** function to free a device handle.

Note: If you are programming in BASIC, **K_GetDevHandle** and **K_FreeDevHandle** are not available. You must use the **DAS1000_GetDevHandle** function instead. Refer to page 4-10 for more information. In BASIC, freeing a device handle is not required.

To reinitialize a Keithley MetraByte DAS card or board during an operation, use the **K_DASDevInit** function. **DAS1000_GetDevHandle**, **K_GetDevHandle**, and **K_DASDevInit** perform the following tasks:

- Abort all operations currently in progress that are associated with the card or board identified by the device handle.
- Verify that the card or board identified by the device handle is the device specified in the configuration file associated with the device.

Retrieving Card Information

The Keithley MetraByte Enabler (KMENABLE.EXE) requests a base address, interrupt level, and memory segment address for each DASCard-1000 Series card from PCMCIA Card Services and then provides information about the assigned resources to your application program. To determine which system resources PCMCIA Card Services assigned, you can use the **DAS1000_GetCardInfo** function in your application program. You specify a DASCard-1000 Series card; the driver returns the socket in which the card is installed, the interrupt level, the base address, the memory segment address, and the card type. Refer to the *DASCard-1000 Series User's Guide* for more information about the Enabler.

Retrieving Revision Levels

If you are using functions from different Keithley DAS Function Call Drivers in the same application program or if you are having problems with your application program, you may want to verify which versions of the Function Call Driver, Keithley DAS Driver Specification, and Keithley DAS Shell are used by your Keithley MetraByte DAS card or board.

The **K_GetVer** function allows you to get both the revision number of the Function Call Driver and the revision number of the Keithley DAS Driver Specification to which the driver conforms.

The **K_GetShellVer** function allows you to get the revision number of the Keithley DAS Shell (the Keithley DAS Shell is a group of functions that are shared by all Keithley MetraByte DAS cards and boards).

Handling Errors

Each FCD function returns a code indicating the status of the function. To ensure that your application program runs successfully, it is recommended that you check the returned code after the execution of each function. If the status code equals 0, the function executed successfully and your program can proceed. If the status code does not equal 0, an error occurred; ensure that your application program takes the appropriate action. Refer to Appendix A for a complete list of error codes.

Each supported language uses a different procedure for error checking; refer to the following pages for more information:

C/C++	page 3-20
Visual Basic for Windows	page 3-33
BASIC	page 3-42

For C-language application programs only, the Function Call Driver provides the **K_GetErrMsg** function, which gets the address of the string corresponding to an error code.

Analog Input Operations

This section describes the following:

- Analog input operation modes available.
- How to allocate and manage memory for analog input operations.
- How to specify the following for an analog input operation:
  - Channels and gains
  - Clock source
  - Buffering mode
  - Trigger source
- How to correct analog input data using calibration factors.

Operation Modes

The operation mode determines which attributes you can specify for an analog input operation and how data is transferred from the DASCard-1000 Series card to computer memory. You can perform analog input operations in single mode, synchronous mode, and interrupt mode, as described in the following sections.

Single Mode

In single mode, the card acquires a single sample from an analog input channel. The driver initiates the conversion; you cannot perform any other operation until the single-mode operation is complete.

Use the **K_ADRead** function to start an analog input operation in single mode. You specify the card you want to use, the analog input channel, the gain at which you want to read the signal, and the variable in which to store the converted data.

Synchronous Mode

In synchronous mode, the card acquires a single sample or multiple samples from one or more analog input channels. A hardware pacer clock initiates conversions. The hardware temporarily stores the acquired data in the 512-word FIFO (first-in, first-out data buffer) on the card, and then transfers the data from the FIFO to a user-defined buffer in computer memory. After the driver transfers the specified number of samples to computer memory, the driver returns control to the application program. You cannot perform any other operation until a synchronous-mode operation is complete.

Use the **K_SyncStart** function to start an analog input operation in synchronous mode.

Interrupt Mode

In interrupt mode, the card acquires a single sample or multiple samples from one or more analog input channels. A hardware clock initiates conversions. Once the analog input operation begins, control returns to your application program. The hardware temporarily stores the acquired data in the FIFO, and then transfers the data from the FIFO to a user-defined buffer in computer memory using an interrupt service routine.

Use the **K_IntStart** function to start an analog input operation in interrupt mode.

You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-17 for more information on buffering modes. Use the **K_IntStop** function to stop a continuous-mode interrupt operation. Use the **K_IntStatus** function to determine the current status of an interrupt operation.

Memory Allocation and Management

Analog input operations require memory buffers in which to store acquired data. For synchronous mode and interrupt mode, you can allocate a single memory buffer; for interrupt mode only, you can allocate multiple buffers (up to a maximum of 150) to increase the number of samples you can acquire. The ways you allocate and manage memory are described in the following sections.

Note: For interrupt-mode operations, the hardware transfers data either when the FIFO is half full (the number of samples is greater than or equal to 256) or when the FIFO has any data (when the number of samples is between 1 and 255). For best performance when using multiple-buffer or continuous-mode operations to acquire data, it is recommended that you allocate a buffer equal to or greater than 256 samples, even if you are not acquiring 256 samples. For single-buffer or single-cycle operations, you can allocate a buffer of any allowable size.

Dimensioning Local Arrays

The simplest way to reserve memory buffers is to dimension arrays within your application program. The advantage of this method is that the arrays are directly accessible to your application program. The limitations of this method are as follows:

- Certain programming languages limit the size of local arrays.
- Local arrays occupy permanent memory areas; these memory areas cannot be freed to make them available to other programs or processes.

Since the DASCard-1000 Series Function Call Driver stores data in 16-bit integers (12 bits of which determine the data), you must dimension all local arrays as integers.

Dynamically Allocating Memory Buffers

The recommended way to reserve memory buffers is to allocate them dynamically outside of your application program's memory area. The advantages of this method are as follows:

- The number of memory buffers and the size of the buffers are limited by the amount of free physical memory available in your computer at run-time.
- Dynamically allocated memory buffers can be freed to make them available to other programs or processes.

The limitation of this method is that for Visual Basic for Windows and BASIC, the data in a dynamically allocated memory buffer is not directly accessible to your program. You must use the **K_MoveBufToArray** function to move the data from the dynamically allocated memory buffer to the program's local array. For Visual Basic for Windows, refer to page 3-27 for more information; for BASIC, refer to page 3-36 for more information.

Use the **K_IntAlloc** function to dynamically allocate a memory buffer for a synchronous-mode or interrupt-mode operation. You specify the operation requiring the buffer and the number of samples to store in the buffer (maximum of 5,000,000 for interrupt mode or 32,767 for synchronous mode). The driver returns the starting address of the buffer and a unique identifier for the buffer; this identifier is called the memory handle. When the buffer is no longer required, you can free the buffer for another use by specifying this memory handle in the **K_IntFree** function.

Notes: For DOS-based languages, the area used for dynamically allocated memory buffers is referred to as the far heap; for Windows-based languages, this area is referred to as the global heap. These heaps are areas of memory left unoccupied as your application program and other programs run.

For DOS-based languages, the **K_IntAlloc** function uses the DOS Int 21h function 48h to dynamically allocate far heap memory. For Windows-based languages, the **K_IntAlloc** function calls the **GlobalAlloc** API function to allocate the desired buffer size from the global heap.

For Windows-based languages, dynamically allocated memory is guaranteed to be fixed and locked in memory.

Assigning the Starting Addresses

After you allocate your buffers or dimension your arrays, you must assign the starting addresses of the arrays or buffers and the number of samples to store in the arrays or buffers. Each supported programming language requires a particular procedure for assigning the starting addresses; refer to the following pages for more information:

C/C++	page 3-13
Visual Basic for Windows	page 3-25
BASIC	page 3-34

If you are using multiple buffers, use the **K_BufListAdd** function to add each buffer to the list of multiple buffers associated with each operation and to assign the starting address of each buffer. Use the **K_BufListReset** function to clear the list of multiple buffers.

Gains and Ranges

Each channel on a DASCard-1001 or DASCard-1002 can measure analog input signals in one of four, software-selectable unipolar or bipolar analog input ranges. Each channel on a DASCard-1003 can measure analog input signals in one unipolar or bipolar analog input range. You specify the input range type (unipolar or bipolar) for the card in the configuration file. Refer to your *DASCard-1000 Series User's Guide* for more information. To set the input range type in your application program, use the **K_SetADMode** function.

Table 2-2 lists the analog input ranges supported by DASCard-1000 Series cards and the gain and gain code associated with each range. Gain codes are used by the FCD functions to represent the gain.

Table 2-2. Analog Input Ranges

Card	Analog Input Range		Gain	Gain Code
	Bipolar	**Unipolar**		
DASCard-1001	±5.0 V	0.0 to +5.0 V	1	0
	±0.5 V	0.0 to +0.5 V	10	1
	±50 mV	0 to +50 mV	100	2
	±5 mV	0 to +5 mV	1000	3
DASCard-1002	±5.0 V	0.0 to +5.0 V	1	0
	±2.5 V	0.0 to +2.5 V	2	1
	±1.25 V	0.0 to +1.25 V	4	2
	±0.625 V	0.0 to +0.625 V	8	3
DASCard-1003	±5.0 V	0.0 to +5.0 V	1	0

For single-mode operations, you specify the gain code in the **K_ADRead** function.

For synchronous-mode and interrupt-mode analog input operations, you specify the gain code in the **K_SetG** or **K_SetStartStopG** function; the function you use depends on how you specify the channels, as described in the following section.

Channels

DASCard-1000 Series cards are software-configurable for either 16 single-ended analog input channels (numbered 0 through 15) or eight differential analog input channels (numbered 0 through 7).

You specify the input configuration (single-ended or differential) in the configuration file. Refer to the *DASCard-1000 Series User’s Guide* for more information. To set the input configuration in your application program, use the **K_SetADConfig** function.

If you require more than the 16 single-ended or eight differential channels, you can use up to 16 EXP-1600 expansion accessories to increase the number of available channels to a maximum of 256.

To use EXP-1600 expansion accessories, the analog input channels on the DASCard-1000 Series card must be configured as single-ended. You assign expansion accessories to consecutive channels on the card, beginning with channel 0. You can also use the remaining channels on the card. Refer to the *DASCard-1000 Series User’s Guide* and to the *EXP-800/1600 User’s Guide* for more information on using expansion accessories.

The maximum supported configuration is 16 EXP-1600 expansion accessories. Table 2-3 lists the software (or logical) channels associated with each expansion accessory.

Table 2-3. Logical Channels

Physical Channel on Card	Software (Logical) Channels	Physical Channel on Card	Software (Logical) Channels
0	0 to 15	8	128 to 143
1	16 to 31	9	144 to 159
2	32 to 47	10	160 to 175
3	48 to 63	11	176 to 191
4	64 to 79	12	192 to 207
5	80 to 95	13	208 to 223
6	96 to 111	14	224 to 239
7	112 to 127	15	240 to 255

Figure 2-1 illustrates the use of three EXP-1600 expansion accessories on a DASCard-1000 Series card configured for single-ended mode.

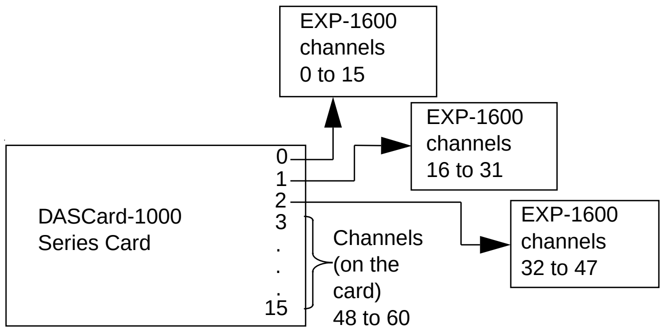

EXP-1600 channels 0 to 15
EXP-1600 channels 16 to 31
EXP-1600 channels 32 to 47
DASCard-1000 Series Card
0
1
2
3
.
.
.
15
Channels (on the card) 48 to 60

Figure 2-1. Analog Input Channels

Note: Because of the overhead required to perform interrupt-mode operations under Windows, it is recommended that you use EXP-1600 expansion accessories in single mode or synchronous mode. The throughput of your DASCard-1000 Series card is reduced when using EXP-1600 expansion accessories.

You can perform an analog input operation on a single channel or on a group of multiple channels. The following sections describe how to specify the channels you are using.

Specifying a Single Channel

You can acquire a single sample or multiple samples from a single analog input channel.

For single-mode analog input operations, you can acquire a single sample from a single analog input channel. Use the **K_ADRead** function to specify the channel and the gain code.

For synchronous-mode and interrupt-mode analog input operations, you can acquire a single sample or multiple samples from a single analog input channel. Use the **K_SetChn** function to specify the channel and the **K_SetG** function to specify the gain code.

Specifying a Group of Consecutive Channels

For synchronous-mode and interrupt-mode analog input operations, you can acquire samples from a group of consecutive channels. Use the **K_SetStartStopChn** function to specify the first and last channels in the group. The channels are sampled in order from first to last; the channels are then sampled again until the required number of samples is read.

For example, assume that you have an EXP-1600 expansion accessory attached to channel 0 on a DASCard-1000 Series card configured for single-ended mode. You specify the start channel as 14, the stop channel as 17, and you want to acquire five samples. Your program reads data first from channels 14 and 15 (on the EXP-1600), then from channels 16 and 17 (physical channels 1 and 2 on the DASCard-1000 Series card), and finally from channel 14 again.

You can specify a start channel that is higher than the stop channel. For example, assume that you are not using any expansion accessories, the card uses a differential input configuration, the start channel is 7, the stop channel is 2, and you want to acquire five samples. Your program reads data first from channel 7 then from channels 0, 1, and 2, and finally from channel 7 again.

Use the **K_SetG** function to specify the gain code for all the channels in the group. (All channels must use the same gain code.) Use the **K_SetStartStopG** function to specify the gain code, the start channel, and the stop channel in a single function call.

Refer to Table 2-2 on page 2-10 for a list of the analog input ranges supported by the DASCard-1000 Series and the gain code associated with each range.

Specifying Channels in a Channel-Gain Queue

For synchronous-mode and interrupt-mode analog input operations, you can acquire samples from channels in a software channel-gain queue. In the channel-gain queue, you specify the channels you want to sample, the order in which you want to sample them, and the gain code for each channel.

Note: Because of the overhead required to perform interrupt-mode operations under Windows, it is recommended that you use channel-gain queues in synchronous mode. The throughput of the DASCard-1000 Series card is reduced when using a channel-gain queue. However, performance is optimized when the channels in the channel-gain queue are sequential and when the gains of all the channels are the same.

You can set up the channels in a channel-gain queue either in consecutive order or in nonconsecutive order. You can also specify the same channel more than once.

The channels are sampled in order from the first channel in the queue to the last channel in the queue; the channels in the queue are then sampled again until the specified number of samples is read.

Refer to Table 2-2 on page 2-10 for a list of the analog input ranges supported by the DASCard-1000 Series and the gain code associated with each range.

The way that you specify the channels and gains in a channel-gain queue depends on the language you are using. Refer to the following pages for more information:

C/C++	page 3-17
Visual Basic for Windows	page 3-31
BASIC	page 3-40

After you create the channel-gain queue in your program, use the **K_SetChnGAry** function to specify the starting address of the channel-gain queue.

Pacer Clocks

For synchronous-mode and interrupt-mode analog input operations, the pacer clock determines the period between conversions. Use the **K_SetClk** function to specify an internal or an external pacer clock. The internal pacer clock is the default pacer clock.

The internal and external pacer clocks are described in the following sections; refer to the *DASCard-1000 Series User's Guide* for more information.

Note: The rate at which the computer can reliably read data from the card depends on a number of factors, including your computer, the operating system/environment, the number of channels you are using, the gains of the channels, and other software issues.

Internal Pacer Clock

The internal pacer clock uses a 16-bit counter on the card. The counter is normally in an idle state. When you start the analog input operation (using **K_SyncStart** or **K_IntStart**), the counter is loaded with its initial value and begins counting down. When the counter counts down to 0, the first conversion is initiated. After the first conversion is initiated, the counter is loaded again and the process repeats.

Use the **K_SetClkRate** function to specify a count value, which represents the number of clock ticks between conversions; each clock tick represents 0.1 µs. For example, if you specify 9,876 clock ticks, the period between conversions is 987.6 µs.

If you are using a DASCard-1003 or if you are using a single channel on a DASCard-1001 or DASCard-1002, you can specify a count value between 71 and 655,350 (7.1 µs to 65.54 ms between conversions). If you are using multiple channels on a DASCard-1001 or DASCard-1002, you can specify a count value between 294 and 655,350 (29.4 µs to 65.54 ms between conversions).

Use the following formula to determine the number of clock ticks to specify:

$$\text{Number of clock ticks} = \frac{10,000,000}{\text{conversion rate}}$$

For example, if you want a conversion rate of 1 ksamples/s, specify 10,000 clock ticks, as shown in the following equation:

$$\frac{10,000,000}{1,000} = 10,000$$

The conversion rate is the rate at which the analog-to-digital converter (ADC) initiates conversions; it does not take into account the number of channels you are using. For example, if you are using five channels and want a conversion rate of 1 ksamples/second per channel, specify 2,000 clock ticks, as shown in the following equation:

$$\left(\frac{10,000,000}{1,000}\right) \div 5 = 2,000$$

The hardware may not be able to convert the analog input channels at the exact rate determined by the number of clock ticks you specify. However, the driver calculates a rate that is as close as possible to the number you specify. To determine the actual number of clock ticks used by the internal pacer clock, use the **K_GetClkRate** function after you start the analog input operation. Refer to page 4-42 for more information.

External Pacer Clock

You connect an external pacer clock to the XCLK/PI0 line of the DASCard-1000 Series card.

When you start an analog input operation (using **K_SyncStart** or **K_IntStart**), conversions are armed. At the next falling edge of the external pacer clock (and at every subsequent falling edge of the external pacer clock), a conversion is initiated.

Note: For the DASCard-1001 and DASCard-1002, the ADC can acquire samples at a maximum of 34 ksamples/s; for the DASCard-1003, the ADC can acquire samples at a maximum of 140 ksamples/s. If you are using an external pacer clock, make sure that the clock initiates conversions at a rate that the ADC can handle.

Buffering Modes

The buffering mode determines how the driver stores the converted data in the buffer. For interrupt-mode analog input operations, you can specify one of the following buffering modes:

- **Single-cycle mode -** In single-cycle mode, after the card converts the specified number of samples and stores them in the buffer, the operation stops automatically. Single-cycle mode is the default buffering mode. To reset the buffering mode to single-cycle, use the **K_ClrContRun** function.
- **Continuous mode -** In continuous mode, the card continuously converts samples and stores them in the buffer until it receives a stop function; any values already stored in the buffer are overwritten. Use the **K_SetContRun** function to specify continuous buffering mode.

Note: Buffering modes are not meaningful for synchronous-mode operations, since only single-cycle mode applies.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For synchronous-mode and interrupt-mode analog input operations, use the **K_SetTrig** function to specify one of the following trigger sources:

- **Internal trigger -** An internal trigger is a software trigger; the trigger event occurs when you start the analog input operation using **K_SyncStart** or **K_IntStart**. Note that a slight delay occurs between the time you start the operation and the time the trigger event occurs. The point at which conversions begin depends on the pacer clock; refer to page 2-15 for more information on the pacer clock. The internal trigger is the default trigger source.
- **External trigger -** An external trigger is either an analog trigger or a digital trigger; when you start the analog input operation using **K_SyncStart** or **K_IntStart**, the application program waits until a trigger event occurs. The point at which conversions begin depends on the pacer clock; refer to page 2-15 for more information on the pacer clock.

Analog and digital triggers are described in the following sections.

Analog Trigger

An analog trigger event occurs when one of the following conditions is met by the analog input signal on a specified analog trigger channel:

- The analog input signal rises above a specified voltage level (positive-edge trigger).
- The analog input signal falls below a specified voltage level (negative-edge trigger).
- The analog input signal is above a specified voltage level (positive-level trigger).
- The analog input signal is below a specified voltage level (negative-level trigger).

Figure 2-2 illustrates these analog trigger conditions, where the specified voltage level is +2.5 V.

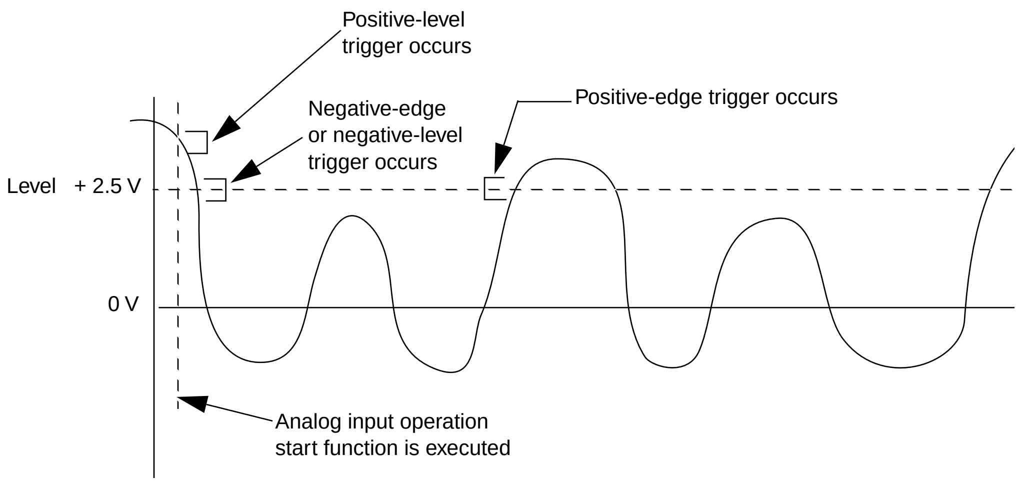


Figure 2-2. Analog Trigger Conditions

Use the **K_SetADTrig** function to specify the following:

- **Analog input channel to use as the trigger channel -** The trigger channel always measures signals at a gain of 1.
- **Voltage level -** You specify the voltage level as a count value. For a bipolar input range type, you specify a count value between –2048 and 2047, where –2048 represents –5 V and 2047 represents 5 V; for a unipolar input range type, you specify a count value between 0 and 4095, where 0 represents 0 V and 4095 represents 5 V. Refer to Appendix B for information on how to convert a voltage value to a count value.
- **Trigger polarity and sensitivity -** The trigger can be a positive-edge, negative-edge, positive-level, or negative-level trigger.

For positive-edge and negative-edge triggers, you can specify a hysteresis value to prevent noise from triggering an operation. Use the **K_SetTrigHyst** function to specify the hysteresis value. The point at which the trigger event occurs is described as follows:

- **Positive-edge trigger -** The analog signal must be below the specified voltage level by at least the amount of the hysteresis value and then rise above the voltage level before the trigger event occurs.
- **Negative-edge trigger -** The analog signal must be above the specified voltage level by at least the amount of the hysteresis value and then fall below the voltage level before the trigger event occurs.

The hysteresis value is an absolute number, which you specify as a count value. For a bipolar input range type, you specify a count value between 0 and 2047, where 0 represents 0 V and 2047 represents 5 V; for a unipolar input range type, you specify a count value between 0 and 4095, where 0 represents 0 V and 4095 represents 5 V. When you add the hysteresis value to the voltage level (for a negative-edge trigger) or subtract the hysteresis value from the voltage level (for a positive-edge trigger), the resulting value must also be between 0 and 2047 for a bipolar input range type or between 0 and 4095 for a unipolar input range type.

For example, assume that you are using a negative-edge trigger on a channel configured for an analog input range of 0 to 5 V. If the voltage level is +4.8 V (3932 counts), you can specify a hysteresis value of 0.1 V (82 counts) because 3932 + 82 is less than 4095, but you cannot specify a hysteresis value of 0.3 V (246 counts) because 3932 + 246 is greater than 4095. Refer to Appendix B for information on how to convert a voltage value to a count value.

In Figure 2-3, the specified voltage level is +4 V and the hysteresis value is 0.1 V. The analog signal must be below +3.9 V and then rise above +4 V before a positive-edge trigger event occurs, the analog signal must be above +4.1 V and then fall below +4 V before a negative-edge trigger event occurs.

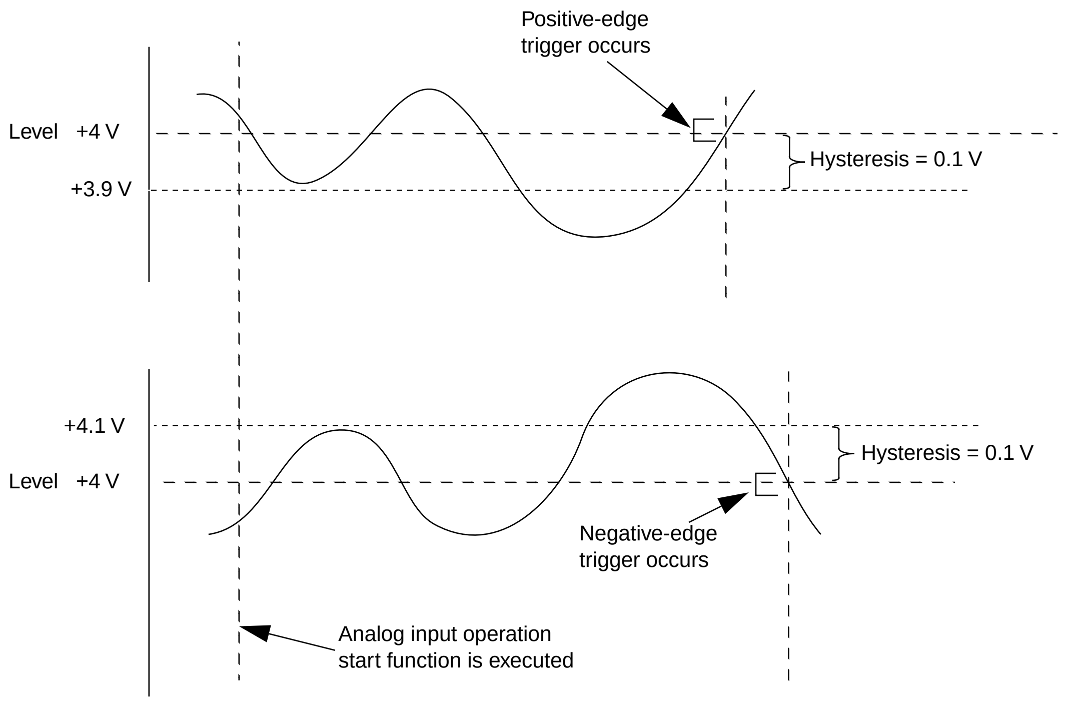

Positive-edge trigger occurs
Level +4 V
+3.9 V
Hysteresis = 0.1 V
+4.1 V
Level +4 V
Hysteresis = 0.1 V
Negative-edge trigger occurs
Analog input operation start function is executed

Figure 2-3. Using a Hysteresis Value

Note: The analog trigger is a software-based trigger. When you start the analog input operation (using **K_IntStart** or **K_SyncStart**), the driver samples the specified trigger channel until the trigger condition is met. If you are performing an operation in interrupt mode, control does not return to your application program until the trigger condition is met. (To terminate the operation if a trigger event does not occur, press **Ctrl** + **Break**.) In addition, a slight time delay occurs between the time the trigger condition is met and the time the driver realizes the trigger condition is met and begins conversions.

Digital Trigger

A digital trigger event occurs when a negative edge is detected on the digital trigger signal connected to the XTRIG/PI1 line of the DASCard-1000 Series card.

Use the **K_SetDITrig** function to specify a digital trigger.

Data Correction

For synchronous-mode and interrupt-mode analog input operations, the data acquired by the card must be corrected to ensure that the data stored in the user-defined buffer is valid. The data is corrected using the calibration factors that are stored in computer memory for each analog input range.

The analog input data can be corrected in one of the following ways:

- **Automatic correction of data -** If speed is not an issue and you are programming under Windows, the driver can automatically correct data as it is acquired. The driver stores the corrected data in the user-defined buffer. By default, automatic data correction is enabled.
- **Correction of data by the driver after the operation is complete -** If you cannot acquire data fast enough using automatic data correction and you are programming under Windows, the driver can store uncorrected data in the user-defined buffer and then correct the data after the operation is complete. Use the **K_SetCalMode** function to disable automatic data correction, and then use the **K_CorrectData** function to correct the data at an appropriate point in your program. The driver overwrites the uncorrected data in the user-defined buffer with the corrected data.
- **Correction of data by the application program after the operation is complete -** If you are programming under Windows and want more control over the correction of your data, the driver can store uncorrected data in the user-defined buffer and the application program can correct the data after the operation is complete. If you are programming under DOS, the driver always stores uncorrected data in the user-defined buffer and the application program must correct the data after the operation is complete.

The application program corrects the data by performing the following steps:

1. If you are programming under Windows, use the **K_SetCalMode** function to disable automatic data correction.
2. Use the **K_GetCalData** function to return the calibration factors for a specified analog input range to a two-element array; the first calibration factor in the array is the gain to apply to the uncorrected data and the second calibration factor in the array is the offset. Note that the gain value returned represents a gain, not a gain code. Refer to page 4-39 for more information about **K_GetCalData**.
3. Since dealing with floating point numbers is language-dependent in DOS, if you are programming under DOS, **K_GetCalData** returns the gain value as a twos complement integer between –127 and +127. Use the following formula to calculate the actual gain to apply to the uncorrected data:

   $$\text{actual gain} = 1.0 + (\text{gain returned by K\_GetCalData} \times 0.001)$$

   If you are programming under Windows, the gain returned by **K_GetCalData** is the actual gain, which is returned as a floating point number; go to step 4.
4. Use one of the following formulas to correct a data value:

   For a bipolar input range type:

   $$\text{corrected data} = (\text{gain} \times \text{uncorrected data}) + \text{offset}$$

   For a unipolar input range type:

   $$\text{corrected data} = (\text{gain} \times (\text{uncorrected data} - 2048)) + \text{offset} + 2048$$

   If you are programming under DOS, the gain used in the formula is the gain value calculated in step 3. If you are programming under Windows, the gain used in the formula is the gain value returned by **K_GetCalData**.

   The offset value is always the offset returned by **K_GetCalData**.

Notes: For automatic correction of data and correction of data by the driver after the operation is complete, you must be programming under Windows. If you are programming under DOS, the application program must correct data after the operation is complete. When programming under DOS, it is not necessary to use **K_SetCalMode** to disable automatic data correction.

If you are programming under Windows and using **K_ADRead** to perform a single-mode analog input operation, data is automatically corrected before it is stored in the variable. If you are programming under DOS and using **K_ADRead**, uncorrected data is stored in the variable and the application program must correct the data.

It is recommended that you periodically update the calibration factors stored in computer memory using the CAL1000.EXE utility. Refer to the *DASCard-1000 Series User's Guide* for more information.

Each supported language uses a different procedure for correcting data; refer to the following pages for more information:

C/C++	page 3-19
Visual Basic for Windows	page 3-32
BASIC	page 3-41

You can convert the corrected count values to voltage, if desired. Refer to Appendix B for more information.

Digital I/O Operations

DASCard-1000 Series cards contain four digital input lines (XCLK/PI0, XTRIG/PI1, PI2, and PI3) and eight digital output lines (PO0 through PO7). If you are not using the digital I/O lines to support an analog input operation, you can use them for general-purpose digital I/O, as described in the following sections.

Digital Input

You can perform a digital input operation in single mode only. Use the **K_DIRead** function to read the value of digital input channel 0, which contains all the digital input lines. You specify the card you want to use, the digital input channel, and the variable in which to store the value. Only bits 0, 1, 2, and 3 of the digital input value are meaningful. Figure 2-4 shows how the digital input bits correspond to the digital input lines.

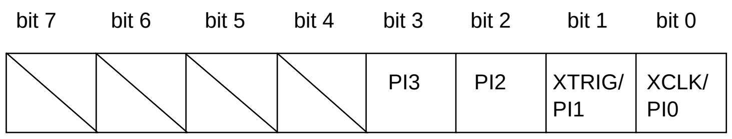


Figure 2-4. Digital Input Bits

A value of 1 in the bit position indicates that the input is high; a value of 0 in the bit position indicates that the input is low. For example, if the value is 5 (00000101), the input at XCLK/PI0 and PI2 is high and the input at XTRIG/PI1 and PI3 is low.

If you are using an external pacer clock, you cannot use XCLK/PI0 for general-purpose digital input operations. If you are using an external digital trigger, you cannot use XTRIG/PI1 for general-purpose digital input operations.

If no signal is connected to a digital input line, the input appears high (value is 1).

Digital Output

You can perform a digital output operation in single mode only. Use the **K_DOWrite** function to write a value to digital output channel 0, which contains all the digital output lines. You specify the card you want to use, the digital output channel, and the digital output value. Figure 2-5 shows how the digital output bits correspond to the digital output lines.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
PO7	PO6	PO5	PO4	PO3	PO2	PO1	PO0

Figure 2-5. Digital Output Bits

A value of 1 in the bit position indicates that the output is high; a value of 0 in the bit position indicates that the output is low. For example, if the value written is 12 (00001100), the output at PO2 and PO3 is forced high, while the output of all other lines is forced low.

If you are using an EXP-1600 expansion accessory for an analog input operation, the driver uses four digital output lines (P0 to P3) to specify the expansion accessory channel that is acquiring data; in this case, you cannot use these digital output lines for general-purpose digital output operations while acquiring analog input data.

3

Programming with the Function Call Driver

This chapter contains an overview of the structure of the Function Call Driver, as well as programming guidelines and language-specific information to assist you when writing application programs with the Function Call Driver.

How the Driver Works

When writing application programs, you can use functions from one or more Keithley MetraByte DAS Function Call Drivers. You initialize each driver using a particular configuration file. If you are using more than one driver or more than one configuration file with a single driver, the driver handle uniquely identifies each driver or each use of the driver.

You can program one or more Keithley MetraByte DAS cards or boards in your application program. You initialize each card or board using a device handle that uniquely identifies it. Each device handle is associated with a particular driver.

The Function Call Driver allows you to perform I/O operations in various operation modes. For single mode, the I/O operation is performed with a single call to a function; the attributes of the I/O operation are specified as arguments to the function. Figure 3-1 illustrates the syntax of the single-mode, analog input operation function **K_ADRead**.

Single-Mode Function		**Attributes of Operation**
K_ADRead (card,	⟷	Card number
channel,	⟷	Analog input channel
gain,	⟷	Gain applied to channel
buffer)	⟷	Buffer for data

Figure 3-1. Single-Mode Function

For other operation modes, such as synchronous and interrupt mode, the driver uses frames to perform the operation. A frame is a data structure whose elements define the attributes of the operation. Each frame is associated with a particular card or board, and therefore, to a particular driver.

Frames help you create structured application programs. You set up the attributes of the operation in advance, using a separate function call for each attribute, and then start the operation at an appropriate point in your program.

Frames are useful for operations that have many defining attributes, since providing a separate argument for each attribute could make a function's argument list unmanageably long. In addition, some attributes, such as the clock source and trigger source, are available only for operations that use frames.

You indicate that you want to perform an operation by getting an available frame for the driver. The driver returns a unique identifier for the frame; this identifier is called the frame handle. You then specify the attributes of the operation by using setup functions to define the elements of the frame associated with the operation. For example, to specify the channel on which to perform an operation, you might use the **K_SetChn** setup function.

You use the frame handle you specified when accessing the frame in all setup functions and other functions related to the operation. This ensures that you are defining the same operation.

When you are ready to perform the operation you have set up, you can start the operation in the appropriate operation mode, referencing the appropriate frame handle. Figure 3-2 illustrates the syntax of the interrupt-mode operation function **K_IntStart**.

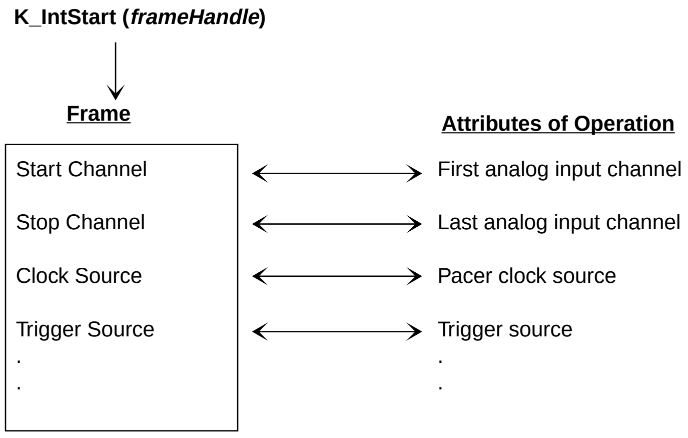

K_IntStart (*frameHandle*)
Frame
Attributes of Operation
Start Channel
Stop Channel
Clock Source
Trigger Source
First analog input channel
Last analog input channel
Pacer clock source
Trigger source

Figure 3-2. Interrupt-Mode Operation

For DASCard-1000 Series cards, synchronous-mode and interrupt-mode analog input operations require frames, called A/D (analog-to-digital) frames. Use the **K_GetADFrame** function to access an available A/D frame.

If you want to perform a synchronous-mode or interrupt-mode analog input operation and all A/D frames have been accessed, you can use the **K_FreeFrame** function to free a frame that is no longer in use. You can then redefine the elements of the frame for the next operation.

When you access a frame, the elements are set to their default values. You can also use the **K_ClearFrame** function to reset all the elements of a frame to their default values.

Table 3-1 lists the elements of an A/D frame for DASCard-1000 Series cards. This table also lists the default value of each element and the setup functions used to define each element.

Table 3-1. A/D Frame Elements

Element	Default Value	Setup Function
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufI K_BufListAdd
Number of Samples	0	K_SetBuf K_SetBufI K_BufListAdd
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[2]
Start Channel	0	K_SetChn K_SetStartStopChn K_SetStartStopG
Stop Channel	0	K_SetStartStopChn K_SetStartStopG
Gain	0 (gain of 1)	K_SetG K_SetStartStopG
Channel-Gain Queue	0 (NULL)	K_SetChnGAry
Clock Source	Internal	K_SetClk
Pacer Clock Rate	0	K_SetClkRate
Trigger Source	Internal	K_SetTrig
Trigger Type	Digital	K_SetADTrig K_SetDITrig
Trigger Channel	0 (for analog trigger)	K_SetADTrig
	0 (for digital trigger)	Not applicable
Trigger Polarity	Positive edge (for analog trigger)	K_SetADTrig
	Positive edge (for digital trigger)	Not applicable[3]

Table 3-1. A/D Frame Elements (cont.)

Element	Default Value	Setup Function
Trigger Level	0	K_SetADTrig
Trigger Hysteresis	0	K_SetTrigHyst
Calibration Mode[4]	Enabled	K_SetCalMode

Notes

[1] You must set this element.

[2] Use this function to reset the value of this particular frame element to its default setting without clearing the frame or getting a new frame. Whenever you clear a frame or get a new frame, this frame element is set to its default value automatically.

[3] Since only negative-edge digital triggers are supported by the hardware, the value of this element is ignored.

[4] This element is valid only when programming under Windows.

Note: The DASCard-1000 Series Function Call Driver provides many other functions that are not related to controlling frames, defining the elements of frames, or reading the values of frame elements. These functions include single-mode operation functions, initialization functions, memory management functions, data correction functions, and miscellaneous functions.

For information about using the FCD functions in your application program, refer to the following sections of this chapter. For detailed information about the syntax of FCD functions, refer to Chapter 4.

Programming Overview

To write an application program using the DASCard-1000 Series Function Call Driver, perform the following steps:

1. Define the application's requirements. Refer to Chapter 2 for a description of the operations supported by the Function Call Driver and the functions that you can use to define each operation.
2. Write your application program. Refer to the following for additional information:
   - Preliminary Tasks, the next section, which describes the programming tasks that are common to all application programs.
   - Analog Input Programming Tasks on page 3-7 and Digital I/O Programming Tasks on page 3-12, which describe operation-specific programming tasks and the sequence in which these tasks must be performed.
   - Chapter 4, which contains detailed descriptions of the FCD functions.
   - The example programs in the DASCard-1000 Series standard software package and the ASO-1000 software package. The FILES.TXT file in the installation directory lists and describes the example programs.
3. Compile and link the program. Refer to the following for information on compile and link statements and other language-specific considerations:
   - C/C++ Programming Information on page 3-13.
   - Visual Basic for Windows Programming Information on page 3-25.
   - BASIC Programming Information on page 3-34.
   - The EXAMPLES.TXT file located in the installation directory.

Preliminary Tasks

For every Function Call Driver application program, you must perform the following preliminary tasks:

1. Include the function and variable type definition file for your language. Depending on the specific language you are using, this file is included in the DASCard-1000 Series standard software package or the ASO-1000 software package.
2. Declare and initialize program variables.
3. Use a driver initialization function (**K_OpenDriver** or **DAS1000_DevOpen**) to initialize the driver.
4. Use a card initialization function (**K_GetDevHandle** or **DAS1000_GetDevHandle**) to specify DASCard-1000 Series card you want to use and to initialize the card. If you are using two cards, use the initialization function twice.

After completing the preliminary tasks, perform the appropriate operation-specific programming tasks. The operation-specific tasks for analog input and digital I/O operations are described in the following sections.

Analog Input Programming Tasks

The following sections describe the operation-specific programming tasks required to perform single-mode, synchronous-mode, and interrupt-mode analog input operations.

Single-Mode Operations

For a single-mode analog input operation, perform the following tasks:

1. Declare the buffer or variable in which to store the single analog input value.
2. Use the **K_ADRead** function to read the single analog input value; specify the attributes of the operation as arguments to the function.

Synchronous-Mode Operations

For a synchronous-mode analog input operation, perform the following tasks:

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Allocate the buffer or dimension the array in which to store the acquired data. Use the **K_IntAlloc** function if you want to allocate the buffer dynamically outside your program's memory area.
3. *If you want to use a channel-gain queue to specify the channels acquiring data*, define and assign the appropriate values to the queue and note the starting address. Refer to page 2-14 for more information about channel-gain queues.
4. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-2 on page 3-8.

Note: When you access a new A/D frame, the frame elements contain default values. If the default value of a particular element is suitable for your operation, you do not have to use the setup function associated with that element. Refer to Table 3-1 on page 3-4 for a list of the default values of A/D frame elements.

Table 3-2. Setup Functions for Synchronous-Mode Analog Input Operations

Attribute	Setup Functions
Buffer	K_SetBuf K_SetBufI
Number of Samples	K_SetBuf K_SetBufI
Start Channel	K_SetChn K_SetStartStopChn K_SetStartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG

Table 3-2. Setup Functions for Synchronous-Mode Analog Input Operations (cont.)

Attribute	Setup Functions
Gain	K_SetG K_SetStartStopG
Channel-Gain Queue	K_SetChnGAry
Clock Source	K_SetClk
Pacer Clock Rate	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig
Trigger Polarity	K_SetADTrig K_SetDITrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst
Calibration Mode	K_SetCalMode

Refer to Chapter 2 for background information about the setup functions; refer to Chapter 4 for detailed descriptions of the setup functions.

5. Use the **K_SyncStart** function to start the synchronous-mode operation.
6. *If you are programming in Visual Basic for Windows or BASIC and you used* ***K_IntAlloc*** *to allocate your buffer*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to the program’s local array.
7. *If you are programming in C/C++ for DOS or BASIC*, use the **K_GetCalData** function to get the calibration factors, and then correct the data. Refer to page 2-22 for more information.

8. *If you used **K_IntAlloc** to allocate your buffer*, use the **K_IntFree** function to deallocate the buffer.
9. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt-Mode Operations

For an interrupt-mode analog input operation, perform the following tasks:

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Allocate the buffers or dimension the arrays in which to store the acquired data. Use the **K_IntAlloc** function if you want to allocate buffers dynamically outside your program's memory area.
3. *If you want to use a channel-gain queue to specify the channels acquiring data*, define and assign the appropriate values to the queue and note the starting address. Refer to page 2-14 for more information about channel-gain queues.
4. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-3.

Note: When you access a new A/D frame, the frame elements contain default values. If the default value of a particular element is suitable for your operation, you do not have to use the setup function associated with that element. Refer to Table 3-1 on page 3-4 for a list of the default values of A/D frame elements.

Table 3-3. Setup Functions for Interrupt-Mode Analog Input Operations

Attribute	Setup Functions
Buffer	K_SetBuf K_SetBufI K_BufListAdd
Number of Samples	K_SetBuf K_SetBufI K_BufListAdd
Buffering Mode	K_SetContRun K_ClrContRun
Start Channel	K_SetChn K_SetStartStopChn K_SetStartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStopG
Channel-Gain Queue	K_SetChnGAry
Clock Source	K_SetClk
Pacer Clock Rate	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig
Trigger Polarity	K_SetADTrig K_SetDITrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst
Calibration Mode	K_SetCalMode

Refer to Chapter 2 for background information about the setup functions; refer to Chapter 4 for detailed descriptions of the setup functions.

5. Use the **K_IntStart** function to start the interrupt-mode operation.
6. Use the **K_IntStatus** function to monitor the status of the interrupt-mode operation.
7. *If you specified continuous buffering mode*, use the **K_IntStop** function to stop the interrupt-mode operation when the appropriate number of samples has been acquired.
8. *If you are programming in Visual Basic for Windows or BASIC and you used **K_IntAlloc** to allocate your buffer*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to the program's local array.
9. *If you are programming in C/C++ for DOS or BASIC*, use the **K_GetCalData** function to get the calibration factors, and then correct the data. Refer to page 2-22 for more information.
10. *If you used **K_IntAlloc** to allocate your buffer*, use the **K_IntFree** function to deallocate the buffer.
11. *If you used **K_BufListAdd** to specify a list of multiple buffers*, use the **K_BufListReset** function to clear the list.
12. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Digital I/O Programming Tasks

For a single-mode digital I/O operation, perform the following tasks:

1. Declare the buffer or variable in which to store the single digital I/O value.
2. Use one of the following digital I/O single-mode operation functions, specifying the attributes of the operation as arguments to the function:

Function	Purpose
K_DIRead	Reads a single digital input value.
K_DOWrite	Writes a single digital output value.

C/C++ Programming Information

The following sections contain information you need to allocate and assign memory buffers, to create channel-gain queues, and to handle errors in C or C++, as well as other language-specific information for Microsoft C/C++ and Borland C/C++.

Notes: Make sure that you use proper typecasting to prevent C/C++ type-mismatch warnings.

Make sure that linker options are set so that case-sensitivity is disabled.

Dynamically Allocating and Assigning Memory Buffers

This section provides code fragments that describe how to allocate and assign dynamically allocated memory buffers when programming in C or C++. Refer to the example programs on disk for more information.

Note: If you are using large or multiple memory buffers and you are programming in Windows Enhanced mode, you may be limited in the amount of memory you can allocate. It is recommended that you install the Keithley Memory Manager before you begin programming to ensure that you can allocate a large enough buffer or buffers. Refer to the *DASCard-1000 Series User's Guide* for more information about the Keithley Memory Manager.

Allocating a Single Memory Buffer

You can use a single, dynamically allocated memory buffer for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to use **K_IntAlloc** to allocate a buffer of size Samples for the frame defined by hFrame and how to use **K_SetBuf** to assign the starting address of the buffer.

```
. . .
void far *AcqBuf;       //Declare pointer to buffer
WORD hMem;              //Declare word for memory handle
. . .
wDasErr = K_IntAlloc (hFrame, Samples, &AcqBuf, &hMem);
wDasErr = K_SetBuf (hFrame, AcqBuf, Samples);
. . .
```

The following code illustrates how to use **K_IntFree** to later free the allocated buffer, using the memory handle stored by **K_IntAlloc**.

```
. . .
wDasErr = K_IntFree (hMem);
. . .
```

Allocating Multiple Memory Buffers

You can use multiple, dynamically allocated memory buffers for interrupt-mode analog input operations.

The following code fragment illustrates how to use **K_IntAlloc** to allocate five buffers of size Samples each for the frame defined by hADFrame and how to use **K_BufListAdd** to assign the starting addresses of the five buffers.

```
. . .
void far *AcqBuf[5];       //Declare 5 pointers to 5 buffers
WORD hMem[5];              //Declare 5 words for 5 memory handles
. . .
for (i = 0; i < 5; i++) {
wDasErr = K_IntAlloc (hADFrame, Samples, &AcqBuf[i],&hMem[i]);
wDasErr = K_BufListAdd (hADFrame, AcqBuf[i], Samples);
}
. . .
```

The following code illustrates how to use **K_IntFree** to later free the allocated buffers, using the memory handles stored by **K_IntAlloc**. If you free the allocated buffers, you must also use **K_BufListReset** to reset the buffer list associated with the frame.

```
. . .
for (i = 0; i < 5; i++) {
   wDasErr = K_IntFree (hMem[i]);
}
wDasErr = K_BufListReset (hADFrame);
. . .
```

Accessing the Data

You access the data stored in dynamically allocated buffers through C/C++ pointer indirection. For example, assume that you want to display the first 10 samples of the second buffer in the multiple-buffer operation described in the previous section (AcqBuf [1]). The following code fragment illustrates how to access and display the data.

```
int huge *pData;           //Declare a pointer called pData
. . .
pData = (int huge*) AcqBuf[1]; //Assign pData to 2nd buffer
for (i = 0; i < 10; i++)
   printf ("Sample #%d %X", i, *(pData+i));
. . .
```

Note: Declaring pData as a huge pointer allows the program to directly access all data in the buffer, regardless of the buffer size.

Dimensioning and Assigning Local Arrays

This section provides code fragments that describe how to dimension and assign local arrays when programming in C or C++. Refer to the example programs on disk for more information.

Dimensioning a Single Array

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to dimension an array of 10,000 samples for the frame defined by hFrame and how to use **K_SetBuf** to assign the starting address of the array.

```
. . .
int Data[10000];  //Dimension array of 10,000 samples
. . .
wDasErr = K_SetBuf (hFrame, Data, 10000);
. . .
```

Dimensioning Multiple Arrays

You can use multiple, local arrays for interrupt-mode analog input operations.

The following code fragment illustrates how to allocate two arrays of 32,000 samples each for the frame defined by hADFrame and how to use **K_BufListAdd** to assign the starting addresses of the arrays.

```
. . .
int Data1[32000];    //Allocate Array #1 of 32,000 samples
int Data2[32000];    //Allocate Array #2 of 32,000 samples
. . .
wDasErr = K_BufListAdd (hADFrame, Data1, 32000);
wDasErr = K_BufListAdd (hADFrame, Data2, 32000);
```

Creating a Channel-Gain Queue

The DASDECL.H and DASDECL.HPP files define a special data type (GainChanTable) that you can use to declare your channel-gain queue. GainChanTable is defined as follows:

```
typedef struct GainChanTable
{
   WORD num_of_codes;
   struct{
      char Chan;
      char Gain;
   } GainChanAry[256];
} GainChanTable;
```

The following example illustrates how to create a channel-gain queue called MyChanGainQueue for a DASCard-1002 card by declaring and initializing a variable of type GainChanTable:

```
GainChanTable MyChanGainQueue =
   {8,         //Number of entries
   0, 0,       //Channel 0, gain of 1
   1, 1,       //Channel 1, gain of 2
   2, 2,       //Channel 2, gain of 4
   3, 3,       //Channel 3, gain of 8
   3, 0,       //Channel 3, gain of 1
   2, 1,       //Channel 2, gain of 2
   1, 2,       //Channel 1, gain of 4
   0, 3};      //Channel 0, gain of 8
```

After you create MyChanGainQueue, you must assign the starting address of MyChanGainQueue to the frame defined by hFrame, as follows:

```
wDasErr = K_SetChnGAry (hFrame, &MyChanGainQueue);
```

When you start the next analog input operation (using **K_SyncStart** or **K_IntStart**), channel 0 is sampled at a gain of 1, channel 1 is sampled at a gain of 2, channel 2 is sampled at a gain of 4, and so on.

Correcting Data (for DOS)

If you are using C/C++ for DOS, the application program must correct the data after the operation is complete. The following code fragment illustrates how to correct the data and how to convert the corrected data to volts.

```
void far *pIntBuf;      /* Pointer to allocated buffer */
WORD hMem;              /* Allocated memory handle */
int far *pData;         /* Temporary pointer to data */
int nCounts;

int nCalData[2];        /* Contains calibration values */
float fGainVal;
int nOffsetVal;
. . .
/* Data pointed to by pIntBuf */
wDasErr = K_IntAlloc (hAD, dwSamples, &pIntBuf, &hMem);
. . .
wDasErr = K_IntStart (hAD);
. . .
/* Gets gain and offset values */
wDasErr = K_GetCalData (hAD, 0, 0, NULL, nCalData);
. . .
/* Determines the actual gain to apply to the data */
fGainVal = (float) (1.0 + (float) nCalData[0] * 0.001);
nOffsetVal = nCalData[1];
. . .
/* Corrects the data and converts counts to volts; assumes that */
/* you are using a bipolar +/-5 V analog input range */
pData = (int far *) pIntBuf;  /* Pointer to 1st sample in buffer */
nCounts= (int) ((float)*pData * fGainVal) + nOffsetVal; /*Corrects*/
fVolts = (float) nCounts * (10.0 / 4096.0);             /*Converts*/
printf (“Sample = %x counts, %.3f volts\n”, *pData, fVolts);
```

Correcting Data (for Windows)

If you are using C/C++ for Windows and have disabled automatic data correction, the application program must correct the data after the operation is complete. The following code fragment illustrates how to correct the data and how to convert the corrected data to volts.

```
void far *pIntBuf;      /* Pointer to allocated buffer */
WORD hMem;              /* Allocated memory handle */
int far *pData;         /* Temporary pointer to data */
int nCounts;

float fCalData[2];      /* Contains calibration values */
float fGainVal;
float fOffsetVal;
. . .
/* Data pointed to by pIntBuf */
wDasErr = K_IntAlloc (hAD, dwSamples, &pIntBuf, &hMem);
. . .
wDasErr = K_SetCalMode (hAD, 0); /* Disables correction by driver */
. . .
wDasErr = K_IntStart (hAD);
. . .
/* Gets gain and offset values */
wDasErr = K_GetCalData (hAD, 0, 0, NULL, fCalData);
fGainVal = fCalData[0];
fOffsetVal = fCalData[1];
. . .
/* Corrects the data and converts counts to volts; assumes that */
/* you are using a bipolar +/-5 V analog input range */
pData = (int far *) pIntBuf;  /* Pointer to 1st sample in buffer */
nCounts= (int) ((float)*pData * fGainVal) + fOffsetVal; /*Corrects*/
fVolts = (float) nCounts * (10.0 / 4096.0);             /*Converts*/
printf (“Sample = %x counts, %.3f volts\n”, *pData, fVolts);
```

Handling Errors

It is recommended that you always check the returned value (wDasErr in the previous examples) for possible errors. The following code fragment illustrates how to check the returned value of the **K_GetDevHandle** function.

```
. . .
if ((wDasErr = K_GetDevHandle (hDrv, BoardNum, &hDev)) ! = 0)
  {
  printf (“Error %X during K_GetDevHandle”, wDasErr);
  exit (1);
  }
```

The following code fragment illustrates how to use the **K_GetErrMsg** function to access the string corresponding to an error code.

```
. . .
if ((wDasErr = K_SetChn (hAD, 2) ! = 0)
  {
  Error = K_GetErrMsg (hDev, wDasErr, &pMessage);
  printf (“%s”, pMessage);
  exit (1);
  }
```

Programming in Microsoft C/C++ (for DOS)

To program in Microsoft C/C++ (for DOS), you need the following files; these files are provided in the ASO-1000 software package.

File	Description
DAS1000.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C (.c programs)
DAS1000.H	Include file when compiling in C (.c programs)
DASDECL.HPP	Include file when compiling in C++ (.cpp programs)
DAS1000.HPP	Include file when compiling in C++ (.cpp programs)
USE1000.OBJ	Linkable object

To create an executable file in Microsoft C/C++ (for DOS), use the following compile and link statements. Note that *filename* indicates the name of your application program.

Type of Compile	**Compile and Link Statements**
C	CL /c *filename*.c LINK *filename*+use1000.obj,,,das1000+dasrface;
C++	CL /c *filename*.cpp LINK *filename*+use1000.obj,,,das1000+dasrface;

Programming in Microsoft C/C++ (for Windows)

To program in Microsoft C/C++ (for Windows), including Microsoft Visual C++, you need the following files; these files are provided in the ASO-1000 software package.

File	**Description**
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DAS1000.DLL	Dynamic Link Library
DASDECL.H	Include file when compiling in C (.c programs)
DAS1000.H	Include file when compiling in C (.c programs)
DASDECL.HPP	Include file when compiling in C++ (.cpp programs)
DAS1000.HPP	Include file when compiling in C++ (.cpp programs)
DASIMP.LIB	DAS Shell Imports
D1000IMP.LIB	DASCard-1000 Imports

To create an executable file in Microsoft C/C++ (for Windows), use the following compile and link statements. Note that *filename* indicates the name of your application program.

Type of Compile	**Compile and Link Statements**
C	CL /c *filename*.c LINK *filename*,,,d1000imp+dasimp,*filename*.def; RC –r *filename*.rc RC –30 *filename*.res
C++	CL /c *filename*.cpp LINK *filename*,,,d1000imp+dasimp,*filename*.def; RC –r *filename*.rc RC –30 *filename*.res

To create an executable file in the Microsoft C/C++ (for Windows) environment, perform the following steps:

1. Create a project file by choosing New from the Project menu.
2. Add all necessary files to the project make file by choosing Edit from the Project menu. Make sure that you include *filename*.c (or *filename*.cpp), *filename*.rc, *filename*.def, DASIMP.LIB, and D1000IMP.LIB, where *filename* indicates the name of your application program.
3. From the Project menu, choose Rebuild All FILENAME.EXE to create a stand-alone executable file (.EXE) that you can execute from within Windows.

Programming in Borland C/C++ (for DOS)

To program in Borland C/C++ (for DOS), you need the following files; these files are provided in the ASO-1000 software package.

File	Description
DAS1000.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C (.c programs)
DAS1000.H	Include file when compiling in C (.c programs)
DASDECL.HPP	Include file when compiling in C++ (.cpp programs)
DAS1000.HPP	Include file when compiling in C++ (.cpp programs)
USE1000.OBJ	Linkable object

To create an executable file in Borland C/C++ (for DOS), use the following compile and link statements. Note that *filename* indicates the name of your application program.

Type of Compile	Compile and Link Statements[1]
C	BCC -ml *filename*.c use1000.obj das1000.lib dasrface.lib
C++	BCC -ml *filename*.cpp use1000.obj das1000.lib dasrface.lib

Notes

[1] These statements assume a large memory model; however, any memory model is acceptable.

Programming in Borland C/C++ (for Windows)

To program in Borland C/C++ (for Windows), you need the following files; these files are provided in the ASO-1000 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DAS1000.DLL	Dynamic Link Library
DASDECL.H	Include file when compiling in C (.c programs)
DAS1000.H	Include file when compiling in C (.c programs)
DASDECL.HPP	Include file when compiling in C++ (.cpp programs)
DAS1000.HPP	Include file when compiling in C++ (.cpp programs)
DASIMP.LIB	DAS Shell Imports
D1000IMP.LIB	DASCard-1000 Imports

To create an executable file in Borland C/C++ (for Windows), use the following compile and link statements. Note that *filename* indicates the name of your application program.

Type of Compile	Compile and Link Statements
C	BCC -c *filename*.c TLINK *filename*,,,d1000imp+dasimp, *filename*.def; BRC -r *filename*.rc BRC -30 *filename*.res
C++	BCC -c *filename*.cpp TLINK *filename*,,,d1000imp+dasimp, *filename*.def; BRC -r *filename*.rc BRC -30 *filename*.res

To create an executable file in the Borland C/C++ (for Windows) environment, perform the following steps:

1. Create a project file by choosing New from the Project menu.
2. Inside the Project window, select the project name and click on the right mouse button.
3. Select the Add node option and add all necessary files to the project make file. Make sure that you include *filename*.c (or *filename*.cpp), *filename*.rc, *filename*.def, DASIMP.LIB, and D1000IMP.LIB, where *filename* indicates the name of your application program.
4. From the Options menu, select Project.
5. From the Project Options dialog box, select Linker\General and make sure that you turn OFF both the Case sensitive link and Case sensitive exports and imports options.
6. From the Project menu, choose Build All to create a stand-alone executable file (.EXE) that you can execute from within Windows.

Visual Basic for Windows Programming Information

The following sections contain information you need to allocate and assign memory buffers, to create channel-gain queues, and to handle errors in Microsoft Visual Basic for Windows, as well as other language-specific information for Microsoft Visual Basic for Windows.

Dynamically Allocating and Assigning Memory Buffers

This section provides code fragments that describe how to allocate and assign dynamically allocated memory buffers when programming in Microsoft Visual Basic for Windows. Refer to the example programs on disk for more information.

Note: If you are using large or multiple memory buffers and you are programming in Windows Enhanced mode, you may be limited in the amount of memory you can allocate. It is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate a large enough buffer. Refer to your *DASCard-1000 Series User's Guide* for more information about the Keithley Memory Manager.

Allocating a Single Memory Buffer

You can use a single, dynamically allocated memory buffer for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to use **K_IntAlloc** to allocate a buffer of size Samples for the frame defined by hFrame and how to use **K_SetBuf** to assign the starting address of the buffer.

```
. . .
Global AcqBuf As Long   ’ Declare pointer to buffer
Global hMem As Integer  ’ Declare integer for memory handle
. . .
wDasErr = K_IntAlloc (hFrame, Samples, AcqBuf, hMem)
wDasErr = K_SetBuf (hFrame, AcqBuf, Samples)
. . .
```

The following code illustrates how to use **K_IntFree** to later free the allocated buffer, using the memory handle stored by **K_IntAlloc**.

```
. . .
wDasErr = K_IntFree (hMem)
. . .
```

Allocating Multiple Memory Buffers

You can use multiple, dynamically allocated memory buffers for interrupt-mode analog input operations.

The following code fragment illustrates how to use **K_IntAlloc** to allocate five buffers of size Samples each for the frame defined by hADFrame and how to use **K_BufListAdd** to assign the starting addresses of the five buffers.

```
. . .
Global AcqBuf(4) As Long         ’ Declare 5 pointers to 5 buffers
Global hMem(4) As Integer        ’ Declare 5 memory handles
. . .
for i% = 0 to 4
   wDasErr = K_IntAlloc (hFrame, Samples, AcqBuf(i%), hMem(i%))
   wDasErr = K_BufListAdd (hFrame, AcqBuf(i%), Samples)
next i%
. . .
```

The following code illustrates how to use **K_IntFree** to later free the allocated buffers, using the memory handles stored by **K_IntAlloc**; if you free the allocated buffers, you must also use **K_BufListReset** to reset the buffer list associated with the frame.

```
. . .
for i% = 0 to 4
   wDasErr = K_IntFree (hMem(i%))
next i%
wDasErr = K_BufListReset (hADFrame)
. . .
```

Accessing the Data from Buffers with Fewer than 64K Bytes

In Microsoft Visual Basic for Windows, you cannot directly access analog input samples stored in dynamically allocated memory buffers. You must use **K_MoveBufToArray** to move a subset (up to 32,766 samples) of the data into a local array as required. The following code fragment illustrates how to move the first 100 samples of the second buffer in the multiple-buffer operation described in the previous section (AcqBuf(1)) to a local array.

```
. . .
Dim Buffer(1000) As Integer   ’ Declare local memory buffer
. . .
wDasErr = K_MoveBufToArray (Buffer(0), AcqBuf(1), 100)
. . .
```

Accessing the Data from Buffers with More than 64K Bytes

When Windows is running, the CPU operates in 16-bit protected mode. Memory is addressed using a 32-bit selector:offset pair. The selector is the CPU's handle to a 64K byte memory page; it is a code whose value is significant only to the CPU. No mathematical relationship exists between a selector and the memory location it is associated with. In general, even consecutively allocated selectors have no relationship to each other.

When a memory buffer of more than 64K bytes (32K values) is used, multiple selectors are required. Under Windows, **K_IntAlloc** uses a "tiled" method to allocate memory whereby a mathematical relationship does exist among the selectors. Specifically, if you allocate a buffer of more than 64K bytes, each selector that is allocated has an arithmetic value that is eight greater than the previous one. The format of the address is a 32-bit value whose high word is the 16-bit selector value and low word is the 16-bit offset value. When the offset reaches 64K bytes, the next consecutive memory address location can be accessed by adding eight to the selector and resetting the offset to zero; to do this, add &h80000 to the buffer starting address.

Table 3-4 illustrates the mapping of consecutive memory locations in protected-mode "tiled" memory, where *xxxxxxxx* indicates the address calculated by the CPU memory mapping mechanism.

Table 3-4. Protected-Mode Memory Architecture

Selector:Offset	**32-Bit Linear Address**
. . . . :	. . . . .
32E6:FFFE	*xxxxxxxx*
32E6:FFFF	*xxxxxxxx* + 1
32EE:0000	*xxxxxxxx* + 2
32EE:0001	*xxxxxxxx* + 3
. . . . :	. . . . .

The following code fragment illustrates moving 1,000 values from a memory buffer (AcqBuf) allocated with 50,000 values to the program's local array (Array), starting at sample number 40,000. First, start with the buffer address passed in **K_SetBuf**. Then, determine how deep (in 64K byte pages) into the buffer the desired sample number (40,000) is located and add &h80000 to the buffer address for each 64K byte page. Finally, add any additional offset after the 64K byte pages to the buffer address.

```
Dim AcqBuf As Long
Dim NumSamps As Long

Dim Array (1000) As Integer
NumSamps = 50000
wDasErr = K_IntAlloc (hFrame, NumSamps, AcqBuf, hMem)
. . .
'Acquisition routine
. . .
DesiredSamp = 40000
DesiredByte = DesiredSamp * 2       'Number of bytes into buffer
AddSelector = DesiredByte / &h10000 'Number of 64K pages into buffer
RemainingOffset = DesiredByte Mod &h10000    'Additional offset

DesiredBuffLoc = AcqBuf + (AddSelector * &h80000) + RemainingOffset

wDasErr = K_MoveBufToArray (Array(0), DesiredBuffLoc, 1000)
```

To move more than 32,766 values from the memory buffer to the program's local array, the program must call **K_MoveBufToArray** more than once. For example, assume that pBuf is a pointer to a dynamically allocated buffer that contains 65,536 values. The following code fragment illustrates how to move 65,536 values from the dynamically allocated buffer to the program's local array.

```
...
Dim Data [3, 16384] As Integer
...
wDasErr = K_MoveBufToArray (Data(0,0), pBuf, 16384)

'Same selector, add 32,768 bytes to offset: add &h8000
wDasErr = K_MoveBufToArray (Data(1,0), pBuf + &h8000, 16384)

'Add 8 to selector, offset = 0: add &h80000
wDasErr = K_MoveBufToArray (Data(2,0), pBuf + &h80000, 16384)
'Add 8 to selector, add 32,768 bytes to offset: add &h88000
wDasErr = K_MoveBufToArray (Data(3,0), pBuf + &h88000, 16384)
```

Dimensioning and Assigning Local Arrays

This section provides code fragments that describe how to dimension and assign local arrays when programming in Microsoft Visual Basic for Windows. Refer to the example programs on disk for more information.

Dimensioning a Single Array

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to dimension an array of 10,000 samples for the frame defined by hFrame and how to use **K_SetBufI** to assign the starting address of the array.

```
   . . .
   Global Data(9999) As Integer     ’ Allocate array
   . . .
   wDasErr = K_SetBufI (hFrame, Data(0), 10000)
   . . .
```

Dimensioning Multiple Arrays

You can use multiple, local arrays for interrupt-mode analog input operations.

The following code fragment illustrates how to dimension two arrays of 32,000 samples each for the frame defined by hADFrame and how to use **K_BufListAdd** to assign the starting addresses of the arrays.

```
. . .
Global Data1(31999) As Integer   ’ Allocate Array #1
Global Data2(31999) As Integer   ’ Allocate Array #2
. . .
wDasErr = K_BufListAdd (hADFrame, Data1(0), 32000)
wDasErr = K_BufListAdd (hADFrame, Data2(0), 32000)
. . .
```

Creating a Channel-Gain Queue

Before you create your channel-gain queue, you must declare an array of integers to accommodate the required number of entries. It is recommended that you declare an array two times the number of entries plus one. For example, to accommodate a channel-gain queue of 256 entries, you should declare an array of 513 integers ((256 x 2) + 1).

Next, you must fill the array with the channel-gain information. After you create the channel-gain queue, you must use **K_FormatChnGAry** to reformat the channel-gain queue so that it can be used by the DASCard-1000 Series Function Call Driver.

The following code fragment illustrates how to create a four-entry channel-gain queue called MyChanGainQueue for a DASCard-1002 card and how to use **K_SetChnGAry** to assign the starting address of MyChanGainQueue to the frame defined by hFrame.

```
. . .
Global MyChanGainQueue(9) As Integer ’(4 channels x 2) + 1
. . .
MyChanGainQueue(0) = 4     ’ Number of channel-gain pairs
MyChanGainQueue(1) = 0     ’ Channel 0
MyChanGainQueue(2) = 0     ’ Gain of 1
MyChanGainQueue(3) = 1     ’ Channel 1
MyChanGainQueue(4) = 1     ’ Gain of 2
MyChanGainQueue(5) = 2     ’ Channel 2
MyChanGainQueue(6) = 2     ’ Gain of 4
MyChanGainQueue(7) = 2     ’ Channel 2
MyChanGainQueue(8) = 3     ’ Gain of 8
. . .
wDasErr = K_FormatChnGAry (MyChanGainQueue(0))
wDasErr = K_SetChnGAry (hFrame, MyChanGainQueue(0))
. . .
```

Once formatted, your Visual Basic for Windows program can no longer read the channel-gain queue. To read or modify the array after it has been formatted, you must use **K_RestoreChnGAry** as follows:

```
. . .
wDasErr = K_RestoreChnGAry (MyChanGainQueue(0))
. . .
```

When you start the next analog input operation (using **K_SyncStart** or **K_IntStart**), channel 0 is sampled at a gain of 1, channel 1 is sampled at a gain of 2, channel 2 is sampled at a gain of 4, and so on.

Correcting Data

If you are using Visual Basic for Windows and have disabled automatic data correction, the application program must correct the data after the operation is complete. The following code fragment illustrates how to correct the data and how to convert the corrected data to volts.

```
Dim fCalData(2) As Single  ’ Contains calibration values
Dim fGainVal As Single     ’ Gain value to apply
Dim fOffsetVal As Single   ’ Offset value to apply
Dim nCounts As Integer     ’ Count value to convert

. . .
wDasErr = K_SetCalMode (hAD, 0)  ’ Disables correction by driver
. . .
’ Gets gain and offset values
wDasErr = K_GetCalData (hAD, 0, 0, 0, fCalData)
. . .
fGainVal = fCalData(0)
fOffsetVal = fCalData(1)
. . .
wDasErr = K_MoveBufToArray (aDataBuf(0), dwBufAddr(0), 100)
. . .
’ Corrects the data and converts counts to volts; assumes that
’ you are using a bipolar +/-5 V analog input range
nCounts = aDataBuf(0)
nCounts = nCounts * fGainVal + fOffsetVal       ’ Corrects data
PRINT nCounts * 0.00244                ’ 1 count = 0.00244 volts
```

Handling Errors

It is recommended that you always check the returned value (wDasErr in the previous examples) for possible errors. The following code fragment illustrates how to check the returned value of the **K_GetDevHandle** function.

```
. . .
wDasErr = K_GetDevHandle (hDrv, BoardNum, hDev)
If (wDasErr <> 0) Then
   MsgBox “K_GetDevHandle Error: ” + Hex$ (wDasErr),
      MB_ICONSTOP, “DASCARD-1000 ERROR”
   End
End If
. . .
```

Programming in Microsoft Visual Basic for Windows

To program in Microsoft Visual Basic for Windows, you need the following files; these files are provided in the ASO-1000 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DAS1000.DLL	Dynamic Link Library
DASDECL.BAS	Include file; must be added to the project
DAS1000.BAS	Include file; must be added to the project

To create an executable file from the Microsoft Visual Basic for Windows environment, choose Make EXE File from the File menu.

BASIC Programming Information

The following sections contain information you need to allocate and assign memory buffers, to create channel-gain queues, and to handle errors in BASIC, as well as other language-specific information for Microsoft QuickBasic and Microsoft Professional Basic.

Dynamically Allocating and Assigning Memory Buffers

This section provides code fragments that describe how to allocate and assign dynamically allocated memory buffers when programming in BASIC. Refer to the example programs on disk for more information.

Reducing the Memory Heap

By default, when BASIC programs run, all available memory is left for use by the internal memory manager. BASIC provides the SetMem function to distribute the available memory (the Far Heap). It is necessary to redistribute the Far Heap if you want to use dynamically allocated buffers. It is recommended that you include the following code at the beginning of BASIC programs to free the Far Heap for the driver's use.

```
FarHeapSize& = SetMem(0)
NewFarHeapSize& = SetMem(–FarHeapSize&/2)
```

Allocating a Single Memory Buffer

You can use a single, dynamically allocated memory buffer for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to use **KIntAlloc** to allocate a buffer of size Samples for the frame defined by hFrame and how to use **KSetBuf** to assign the starting address of the buffer.

```
. . .
Dim AcqBuf As Long      ’ Declare pointer to buffer
Dim hMem As Integer     ’ Declare integer for memory handle
. . .
wDasErr = KIntAlloc% (hFrame, Samples, AcqBuf, hMem)
wDasErr = KSetBuf% (hFrame, AcqBuf, Samples)
. . .
```

The following code illustrates how to use **KIntFree** to later free the allocated buffer, using the memory handle stored by **KIntAlloc**.

```
   . . .
   wDasErr = KIntFree% (hMem)
   . . .
```

Allocating Multiple Memory Buffers

You can use multiple, dynamically allocated memory buffers for interrupt-mode analog input operations.

The following code fragment illustrates how to use **KIntAlloc** to allocate five buffers of size Samples each for the frame defined by hADFrame and how to use **KBufListAdd** to assign the starting addresses of the five buffers.

```
. . .
Dim AcqBuf(4) As Long            ’ Declare 5 pointers to 5 buffers
Dim hMem(4) As Integer           ’ Declare 5 memory handles
. . .
for i% = 0 to 4
   wDasErr = KIntAlloc% (hFrame, Samples, AcqBuf(i%), hMem(i%))
   wDasErr = KBufListAdd% (hFrame, AcqBuf(i%), Samples)
next i%
. . .
```

The following code illustrates how to use **KIntFree** to later free the allocated buffers, using the memory handles stored by **KIntAlloc**; if you free the allocated buffers, you must also use **KBufListReset** to reset the buffer list associated with the frame.

```
   . . .
   for i% = 0 to 4
      wDasErr = KIntFree% (hMem(i%))
   next i%
   wDasErr = KBufListReset% (hADFrame)
   . . .
```

Accessing the Data from Buffers with Fewer than 64K Bytes

In BASIC, you cannot directly access analog input samples stored in a dynamically allocated memory buffer. You must use **KMoveBufToArray** to move a subset of the data (up to 32,766 samples) into a local array. The following code fragment illustrates how to move the first 100 samples of the second buffer in the multiple-buffer operation described in the previous section (AcqBuf(1)) into a local memory buffer.

```
. . .
Dim Buffer(1000) As Integer   ’ Declare local memory buffer
. . .
wDasErr = KMoveBufToArray% (Buffer(0), AcqBuf(1), 100)
. . .
```

Accessing the Data from Buffers with More than 64K Bytes

Under DOS, the CPU operates in real mode. Memory is addressed using a 32-bit segment:offset pair. Memory is allocated from the far heap, the reserve of conventional memory that occupies the first 640K bytes of the 1M byte of memory that the CPU can address in real mode. In the segmented real-mode architecture, the 16-bit segment:16-bit offset pair combines into a 20-bit linear address using an overlapping scheme. For a given segment value, you can address 64K bytes of memory by varying the offset.

When a memory buffer of more than 64K bytes (32K integer values) is used, multiple segments are required. When an offset reaches 64K bytes, the next linear memory address location can be accessed by adding &h1000 to the buffer segment and resetting the offset to zero.

Table 3-5 illustrates the mapping of consecutive memory locations at a segment page boundary.

Table 3-5. Real-Mode Memory Architecture

Segment:Offset	20-Bit Linear Address
. . . . :	. . . . .
74E4:FFFE	84E3E
74E4:FFFF	84E3F
84E4:0000	84E40
84E4:0001	84E41
. . . . :	. . . . .

The following code fragment illustrates how to move 1,000 values from a memory buffer (AcqBuf) allocated with 50,000 values to the program's local array (Array), starting at the sample at buffer index 40,000. You must first calculate the linear address of the buffer's starting point, then add the number of bytes deep into the buffer that the desired starting sample is located, and finally convert this adjusted linear address to a segment:offset format.

```
Dim AcqBuf As Long
Dim NumSamps As Long
Dim LinAddrBuff As Long
Dim DesLocAddr As Long
Dim AdjSegOffset As Long

Dim Array(1000) As Integer
. . .                  'Initialize array with desired values
NumSamps = 50000
wDasErr = KIntAlloc% (hFrame, NumSamps, AcqBuf, hMem)
. . .
'Acquisition routine
. . .
DesiredSamp = 40000
DesiredByte = DesiredSamp * 2       'Number of bytes into buffer

'To obtain the 20-bit linear address of the buffer, shift the
'segment:offset to the right 16 bits (leaves segment only),
'multiply by 16, then add offset
LinAddrBuff = (AcqBuf / &h10000) * 16 + (AcqBuf AND &hFFFF)
```

```
'20-bit linear address of desired location in buffer
DesLocAddr = LinAddrBuff + DesiredByte

'Convert desired location to segment:offset format
AdjSegOffset = (DesLocAddr / 16) * &h10000 + (DesLocAddr AND &hF)

wDasErr = KMoveBufToArray% (Array(0), AdjSegOffset, 1000)
```

To move more than 32,767 values from the memory buffer to the program's local array, the program must call **KMoveBufToArray** more than once. For example, assume that pBuf is a pointer to a dynamically allocated buffer that contains 65,536 values. The following code fragment illustrates how to move 65,536 values from the memory buffer to the program's local array (Data).

Although it is recommended that you perform all calculations on the linear address and then convert the result to the segment:offset format (as shown in the previous code fragment), this example illustrates an alternative method of calculating the address by working on the segment:offset form of the address directly. You can use this method if you already know how deep you want to go into the buffer with each move and the offset of the starting buffer address is zero, as is the case when the buffer is allocated with **KIntAlloc**. In this method, you add &h10000000 to the buffer address for each 64K byte page and then add the remainder of the buffer.

```
...
Dim Data[3,16384] As Integer
...
wDasErr = KMoveBufToArray% (Data(0,0), pBuf, 16384)

'Same segment, add 32,768 bytes to offset: add &h8000
wDasErr = KMoveBufToArray% (Data(1,0), pBuf + &h8000, 16384)

'Next segment, offset = 0: add &h10000000
wDasErr = KMoveBufToArray% (Data(2,0), pBuf + &h10000000, 16384)

'Next segment, remainder = 32,768 bytes: add &h10008000
wDasErr = KMoveBufToArray% (Data(3,0), pBuf + &h10008000, 16384)
```

Dimensioning and Assigning Local Arrays

This section provides code fragments that describe how to dimension and assign local arrays when programming in BASIC. Refer to the example programs on disk for more information.

Dimensioning a Single Array

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations.

The following code fragment illustrates how to dimension an array of 10,000 samples for the frame defined by hFrame and how to use **KSetBufI** to assign the starting address of the array.

```
   . . .
   Dim Data(9999) As Integer        ’ Allocate array
   . . .
   wDasErr = KSetBufI% (hFrame, Data(0), 10000)
   . . .
```

Dimensioning Multiple Arrays

You can use multiple, local arrays for interrupt-mode analog input operations.

The following code fragment illustrates how to dimension two arrays of 32,000 samples each for the frame defined by hADFrame and how to use **KBufListAdd** to assign the starting addresses of the arrays.

```
. . .
Dim Data1(31999) As Integer      ’ Allocate Array #1
Dim Data2(31999) As Integer      ’ Allocate Array #2
. . .
wDasErr = KBufListAdd% (hADFrame, Data1(0), 32000)
wDasErr = KBufListAdd% (hADFrame, Data2(0), 32000)
. . .
```

Note: The declaration of the second parameter of the **KBufListAdd** function in the DASDECL.BI file is not applicable to local arrays. You must change the declaration of *pBuf* from BYVAL *pBuf* AS LONG to SEG *pBuf* AS INTEGER.

Creating a Channel-Gain Queue

Before you create your channel-gain queue, you must declare an array of integers to accommodate the required number of entries. It is recommended that you declare an array two times the number of entries plus one. For example, to accommodate a channel-gain queue of 256 entries, you should declare an array of 513 integers ((256 x 2) + 1).

Next, you must fill the array with the channel-gain information. After you create the channel-gain queue, you must use **KFormatChnGAry** to reformat the channel-gain queue so that it can be used by the DASCard-1000 Series Function Call Driver.

The following code fragment illustrates how to create a four-entry channel-gain queue called MyChanGainQueue for a DASCard-1002 card and how to use **KSetChnGAry** to assign the starting address of MyChanGainQueue to the frame defined by hFrame.

```
. . .
Dim MyChanGainQueue(9) As Integer ’(4 channels x 2) + 1
. . .
MyChanGainQueue(0) = 4     ’ Number of channel-gain pairs
MyChanGainQueue(1) = 0     ’ Channel 0
MyChanGainQueue(2) = 0     ’ Gain of 1
MyChanGainQueue(3) = 1     ’ Channel 1
MyChanGainQueue(4) = 1     ’ Gain of 2
MyChanGainQueue(5) = 2     ’ Channel 2
MyChanGainQueue(6) = 2     ’ Gain of 4
MyChanGainQueue(7) = 2     ’ Channel 2
MyChanGainQueue(8) = 3     ’ Gain of 8
. . .
wDasErr = KFormatChnGAry% (MyChanGainQueue(0))
wDasErr = KSetChnGAry% (hFrame, MyChanGainQueue(0))
. . .
```

Once formatted, your BASIC program can no longer read the channel-gain array. To read or modify the array after it has been formatted, you must use **KRestoreChnGAry** as follows:

```
. . .
wDasErr = KRestoreChnGAry% (MyChanGainQueue(0))
. . .
```

When you start the next analog input operation (using **KSyncStart** or **KIntStart**), channel 0 is sampled at a gain of 1, channel 1 is sampled at a gain of 2, channel 2 is sampled at a gain of 4, and so on.

Correcting Data

If you are using BASIC, the application program must correct the data after the operation is complete. The following code fragment illustrates how to correct the data and how to convert the corrected data to volts.

```
DIM nCalData(2) AS INTEGER    ’ Contains calibration values
DIM fGainVal AS SINGLE        ’ Gain value to apply
DIM nOffsetVal AS INTEGER     ’ Offset value to apply
DIM nCounts AS INTEGER        ’ Count value to convert

. . .
’ Gets gain and offset values
wDasErr = KGETCALDATA% (hAD, 0, 0, 0, nCalData)
. . .
’ Determines the actual gain to apply to the data
fGainVal = 1! + nCalData(0) * 0.001
nOffsetVal = nCalData(1)
. . .
wDasErr = KMOVEBUFTOARRAY% (aDataBuf(0), dwBufAddr(0), 100)
. . .
’ Corrects the data and converts counts to volts; assumes that
’ you are using a bipolar +/-5 V analog input range
nCounts = aDataBuf(0)
nCounts = nCounts * fGainVal + nOffsetVal       /* Corrects data */
PRINT nCounts * 0.00244                ’ 1 count = 0.00244 volts
```

Handling Errors

It is recommended that you always check the returned value (wDasErr in the previous examples) for possible errors. The following code fragment illustrates how to check the returned value of the **DAS1000GetDevHandle** function.

```
. . .
wDasErr = DAS1000GETDEVHANDLE% (BoardNum, hDev)
IF (wDasErr <> 0) THEN
BEEP
PRINT “Error”;HEX$(wDasErr);“occurred during‘DAS1000GETDEVHANDLE%’”
 END
END IF
. . .
```

Programming in Microsoft QuickBasic

To program in Microsoft QuickBasic, you need the following files; these files are provided in the DASCard-1000 Series standard software package.

File	Description
D1000Q45.LIB	Linkable driver for QuickBasic (Version 4.5) stand-alone, executable (.EXE) programs
D1000Q45.QLB	Command-line loadable driver for the QuickBasic (Version 4.5) integrated environment
QB4DECL.BI	Include file
DASDECL.BI	Include file
DAS1000.BI	Include file

For Microsoft QuickBasic, you can create an executable file from within the programming environment, or you can use a compile and link statement.

To create an executable file from within the programming environment, perform the following steps:

1. Enter the following to invoke the environment:

   ```
   QB /L D1000Q45 filename.bas
   ```

   where *filename* indicates the name of your application program.
2. From the Run menu, choose Make EXE File.

To use a compile and link statement, enter the following:

```
BC filename.bas /O
Link filename.obj,,,D1000Q45.lib+BCOM45.lib;
```

where *filename* indicates the name of your application program.

Programming in Microsoft Professional Basic

To program in Microsoft Professional Basic, you need the following files; these files are provided in the DASCard-1000 Series standard software package.

File	Description
D1000QBX.LIB	Linkable driver for Professional Basic, stand-alone, executable (.EXE) programs
D1000QBX.QLB	Command-line loadable driver for the Professional Basic integrated environment
DASDECL.BI	Include file
DAS1000.BI	Include file

For Microsoft Professional Basic, you can create an executable file from within the programming environment, or you can use a compile and link statement.

To create an executable file from within the programming environment, perform the following steps:

1. Enter the following to invoke the environment:

   ```
   QBX /L D1000QBX filename.bas
   ```

   where *filename* indicates the name of your application program.

2. From the Run menu, choose Make EXE File.

To use a compile and link statement, enter the following:

```
BC filename.bas /o;
Link filename.obj,,,D1000QBX.lib;
```

where *filename* indicates the name of your application program.

4
Function Reference

The FCD functions are organized into the following groups:

- Initialization functions
- Operation functions
- Frame management functions
- Memory management functions
- Buffer address functions
- Buffering mode functions
- Channel and gain functions
- Clock functions
- Trigger functions
- Data correction functions
- Miscellaneous functions

The particular functions associated with each function group are presented in Table 4-1. The remainder of the chapter presents detailed descriptions of all the FCD functions, arranged in alphabetical order.

Table 4-1. Functions

Function Type	Function Name	Page Number
Initialization	DAS1000_DevOpen	page 4-6
	DAS1000_GetDevHandle	page 4-10
	K_OpenDriver	page 4-62
	K_CloseDriver	page 4-19
	K_GetDevHandle	page 4-44
	K_FreeDevHandle	page 4-31
	K_DASDevInit	page 4-24
Operation	K_ADRead	page 4-12
	K_DIRead	page 4-25
	K_DOWrite	page 4-27
	K_SyncStart	page 4-100
	K_IntStart	page 4-54
	K_IntStatus	page 4-55
	K_IntStop	page 4-58
Frame Management	K_GetADFrame	page 4-35
	K_FreeFrame	page 4-32
	K_ClearFrame	page 4-18
Memory Management	K_IntAlloc	page 4-51
	K_IntFree	page 4-53
	K_MoveBufToArray	page 4-60
Buffer Address	K_SetBuf	page 4-71
	K_SetBufI	page 4-73
	K_BufListAdd	page 4-14
	K_BufListReset	page 4-16
Buffering Mode	K_SetContRun	page 4-85
	K_ClrContRun	page 4-20

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Channel and Gain	K_SetChn	page 4-77
	K_SetStartStopChn	page 4-91
	K_SetG	page 4-89
	K_SetStartStopG	page 4-93
	K_SetChnGAry	page 4-79
	K_FormatChnGAry	page 4-29
	K_RestoreChnGAry	page 4-64
	K_SetADConfig	page 4-65
	K_SetADMode	page 4-67
	K_GetADConfig	page 4-33
	K_GetADMode	page 4-37
Clock	K_SetClk	page 4-81
	K_SetClkRate	page 4-83
	K_GetClkRate	page 4-42
Trigger	K_SetTrig	page 4-96
	K_SetADTrig	page 4-69
	K_SetTrigHyst	page 4-98
	K_SetDITrig	page 4-87
Data Correction	K_CorrectData	page 4-22
	K_GetCalData	page 4-39
	K_SetCalMode	page 4-75
Miscellaneous	K_GetErrMsg	page 4-46
	K_GetVer	page 4-49
	K_GetShellVer	page 4-47
	DAS1000_GetCardInfo	page 4-8

Keep the following conventions in mind throughout this chapter:

- Although the function names are shown with underscores, do not use the underscores in the BASIC languages.
- The data types DWORD, WORD, and BYTE are defined in the language-specific include files.
- Variable names are shown in italics.
- The return value for all FCD functions is an integer error/status code. Error/status code 0 indicates that the function executed successfully. A nonzero error/status code indicates that an error occurred. Refer to Appendix A for additional information.
- In the usage section, the variables are not defined. It is assumed that the variables are defined as shown in the prototype. The name of each variable in both the prototype and usage sections includes a prefix that indicates the associated data type. These prefixes are described in Table 4-2.

Table 4-2. Data Type Prefixes

Prefix	Data Type	Comments
sz	Pointer to string terminated by zero	This data type is typically used for variables that specify the driver's configuration file name.
h	Handle to device, frame, and memory block	This data type is used for handle-type variables. You declare handle-type variables in your program as long or DWORD, depending on the language you are using. The actual variable is passed to the driver by value.
ph	Pointer to a handle-type variable	This data type is used when calling the FCD functions to get a driver handle, a frame handle, or a memory handle. The actual variable is passed to the driver by reference.
p	Pointer to a variable	This data type is used for pointers to all types of variables, except handles (h). It is typically used when passing a parameter of any type to the driver by reference.
n	Number value	This data type is used when passing a number, typically a byte, to the driver by value.
w	16-bit word	This data type is typically used when passing an unsigned integer to the driver by value.
a	Array	This data type is typically used in conjunction with other prefixes listed here; for example, *anVar* denotes an array of numbers.
f	Float	This data type denotes a single-precision floating-point number.
d	Double	This data type denotes a double-precision floating-point number.
dw	32-bit double word	This data type is typically used when passing an unsigned long to the driver by value.

DAS1000_DevOpen

Purpose

Initializes the DASCard-1000 Series Function Call Driver.

Prototype

C/C++
DASErr far pascal DAS1000_DevOpen (char far **szCfgFile*,
char far **pBoards*);

Visual Basic for Windows
Declare Function DAS1000_DevOpen Lib "DAS1000.DLL"
(ByVal *szCfgFile* As String, *pBoards* As Integer) As Integer

BASIC
DECLARE FUNCTION DAS1000DEVOPEN% ALIAS
"DAS1000_DevOpen" (BYVAL *szCfgFile* AS LONG,
SEG *pBoards* AS INTEGER)

Parameters

szCfgFile	Driver configuration file. Valid values: The name of a configuration file. **0** for DAS1000.CFG
pBoards	Number of cards defined in *szCfgFile*. Value stored: **1** or **2**

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the driver according to the information in the configuration file specified by *szCfgFile* and stores the number of DASCard-1000 Series cards defined in *szCfgFile* in *pBoards*.

You create a configuration file using the CFG1000.EXE utility. If *szCfgFile* = 0, **DAS1000_DevOpen** looks for the DAS1000.CFG configuration file in the current directory and uses those settings, if available. Refer to your *DASCard-1000 Series User's Guide* for more information about configuration files.

See Also

K_OpenDriver

Usage

C/C++

```
#include "DAS1000.H"   // Use DAS1000.HPP for C++
...
char nBoards;
...
wDasErr = DAS1000_DevOpen ("DAS1002.CFG", &nBoards);
```

Visual Basic for Windows

(Add DAS1000.BAS to your project)

```
...
DIM szCfgName AS STRING
DIM nBoards AS INTEGER
...
szCfgName = "DAS1002.CFG" + CHR$(0)
wDasErr = DAS1000_DevOpen(szCfgName, nBoards)
```

BASIC

```
' $INCLUDE: 'DAS1000.BI'
...
DIM szCfgName AS STRING
DIM nBoards AS INTEGER
...
szCfgName = "DAS1002.CFG" + CHR$(0)
wDasErr = DAS1000DEVOPEN%(SSEGADD(szCfgName),nBoards)
```

DAS1000_GetCardInfo

Purpose Gets the system resources allocated for a DASCard-1000 Series card.

Prototype

C/C++
DASErr far pascal DAS1000_GetCardInfo (WORD *nBrdNum*,
short far **pSocket*, short far **pIRQ*, short far **pIOBase*,
short far **pMemBase*, short far **pBrdType*);

Visual Basic for Windows
Declare Function DAS1000_GetCardInfo Lib "DAS1000.DLL"
(ByVal *nBrdNum* As Integer, *pSocket* As Integer, *pIRQ* As Integer,
pIOBase As Integer, *pMemBase* As Integer, *pBrdType* As Integer)
As Integer

BASIC
DECLARE FUNCTION DAS1000GETCARDINFO% ALIAS
"DAS1000_GetCardInfo" (BYVAL *nBrdNum* AS INTEGER,
SEG *pSocket* AS INTEGER, SEG *pIRQ* AS INTEGER,
SEG *pIOBase* AS INTEGER, SEG *pMemBase* AS INTEGER,
SEG *pBrdType* AS INTEGER)

Parameters

nBrdNum	Card number. Valid values: **0** or **1**
pSocket	Socket in which the card is installed.
pIRQ	Interrupt level allocated by PCMCIA software.
pIOBase	Base address allocated by PCMCIA software.
pMemBase	Memory segment address allocated by PCMCIA software.
pBrdType	Card type. Value stored: **0** for DASCard-1001 **1** for DASCard-1002 **2** for DASCard-1003

Return Value Error/status code. Refer to Appendix A.

Remarks

For the DASCard-1000 Series card specified by *nBrdNum*, this function stores the socket number in *pSocket*, the interrupt level in *pIRQ*, the base address in *pIOBase*, the memory segment address in *pMemBase*, and the card type in *pBrdType*.

The card number specified in *nBrdNum* refers to the card number specified in the configuration file. If you are using one card, *nBrdNum* is always 0; if you are using two cards, *nBrdNum* is the same as the socket number.

Usage

C/C++

```
#include "DAS1000.H"   // Use DAS1000.HPP for C++
...
WORD wSock, wIRQ, wIO, wMem, wType;
...
wDasErr = DAS1000_GetCardInfo (0,&wSock,&wIRQ,&wIO,&wMem,&wType);
```

Visual Basic for Windows

(Add DAS1000.BAS to your project)

```
...
Global wSock As Integer
Global wIRQ As Integer
Global wIO As Integer
Global wMem As Integer
Global wType As Integer
...
wDasErr = DAS1000_GetCardInfo (0, wSock, wIRQ, wIO, wMem, wType)
```

BASIC

```
' $INCLUDE: 'DAS1000.BI'
...
DIM wSock AS INTEGER
DIM wIRQ AS INTEGER
DIM wIO AS INTEGER
DIM wMem AS INTEGER
DIM wType AS INTEGER
...
wDasErr = DAS1000GETCARDINFO% (0, wSock, wIRQ, wIO, wMem, wType)
```

DAS1000_GetDevHandle

Purpose

Initializes a DASCard-1000 Series card.

Prototype

C/C++
DASErr far pascal DAS1000_GetDevHandle (WORD *nBrdNum*, DWORD far **phDev*);

Visual Basic for Windows
Declare Function DAS1000_GetDevHandle Lib "DAS1000.DLL" (ByVal *nBrdNum* As Integer, *phDev* As Long) As Integer

BASIC
DECLARE FUNCTION DAS1000GETDEVHANDLE% ALIAS "DAS1000_GetDevHandle" (BYVAL *nBrdNum* AS INTEGER, SEG *phDev* AS LONG)

Parameters

nBrdNum	Card number. Valid values: **0** or **1**
phDev	Handle associated with the card.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DASCard-1000 Series card specified by *nBrdNum* and stores the device handle of the specified card in *phDev.*

The card number specified in *nBrdNum* refers to the card number specified in the configuration file. If you are using one card, *nBrdNum* is always 0; if you are using two cards, *nBrdNum* is the same as the socket number.

The value stored in *phDev* is intended to be used exclusively as an argument to functions that require a device handle. Your program should not modify the value stored in *phDev.*

See Also

K_GetDevHandle

Usage

C/C++

```
#include "DAS1000.H"   // Use DAS1000.HPP for C++
...
DWORD hDev;
...
wDasErr = DAS1000_GetDevHandle (0, &hDev);
```

Visual Basic for Windows

(Add DAS1000.BAS to your project)

```
...
Global hDev As Long   ' Device Handle
...
wDasErr = DAS1000_GetDevHandle (0, hDev)
```

BASIC

```
' $INCLUDE: 'DAS1000.BI'
...
DIM hDev AS LONG   ' Device Handle
...
wDasErr = DAS1000GETDEVHANDLE% (0, hDev)
```

K_ADRead

Purpose

Reads a single analog input value.

Prototype

C/C++
DASErr far pascal K_ADRead (DWORD *hDev*, BYTE *nChan*, BYTE *nGain*, void far **pData*);

Visual Basic for Windows
Declare Function K_ADRead Lib "DASSHELL.DLL" (ByVal *hDev* As Long, ByVal *nChan* As Integer, ByVal *nGain* As Integer, *pData* As Integer) As Integer

BASIC
DECLARE FUNCTION KADREAD% ALIAS "K_ADRead" (BYVAL *hDev* AS LONG, BYVAL *nChan* AS INTEGER, BYVAL *nGain* AS INTEGER, SEG *pData* AS INTEGER)

Parameters

hDev — Handle associated with the card.

nChan — Analog input channel.
Valid values: **0** to **255**

nGain — Gain code.
Valid values:

Card	Gain	Gain Code
DASCard-1001	1	0
	10	1
	100	2
	500	3
DASCard-1002	1	0
	2	1
	4	2
	8	3
DASCard-1003	1	0

pData	Acquired analog input value.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function reads the analog input channel *nChan* on the DASCard-1000 Series card specified by *hDev* at the gain represented by *nGain*, and stores the count value in *pData*.

If you are programming under Windows, the count value in *pData* is the corrected count value. If you are programming under DOS, the count value in *pData* is uncorrected and the application program must correct it. Refer to page 2-22 for information.

Refer to Appendix B for information on converting the corrected count value stored in *pData* to voltage.

See Also

K_IntStart, K_SyncStart

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
int wADValue;
...
wDasErr = K_ADRead (hDev, 0, 0, &wADValue);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global wADValue As Integer
...
wDasErr = K_ADRead (hDev, 0, 0, wADValue)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wADValue AS INTEGER
...
wDasErr = KADREAD% (hDev, 0, 0, wADValue)
```

K_BufListAdd

Purpose

Adds a buffer or array to the list of multiple buffers/arrays.

Prototype

C/C++
DASErr far pascal K_BufListAdd (DWORD *hFrame*, void far **pBuf*, DWORD *dwSamples*);

Visual Basic for Windows
Declare Function K_BufListAdd Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *pBuf* As Long,
ByVal *dwSamples* As Long) As Integer

BASIC
DECLARE FUNCTION KBUFLISTADD% ALIAS "K_BufListAdd"
(BYVAL *hFrame* AS LONG, BYVAL *pBuf* AS LONG,
BYVAL *dwSamples* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer or array.
dwSamples	Number of samples in the buffer or array. Valid values: **1** to **5000000**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the interrupt-mode operation defined by *hFrame*, this function adds the buffer or array at the address pointed to by *pBuf* to the list of multiple buffers/arrays; the number of samples in the buffer/array is specified in *dwSamples*.

Since multiple buffers/arrays are not supported in synchronous mode, you cannot use this function for synchronous-mode operations.

Before you add the buffer or array to the list, you must either allocate the buffer dynamically using **K_IntAlloc**, or dimension the array locally.

If you are programming in BASIC and using this function to specify the starting address of a local array, the declaration of *pBuf* in the DASDECL.BI file is not applicable. You must change the declaration of *pBuf* from BYVAL *pBuf* AS LONG to SEG *pBuf* AS INTEGER.

Make sure that you add buffers or arrays to the list in the order in which you want to use them. The first buffer/array you add is #1, the second buffer you add is #2, and so on. You can add up to 149 buffers. You can use **K_IntStatus** to determine which buffer/array is currently in use.

See Also K_BufListReset, K_IntAlloc

Usage

C/C++

```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
void far *pBuf[5];   // Buffer pointers
WORD hMem[5];   // Buffer handles
...
for (i = 0; i < 5; i++) {
   wDasErr = K_IntAlloc (hAD, dwSamples, &pBuf[i], &hMem[i]);
   wDasErr = K_BufListAdd (hAD, pBuf[i], dwSamples);
}
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global pBuf(4) As Long   ' Buffer pointers
Global hMem(4) As Integer   ' Buffer handles
...
For I% = 0 To 4
   wDasErr = K_IntAlloc (hAD, dwSamples, pBuf(I%), hMem(I%))
   wDasErr = K_BufListAdd (hAD, pBuf(I%), dwSamples)
Next I%
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM pBuf(4) AS LONG   ' Buffer pointers
DIM hMem(4) AS INTEGER   ' Buffer handles
...
FOR I% = 0 TO 4
   wDasErr = KINTALLOC% (hAD, dwSamples, pBuf(I%), hMem(I%))
   wDasErr = KBUFLISTADD% (hAD, pBuf(I%), dwSamples)
NEXT I%
```

K_BufListReset

Purpose Clears the list of multiple buffers/arrays.

Prototype

C/C++
DASErr far pascal K_BufListReset (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_BufListReset Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KBUFLISTRESET% ALIAS "K_BufListReset"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value Error/status code. Refer to Appendix A.

Remarks For the interrupt-mode operation defined by *hFrame*, this function clears all buffers or arrays from the list of multiple buffers/arrays.

This function does not deallocate buffers in the list. If dynamically allocated buffers are no longer needed, you can use **K_IntFree** to free the buffers before resetting the list.

See Also K_BufListAdd

Usage

C/C++
```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
wDasErr = K_BufListReset (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_BufListReset (hAD)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KBUFLISTRESET% (hAD)
```

K_ClearFrame

Purpose

Sets the elements of a frame to their default values.

Prototype

C/C++
DASErr far pascal K_ClearFrame (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_ClearFrame Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KCLEARFRAME% ALIAS "K_ClearFrame"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function sets the elements of the frame specified by *hFrame* to their default values.

Refer to page 3-4 for the default values of an A/D frame.

See Also

K_GetADFrame

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_ClearFrame (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_ClearFrame (hAD)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KCLEARFRAME% (hAD)
```

K_CloseDriver

Purpose Closes a previously initialized Keithley DAS Function Call Driver.

Prototype

C/C++
DASErr far pascal K_CloseDriver (DWORD *hDrv*);

Visual Basic for Windows
Declare Function K_CloseDriver Lib "DASSHELL.DLL"
(ByVal *hDrv* As Long) As Integer

BASIC
Not supported

Parameters

hDrv	Driver handle you want to free.

Return Value Error/status code. Refer to Appendix A.

Remarks This function frees the driver handle specified by *hDrv* and closes the associated use of the Function Call Driver. This function also frees all device handles and frame handles associated with *hDrv*.

If *hDrv* is the last driver handle specified for the Function Call Driver, the driver is shut down (for all languages) and unloaded (for Windows-based languages only).

See Also K_FreeDevHandle

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_CloseDriver (hDrv);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_CloseDriver (hDrv)
```

K_ClrContRun

Purpose

Specifies single-cycle buffering mode.

Prototype

C/C++
DASErr far pascal K_ClrContRun (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_ClrContRun Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KCLRCONTRUN% ALIAS "K_ClrContRun"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function sets the buffering mode for the operation defined by *hFrame* to single-cycle mode and sets the Buffering Mode element in the frame accordingly.

K_GetADFrame and **K_ClearFrame** also enable single-cycle buffering mode.

Refer to page 2-17 for more information on buffering modes.

See Also

K_SetContRun

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_ClrContRun (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_ClrContRun (hAD)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KCLRCONTRUN% (hAD)
```

K_CorrectData

Purpose

Corrects acquired analog input values using the calibration factors stored in computer memory.

Prototype

C/C++
DASErr far pascal K_CorrectData (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_CorrectData Lib "DAS1000.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
Not supported

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function corrects the analog data in the memory buffer or array and stores the corrected values back in the same buffer or array.

This function is available only when programming under Windows.

To ensure accurate results, make sure that you do not modify any of the elements in the frame specified by *hFrame* from the time you start the analog input operation until the time you call this function. If you want to perform another analog input operation before you correct data from a previous analog input operation, make sure that you use a different frame with a different buffer address.

See Also

K_GetCalData, K_SetCalMode

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_CorrectData (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
wDasErr = K_CorrectData (hAD)
```

K_DASDevInit

Purpose Reinitializes a Keithley MetraByte DAS card or board.

Prototype

C/C++
DASErr far pascal K_DASDevInit (DWORD *hDev*);

Visual Basic for Windows
Declare Function K_DASDevInit Lib "DASSHELL.DLL"
(ByVal *hDev* As Long) As Integer

BASIC
DECLARE FUNCTION KDASDEVINIT% ALIAS "K_DASDevInit"
(BYVAL *hDev* AS LONG)

Parameters

hDev	Handle associated with the card or board.

Return Value Error/status code. Refer to Appendix A.

Remarks This function stops all current operations on the card or board specified by *hDev*.

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_DASDevInit (hDev);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_DASDevInit (hDev)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KDASDEVINIT% (hDev)
```

K_DIRead

Purpose Reads a single digital input value.

Prototype

C/C++
DASErr far pascal K_DIRead (DWORD *hDev*, BYTE *nChan*,
void far **pData*);

Visual Basic for Windows
Declare Function K_DIRead Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, ByVal *nChan* As Integer, *pData* As Any)
As Integer

BASIC
DECLARE FUNCTION KDIREAD% ALIAS "K_DIRead"
(BYVAL *hDev* AS LONG, BYVAL *nChan* AS INTEGER,
SEG *pData* AS ANY)

Parameters

hDev	Handle associated with the card.
nChan	Digital input channel. Valid value: **0**
pData	Digital input value.

Return Value Error/status code. Refer to Appendix A.

Remarks This function reads the values of all digital input lines on the DASCard-1000 Series card specified by *hDev* and stores the value in *pData*.

The acquired digital value is stored in bits 0, 1, 2, and 3 of *pData*; the values in the remaining bits are not defined. Refer to page 2-25 for more information.

See Also K_DOWrite

K_DIRead (cont.)

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wDIValue;
...
wDasErr = K_DIRead (hDev, 0, &wDIValue);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global wDIValue As Integer
...
wDasErr = K_DIRead (hDev, 0, wDIValue)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wDIValue AS INTEGER
...
wDasErr = KDIREAD% (hDev, 0, wDIValue)
```

K_DOWrite

Purpose

Writes a single digital output value to the digital output channel.

Prototype

C/C++
DASErr far pascal K_DOWrite (DWORD *hDev*, BYTE *nChan*, DWORD *dwData*);

Visual Basic for Windows
Declare Function K_DOWrite Lib "DASSHELL.DLL" (ByVal *hDev* As Long, ByVal *nChan* As Integer, ByVal *dwData* As Long) As Integer

BASIC
DECLARE FUNCTION KDOWRITE% ALIAS "K_DOWrite" (BYVAL *hDev* AS LONG, BYVAL *nChan* AS INTEGER, BYVAL *dwData* AS LONG)

Parameters

hDev	Handle associated with the card.
nChan	Digital output channel. Valid value: **0**
dwData	Digital output value. Valid values: **0** to **255**

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function writes the value *dwData* to the digital output lines on the DASCard-1000 Series card specified by *hDev*.

Refer to page 2-24 for a description of the digital output lines.

See Also

K_DIRead

K_DOWrite (cont.)

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD dwDOValue;
...
dwDOValue = 0x5;
wDasErr = K_DOWrite (hDev, 0, dwDOValue);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global dwDOValue As Long
...
dwDOValue = &H5
wDasErr = K_DOWrite (hDev, 0, dwDOValue)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwDOValue AS LONG
...
dwDOValue = &H5
wDasErr = KDOWRITE% (hDev, 0, dwDOValue)
```

K_FormatChnGAry

Purpose

Converts the format of a channel-gain queue.

Prototype

C/C++
Not supported

Visual Basic for Windows
Declare Function K_FormatChnGAry Lib "DASSHELL.DLL" (*pArray* As Integer) As Integer

BASIC
DECLARE FUNCTION KFORMATCHNGARY% ALIAS "K_FormatChnGAry" (SEG *pArray* AS INTEGER)

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function converts a channel-gain queue created in BASIC or Visual Basic for Windows using 16-bit values to a channel-gain queue of 8-bit values that the **K_SetChnGAry** function can use, and stores the starting address of the converted channel-gain queue in *pArray*.

After you use this function, your program can no longer read the converted channel-gain queue. You must use the **K_RestoreChnGAry** function to return the queue to its original format. Refer to page 4-64 for more information.

See Also

K_SetChnGAry, K_RestoreChnGAry

K_FormatChnGAry (cont.)

Usage

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global ChanGainArray(16) As Integer   ' Chan/Gain array
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 0
ChanGainArray(3) = 1: ChanGainArray(4) = 1
wDasErr = K_FormatChnGAry (ChanGainArray(0))
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ChanGainArray(16) AS INTEGER   ' Chan/Gain array
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 0
ChanGainArray(3) = 1: ChanGainArray(4) = 1
wDasErr = KFORMATCHNGARY% (ChanGainArray(0))
```

Purpose Frees a previously specified device handle.

Prototype

C/C++
DASErr far pascal K_FreeDevHandle (DWORD *hDev*);

Visual Basic for Windows
Declare Function K_FreeDevHandle Lib "DASSHELL.DLL"
(ByVal *hDev* As Long) As Integer

BASIC
Not supported

Parameters

hDev	Device handle you want to free.

Return Value Error/status code. Refer to Appendix A.

Remarks This function frees the device handle specified by *hDev* as well as all frame handles associated with *hDev.*

See Also K_GetDevHandle

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_FreeDevHandle (hDev);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_FreeDevHandle (hDev)
```

K_FreeFrame

Purpose

Frees a frame.

Prototype

C/C++

DASErr far pascal K_FreeFrame (DWORD *hFrame*);

Visual Basic for Windows

Declare Function K_FreeFrame Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC

DECLARE FUNCTION KFREEFRAME% ALIAS "K_FreeFrame"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to frame you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function frees the frame specified by *hFrame*, making the frame available for another operation.

See Also

K_GetADFrame

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_FreeFrame (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_FreeFrame (hAD)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KFREEFRAME% (hAD)
```

K_GetADConfig

Purpose

Gets the analog input channel configuration.

Prototype

C/C++
DASErr far pascal K_GetADConfig (DWORD *hDev*,
WORD far **pMode*);

Visual Basic for Windows
Declare Function K_GetADConfig Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *pMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KGETADCONFIG% ALIAS
"K_GetADConfig" (BYVAL *hDev* AS LONG,
SEG *pMode* AS INTEGER)

Parameters

hDev	Handle associated with the card.
pMode	Analog input channel configuration. Value stored: **0** for Differential **1** for Single-ended

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DASCard-1000 Series card specified by *hDev*, this function stores the code that represents the analog input channel configuration in *pMode*.

See Also

K_SetADConfig

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wADConfig;
...
wDasErr = K_GetADConfig (hDev, &wADConfig);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global wADConfig As Integer
...
wDasErr = K_GetADConfig (hDev, wADConfig)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wADConfig AS INTEGER
...
wDasErr = KGETADCONFIG% (hDev, wADConfig)
```

K_GetADFrame

Purpose

Accesses an A/D frame for an analog input operation.

Prototype

C/C++
DASErr far pascal K_GetADFrame (DWORD *hDev*,
DWORD far * *phFrame*);

Visual Basic for Windows
Declare Function K_GetADFrame Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *phFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KGETADFRAME% ALIAS "K_GetADFrame"
(BYVAL *hDev* AS LONG, SEG *phFrame* AS LONG)

Parameters

hDev	Handle associated with the card.
phFrame	Handle to the frame that defines the operation.

Remarks

This function specifies that you want to perform a synchronous-mode or interrupt-mode analog input operation on the DASCard-1000 Series card specified by *hDev*, and accesses an available A/D frame with the handle *phFrame*.

The frame is initialized to its default settings; refer to Table 3-1 on page 3-4 for a list of the default settings.

The value stored in *phFrame* is intended to be used exclusively as an argument to functions that require a frame handle. Your program should not modify the value stored in *phFrame*.

See Also

K_ClearFrame, K_FreeFrame

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD hAD;
...
wDasErr = K_GetADFrame (hDev, &hAD);
```

K_GetADFrame (cont.)

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global hAD As Long
...
wDasErr = K_GetADFrame (hDev, hAD)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM hAD AS LONG
...
wDasErr = KGETADFRAME% (hDev, hAD)
```

K_GetADMode

Purpose Gets the analog input range type.

Prototype

C/C++
DASErr far pascal K_GetADMode (DWORD *hDev*,
WORD far **pMode*);

Visual Basic for Windows
Declare Function K_GetADMode Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *pMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KGETADMODE% ALIAS "K_GetADMode"
(BYVAL *hDev* AS LONG, SEG *pMode* AS INTEGER)

Parameters

hDev	Handle associated with the card.
pMode	Analog input range type. Value stored: **0** for Bipolar **1** for Unipolar

Return Value Error/status code. Refer to Appendix A.

Remarks For the DASCard-1000 Series card specified by *hDev*, this function stores the code that represents the analog input range type in *pMode.*

See Also K_SetADMode

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wADMode;
...
wDasErr = K_GetADMode (hDev, &wADMode);
```

K_GetADMode (cont.)

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global wADMode As Integer
...
wDasErr = K_GetADMode (hDev, wADMode)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wADMode AS INTEGER
...
wDasErr = KGETADMODE% (hDev, wADMode)
```

K_GetCalData

Purpose Gets the calibration factors and formula for correcting data.

Prototype

C/C++
DASErr far pascal K_GetCalData (DWORD *hFrame*, short *nChan*, short *nGain*, short far * *pType*, void far * *pCalData*);

Visual Basic for Windows
Declare Function K_GetCalData Lib "DAS1000.DLL" (ByVal *hFrame* As Long, ByVal *nChan* As Integer, ByVal *nGain* As Integer, *pType* As Integer, *pCalData* As Single) As Integer

BASIC
DECLARE FUNCTION KGETCALDATA% ALIAS "K_GetCalData" (BYVAL *hFrame* AS LONG, BYVAL *nChan* AS INTEGER, BYVAL *nGain* AS INTEGER, SEG *pType* AS INTEGER, SEG *pCalData* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nChan	Channel whose data you want to correct. Valid values: **0**

K_GetCalData (cont.)

nGain Gain code.
Valid values:

Card	Gain	Gain Code
DASCard-1001	1	0
	10	1
	100	2
	500	3
DASCard-1002	1	0
	2	1
	4	2
	8	3
DASCard-1003	1	0

pType Pointer to the location of the correction formula.

pCalData Pointer to a two-element array containing the calibration factors.

Return Value Error/status code. Refer to Appendix A.

Remarks For the operation defined by *hFrame*, this function gets the calibration factors and formula needed to correct the data sampled at the gain specified by *nGain*, and stores the location of the formula in *pType* and the location of the calibration factors in *pCalData*.

For DASCard-1000 Series cards, the channel number specified in *nChan* is ignored; therefore, you can use the default channel number 0.

The gain code specified in *nGain* must be the same as the gain code used when the analog input signal was sampled; if you specify a different gain code, the wrong calibration factors may be returned and your corrected data may be invalid.

For DASCard-1000 Series cards, only one formula is supported; therefore, you can pass a NULL pointer to *pType*. The value stored in *pType* is always 4, which represents the formula $y = mx + b$. Refer to page 2-22 for information on using this formula to correct data.

For DASCard-1000 Series cards, *pCalData* points to a two-element array, which contains the two calibration factors; the first element contains the gain to apply to the uncorrected data (*m*) and the second element contains the offset (*b*). If you are programming under Windows, dimension an array of floating point numbers; if you are programming under DOS, dimension an array of integers. Note that the actual gain value is returned, not the gain code. Refer to page 2-22 for more information.

See Also K_CorrectData, K_SetCalMode

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pCalData;
...
wDasErr = K_GetCalData (hAD, 0, 0, NULL, &pCalData);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
Global pCalData As Integer
...
wDasErr = K_GetCalData (hAD, 0, 0, 0, pCalData(0))
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
DIM pCalData AS INTEGER
...
wDasErr = KGETCALDATA% (hAD, 0, 0, 0, pCalData(0))
```

K_GetClkRate

Purpose

Gets the clock rate (number of clock ticks) used by the internal pacer clock.

Prototype

C/C++
DASErr far pascal K_GetClkRate (DWORD *hFrame*,
DWORD far **pRate*);

Visual Basic for Windows
Declare Function K_GetClkRate Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pRate* As Long) As Integer

BASIC
DECLARE FUNCTION KGETCLKRATE% ALIAS "K_GetClkRate"
(BYVAL *hFrame* AS LONG, SEG *pRate* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pRate	Number of clock ticks between conversions.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the number of clock ticks between conversions in *pRate*.

After a synchronous-mode or interrupt-mode analog input operation starts, the value stored in *pRate* represents the actual count used, not necessarily the count set by **K_SetClkRate**.

The *pRate* variable contains the value of the Pacer Clock Rate element.

See Also

K_SetClkRate

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD dwRate;
...
wDasErr = K_GetClkRate (hAD, &dwRate);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global dwRate As Long
...
wDasErr = K_GetClkRate (hAD, dwRate)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwRate AS LONG
...
wDasErr = KGETCLKRATE% (hAD, dwRate)
```

K_GetDevHandle

Purpose	Initializes any Keithley MetraByte DAS card or board.

Prototype

C/C++
DASErr far pascal K_GetDevHandle (DWORD *hDrv*,
WORD *nBoardNum*, DWORD far * *pDev*);

Visual Basic for Windows
Declare Function K_GetDevHandle Lib "DASSHELL.DLL"
(ByVal *hDrv* As Long, ByVal *nBoardNum* As Integer, *pDev* As Long)
As Integer

BASIC
Not supported

Parameters

hDrv	Driver handle of the associated Function Call Driver.
nBoardNum	Card number. Valid values: **0** or **1**
pDev	Handle associated with the card.

Return Value Error/status code. Refer to Appendix A.

Remarks

This function initializes the DASCard-1000 Series card associated with *hDrv* and specified by *nBoardNum*, and stores the device handle of the specified card in *pDev.*

The card number specified in *nBoardNum* refers to the card number specified in the configuration file. If you are using one card, *nBoardNum* is always 0; if you are using two cards, *nBoardNum* is the same as the socket number.

The value stored in *pDev* is intended to be used exclusively as an argument to functions that require a device handle. Your program should not modify the value stored in *pDev.*

See Also K_FreeDevHandle

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD hDev;
...
wDasErr = K_GetDevHandle (hDrv, 0, &hDev);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global hDev As Long
...
wDasErr = K_GetDevHandle (hDrv, 0, hDev)
```

K_GetErrMsg

Purpose Gets the address of an error message string.

Prototype

C/C++
DASErr far pascal K_GetErrMsg (DWORD *hDev*, short *nDASErr*, char far * far * *pErrMsg*);

Visual Basic for Windows
Not supported

BASIC
Not supported

Parameters

hDev	Handle associated with the card.
nDASErr	Error message number.
pErrMsg	Address of error message string.

Return Value Error/status code. Refer to Appendix A.

Remarks For the DASCard-1000 Series card specified by *hDev*, this function stores the address of the string corresponding to error message number *nDASErr* in *pErrMsg*.

Refer to page 2-4 and to page 3-20 for more information about error handling. Refer to Appendix A for a list of error codes and their meanings.

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
char far *pErrMsg;
...
wDasErr = K_GetErrMsg (hDev, nDASErr, &pErrMsg);
```

Purpose

Gets the current DAS shell version.

Prototype

C/C++
DASErr far pascal K_GetShellVer (WORD far **pVersion*);

Visual Basic for Windows
Declare Function K_GetShellVer Lib "DASSHELL.DLL"
(*pVersion* As Integer) As Integer

BASIC
DECLARE FUNCTION KGETSHELLVER% ALIAS "K_GetShellVer"
(SEG *pVersion* AS INTEGER)

Parameters

pVersion	A word value containing the major and minor version numbers of the DAS shell.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stores the current DAS Shell version in *pVersion*.

To obtain the major version number of the DAS shell, divide *pVersion* by 256. To obtain the minor version number of the DAS shell, perform a Boolean AND operation with *pVersion* and 255 (0FFh).

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wShellVer;
...
wDasErr = K_GetShellVer (&wShellVer);
printf ("Shell Ver %d.%d", wShellVer >> 8, wShellVer & 0xff);
```

K_GetShellVer (cont.)

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global wShellVer As Integer
...
wDasErr = K_GetShellVer (wShellVer)
ShellVer$ = LTRIM$ (STR$ (INT (wShellVer / 256))) + "." + :
   LTRIM$ (STR$ (wShellVer AND &HFF))
PRINT "Driver Ver: " + ShellVer$
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wShellVer AS INTEGER
...
wDasErr = KGETSHELLVER% (wShellVer)
ShellVer$ = LTRIM$ (STR$ (INT (wShellVer / 256))) + "." + :
   LTRIM$ (STR$ (wShellVer AND &HFF))
PRINT "Shell Ver: " + ShellVer$
```

K_GetVer

Purpose

Gets revision numbers.

Prototype

C/C++
DASErr far pascal K_GetVer (DWORD *hDev*, short far * *pSpecVer*, short far * *pDrvVer*);

Visual Basic for Windows
Declare Function K_GetVer Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *pSpecVer* As Integer, *pDrvVer* As Integer)
As Integer

BASIC
DECLARE FUNCTION KGETVER% ALIAS "K_GetVer"
(BYVAL *hDev* AS LONG, SEG *pSpecVer* AS INTEGER,
SEG *pDrvVer* AS INTEGER)

Parameters

hDev	Handle associated with the card.
pSpecVer	Revision number of the Keithley DAS Driver Specification to which the driver conforms.
pDrvVer	Driver version number.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DASCard-1000 Series card specified by *hDev*, this function stores the revision number of the Function Call Driver in *pDrvVer* and the revision number of the driver specification in *pSpecVer.*

The values stored in *pSpecVer* and *pDrvVer* are two-byte (16-bit) integers; the high byte of each contains the major revision level and the low byte of each contains the minor revision level. For example, if the driver version number is 2.10, the major revision level is 2 and the minor revision level is 10; therefore, the high byte of *pDrvVer* contains the value of **2** (512) and the low byte of *pDrvVer* contains the value of **10**; the value of both bytes is 522.

To obtain the major version number of the Function Call Driver, divide *pDrvVer* by 256; to obtain the minor version number of the Function Call Driver, perform a Boolean AND operation with *pDrvVer* and 255 (0FFh).

To obtain the major version number of the driver specification, divide *pSpecVer* by 256; to obtain the minor version number of the driver specification, perform a Boolean AND operation with *pSpecVer* and 255 (0FFh).

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
short nSpecVer, nDrvVer;
...
wDasErr = K_GetVer (hDev, &nSpecVer, &nDrvVer);
printf ("Driver Ver %d.%d", nDrvVer >> 8, nDrvVer & 0xff);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global nSpecVer As Integer
Global nDrvVer As Integer
...
wDasErr = K_GetVer (hDev, nSpecVer, nDrvVer)
DrvVer$ = LTRIM$ (STR$ (INT (nDrvVer / 256))) + "." + :
   LTRIM$ (STR$ (nDrvVer AND &HFF))
PRINT "Driver Ver: " + DrvVer$
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM nSpecVer AS INTEGER
DIM nDrvVer AS INTEGER
...
wDasErr = KGETVER% (hDev, nSpecVer, nDrvVer)
DrvVer$ = LTRIM$ (STR$ (INT (nDrvVer / 256))) + "." + :
   LTRIM$ (STR$ (nDrvVer AND &HFF))
PRINT "Driver Ver: " + DrvVer$
```

K_IntAlloc

Purpose

Allocates a buffer for a synchronous-mode or interrupt-mode operation.

Prototype

C/C++
DASErr far pascal K_IntAlloc (DWORD *hFrame*, DWORD *dwSamples*, void far * far **pBuf*, WORD far **pMem*);

Visual Basic for Windows
Declare Function K_IntAlloc Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *dwSamples* As Long, *pBuf* As Long, *pMem* As Integer) As Integer

BASIC
DECLARE FUNCTION KINTALLOC% ALIAS "K_IntAlloc" (BYVAL *hFrame* AS LONG, BYVAL *dwSamples* AS LONG, SEG *pBuf* AS LONG, SEG *pMem* AS INTEGER)

Parameters

hFrame Handle to the frame that defines the operation.

dwSamples Number of samples.
Valid values:

Synchronous mode	**1** to **32767**
Interrupt mode	**1** to **5000000**

pBuf Starting address of the allocated buffer.

pMem Handle associated with the allocated buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function allocates a buffer of the size specified by *dwSamples*, and stores the starting address of the buffer in *pBuf* and the handle of the buffer in *pMem*.

The data in the allocated buffer is stored as counts, either corrected or uncorrected. Refer to page 2-22 for information on correcting the data, if necessary. Refer to Appendix B for information on converting the corrected count value to voltage.

K_IntAlloc (cont.)

BASIC requires that you redistribute available memory before you dynamically allocate a buffer. Refer to page 3-25 for additional information.

See Also K_IntFree, K_SetBuf

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pBuf;   // Pointer to allocated buffer
WORD hMem;   // Memory Handle to buffer
...
wDasErr = K_IntAlloc (hAD, 1000, &pBuf, &hMem);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global pBuf As Long
Global hMem As Integer
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM pBuf AS LONG
DIM hMem AS INTEGER
...
wDasErr = KINTALLOC% (hAD, 1000, pBuf, hMem)
```

Purpose

Frees a buffer allocated for a synchronous-mode or interrupt-mode operation.

Prototype

C/C++
DASErr far pascal K_IntFree (WORD *hMem*);

Visual Basic for Windows
Declare Function K_IntFree Lib "DASSHELL.DLL"
(ByVal *hMem* As Integer) As Integer

BASIC
DECLARE FUNCTION KINTFREE% ALIAS "K_IntFree"
(BYVAL *hMem* AS INTEGER)

Parameters

hMem	Handle to buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function frees the buffer specified by *hMem*; the buffer was previously allocated dynamically using **K_IntAlloc**.

See Also

K_IntAlloc

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_IntFree (hMem);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_IntFree (hMem)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KINTFREE% (hMem)
```

K_IntStart

Purpose

Starts an interrupt-mode operation.

Prototype

C/C++
DASErr far pascal K_IntStart (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_IntStart Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KINTSTART% ALIAS "K_IntStart"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function starts the interrupt-mode operation defined by *hFrame*.

Refer to page 3-10 for a discussion of the programming tasks associated with interrupt-mode analog input operations.

See Also

K_IntStatus, K_IntStop

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_IntStart (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_IntStart (hAD)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KINTSTART% (hAD)
```

Purpose Gets status of an interrupt-mode operation.

Prototype

C/C++
DASErr far pascal K_IntStatus (DWORD *hFrame*, short far **pStatus*, DWORD far **pCount*);

Visual Basic for Windows
Declare Function K_IntStatus Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pStatus* As Integer, *pCount* As Long)
As Integer

BASIC
DECLARE FUNCTION KINTSTATUS% ALIAS "K_IntStatus"
(BYVAL *hFrame* AS LONG, SEG *pStatus* AS INTEGER,
SEG *pCount* AS LONG)

Parameters

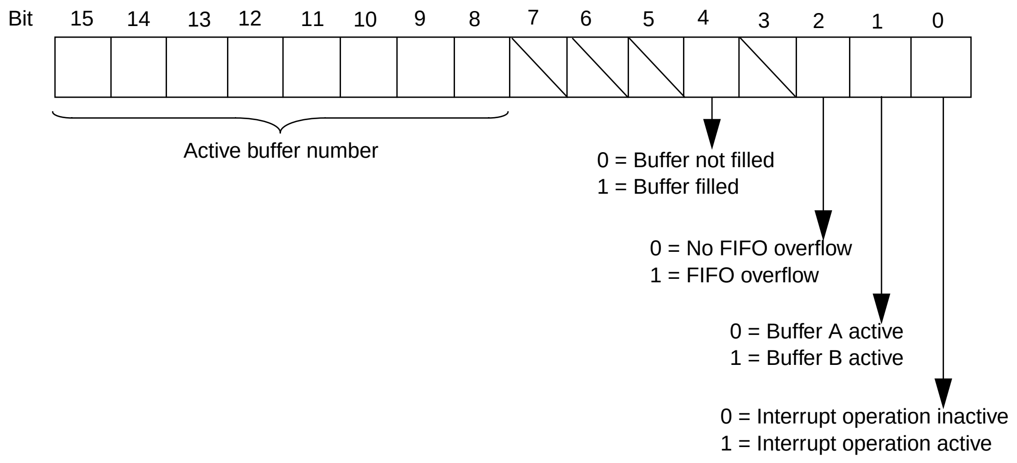
hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt-mode operation; see **Remarks** for value stored.
pCount	Current number of samples that were transferred for the active buffer.

Return Value Error/status code. Refer to Appendix A.

Remarks For the interrupt-mode operation defined by *hFrame*, this function stores the status in *pStatus* and the number of samples transferred into the current buffer in *pCount*.

K_IntStatus (cont.)

The value stored in *pStatus* depends on the settings in the Status word, as shown below:



The bits are described as follows:

- Bit 0: Indicates whether an interrupt-mode operation is in progress.
- Bit 1 is the Active Buffer toggle bit. If you are using multiple buffers, this bit toggles each time samples are stored in a new buffer. If you are using a single buffer, this bit is always 0.
- Bit 2: This bit indicates whether the oncard FIFO overflowed. The overflow event automatically stops all conversions.
- Bit 4 indicates whether the buffers used for an interrupt-mode operation running in continuous buffering mode have been filled. If this bit is set, all the buffers have been filled at least once.
- Bits 8-15 indicate which buffer in a multiple-buffer list is currently active. To determine the active buffer number, divide the value of the Status word by 256. The first buffer added to the list is Buffer 1, the second buffer added to the list is Buffer 2, and so on.
- Bits 3, 5, 6, and 7 are unassigned.

See Also K_IntStart, K_IntStop

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wStatus;
DWORD dwCount;
...
wDasErr = K_IntStatus (hAD, &wStatus, &dwCount);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global wStatus As Integer
Global dwCount As Long
...
wDasErr = K_IntStatus (hAD, wStatus, dwCount)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wStatus AS INTEGER
DIM dwCount AS LONG
...
wDasErr = KINTSTATUS% (hAD, wStatus, dwCount)
```

K_IntStop

Purpose	Stops an interrupt-mode operation.

Prototype

C/C++
DASErr far pascal K_IntStop (DWORD *hFrame*, short far **pStatus*, DWORD far **pCount*);

Visual Basic for Windows
Declare Function K_IntStop Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pStatus* As Integer, *pCount* As Long)
As Integer

BASIC
DECLARE FUNCTION KINTSTOP% ALIAS "K_IntStop"
(BYVAL *hFrame* AS LONG, SEG *pStatus* AS INTEGER,
SEG *pCount* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt-mode operation; see **Remarks** for **K_IntStatus** on page 4-56 for value stored.
pCount	Current number of samples that were transferred for the active buffer.

Return Value Error/status code. Refer to Appendix A.

Remarks

This function stops the interrupt-mode operation defined by *hFrame* and stores the status of the interrupt-mode operation in *pStatus* and the number of samples transferred into the current buffer in *pCount*.

If an interrupt-mode operation is not in progress, **K_IntStop** is ignored.

See Also K_IntStart, K_IntStatus

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wStatus;
DWORD dwCount;
...
wDasErr = K_IntStop (hAD, &wStatus, &dwCount);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global wStatus As Integer
Global dwCount As Long
...
wDasErr = K_IntStop (hAD, wStatus, dwCount)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM wStatus AS INTEGER
DIM dwCount AS LONG
...
wDasErr = KINTSTOP% (hAD, wStatus, dwCount)
```

K_MoveBufToArray

Purpose

Transfers data from a buffer allocated through **K_IntAlloc** to the program’s local array.

Prototype

C/C++
Not supported

Visual Basic for Windows
Declare Function K_MoveBufToArray Lib "DASSHELL.DLL" Alias "K_MoveDataBuf" (*pDest* As Integer, ByVal *pSource* As Long, ByVal *nCount* As Integer) As Integer

BASIC
DECLARE FUNCTION KMOVEBUFTOARRAY% ALIAS "K_MoveDataBuf" (SEG *pDest* AS INTEGER, BYVAL *pSource* AS LONG, BYVAL *nCount* AS INTEGER)

Parameters

pDest	Address of destination array.
pSource	Address of source buffer.
nCount	Number of samples to transfer. Value values: **0** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function transfers the number of samples specified by *nCount* from the buffer at address *pSource* to the array at address *pDest*.

If the buffer used to store acquired data for your program was allocated through **K_IntAlloc**, the buffer is not accessible to your program and you must use this function to move the data from the allocated buffer to the program’s local array. If the array used to store acquired data for your program was dimensioned locally within the program’s memory area, the array is accessible to your program and you do not have to use this function.

See Also

K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
...
wDasErr = K_MoveBufToArray (ADArray(0), pBuf, 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KINTALLOC% (hAD, 1000, pBuf, hMem)
...
wDasErr = KMOVEBUFTOARRAY% (ADArray(0), pBuf, 1000)
```

K_OpenDriver

Purpose

Initializes any Keithley DAS Function Call Driver.

Prototype

C/C++
DASErr far pascal K_OpenDriver (char far * *szDrvName*, char far * *szCfgName*, DWORD far * *pDrv*);

Visual Basic for Windows
Declare Function K_OpenDriver Lib "DASSHELL.DLL" (ByVal *szDrvName* As String, ByVal *szCfgName* As String, *pDrv* As Long) As Integer

BASIC
Not supported

Parameters

szDrvName	Driver name. Valid value: **"DAS1000"** (for DASCard-1000 Series cards)
szCfgName	Driver configuration file. Valid value: The name of a configuration file **0** if driver has already been opened.
pDrv	Handle associated with the driver.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DASCard-1000 Series Function Call Driver according to the information in the configuration file specified by *szCfgName*, and stores the driver handle in *pDrv*.

You can use this function to initialize the Function Call Driver associated with any Keithley MetraByte DAS card or board.

For DASCard-1000 Series cards, the string stored in *szDrvName* must be DAS1000.

You create a configuration file using the CFG1000.EXE utility. If *szCfgName* = 0, **K_OpenDriver** checks whether the driver has already been opened and linked to a configuration file and if it has, uses the current configuration; this is useful in the Windows environment. Refer to your *DASCard-1000 Series User's Guide* for more information about configuration files.

The value stored in *pDrv* is intended to be used exclusively as an argument to functions that require a driver handle. Your program should not modify the value stored in *pDrv*.

See Also DAS1000_DevOpen

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD hDrv;
...
wDasErr = K_OpenDriver ("DAS1000", "DAS1002.CFG", &hDrv);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
DIM hDrv As Long
...
wDasErr = K_OpenDriver ("DAS1000", "DAS1002.CFG", hDrv)
```

K_RestoreChnGAry

Purpose

Restores a converted channel-gain queue.

Prototype

C/C++
Not supported

Visual Basic for Windows
Declare Function K_RestoreChnGAry Lib "DASSHELL.DLL"
(*pArray* As Integer) As Integer

BASIC
DECLARE FUNCTION KRESTORECHNGARY% ALIAS
"K_RestoreChnGAry" (SEG *pArray* AS INTEGER)

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function restores the channel-gain queue at the address specified by *pArray* to its original format so that it can be used by your BASIC or Visual Basic for Windows program. The channel-gain queue was converted using **K_FormatChnGAry**.

See Also

K_FormatChnGAry, K_SetChnGAry

Usage

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Global ChanGainArray(16) As Integer ' Chan/Gain array
...
wDasErr = K_RestoreChnGAry (ChanGainArray(0))
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ChanGainArray(16) AS INTEGER   ' Chan/Gain array
...
wDasErr = KRESTORECHNGARY% (ChanGainArray(0))
```

Purpose

Sets the analog input channel configuration.

Prototype

C/C++
DASErr far pascal K_SetADConfig (DWORD *hDev*, WORD *nMode*);

Visual Basic for Windows
Declare Function K_SetADConfig Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, ByVal *nMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETADCONFIG% ALIAS "K_SetADConfig"
(BYVAL *hDev* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hDev	Handle associated with the card.
nMode	Analog input channel configuration. Valid values: **0** for Differential **1** for Single-ended

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DASCard-1000 Series card specified by *hDev*, this function specifies the analog input channel configuration in *nMode*.

See Also

K_GetADConfig

Usage

C/C++
```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
wDasErr = K_SetADConfig (hDev, 1);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetADConfig (hDev, 1)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETADCONFIG% (hDev, 1)
```

K_SetADMode

Purpose

Sets the analog input range type.

Prototype

C/C++
DASErr far pascal K_SetADMode (DWORD *hDev*, WORD *nMode*);

Visual Basic for Windows
Declare Function K_SetADMode Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, ByVal *nMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETADMODE% ALIAS "K_SetADMode"
(BYVAL *hDev* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hDev	Handle associated with the card.
nMode	Analog input range type. Valid values: **0** for Bipolar **1** for Unipolar

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DASCard-1000 Series card specified by *hDev*, this function specifies the analog input range type in *nMode*.

See Also

K_GetADMode

Usage

C/C++
```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
wDasErr = K_SetADMode (hDev, 0);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetADMode (hDev, 0)
```

K_SetADMode (cont.)

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETADMODE% (hDev, 0)
```

Purpose

Sets up an analog trigger.

Prototype

C/C++
DASErr far pascal K_SetADTrig (DWORD *hFrame,* short *nOpt*, short *nChan*, DWORD *dwLevel*);

Visual Basic for Windows
Declare Function K_SetADTrig Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *nOpt* As Integer, ByVal *nChan* As Integer, ByVal *dwLevel* As Long) As Integer

BASIC
DECLARE FUNCTION KSETADTRIG% ALIAS "K_SetADTrig" (BYVAL *hFrame* AS LONG, BYVAL *nOpt* AS INTEGER, BYVAL *nChan* AS INTEGER, BYVAL *dwLevel* AS LONG)

Parameters

hFrame Handle to the frame that defines the operation.

nOpt Analog trigger polarity and sensitivity.
Valid values:

Value	Polarity	Sensitivity
0	Positive	Edge-sensitive
1	Positive	Level-sensitive
2	Negative	Edge-sensitive
3	Negative	Level-sensitive

nChan Analog input channel used as trigger channel.
Valid values: **0** to **255**

dwLevel Level at which the trigger event occurs.
Valid values:

Bipolar	**–2048** to **2047**
Unipolar	**0** to **4095**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the channel used for an analog trigger in *nChan*, the level used for the analog trigger in *dwLevel*, and the trigger polarity and trigger sensitivity in *nOpt*.

You specify the value for *dwLevel* in counts. Refer to Appendix B for information on converting the actual voltage to a count.

The *nOpt* variable sets the value of the Trigger Polarity and Trigger Sensitivity elements; the *nChan* variable sets the value of the Trigger Channel element; the *dwLevel* variable sets the value of the Trigger Level element.

K_SetADTrig does not affect the operation defined by *hFrame* unless the Trigger Source element is set to External (by a call to **K_SetTrig**) before *hFrame* is used as a calling argument to **K_SyncStart** or **K_IntStart**.

See Also

K_SetTrig

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetTrig (hAD, 1);
wDasErr = K_SetADTrig (hAD, 0, 0, 2047);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetTrig (hAD, 1)
wDasErr = K_SetADTrig (hAD, 0, 0, 2047)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETTRIG% (hAD, 1)
wDasErr = KSETADTRIG% (hAD, 0, 0, 2047)
```

Purpose

Specifies the starting address of a previously allocated or dimensioned buffer and the number of samples to acquire.

Prototype

C/C++
DASErr far pascal K_SetBuf (DWORD *hFrame*, void far **pBuf*, DWORD *dwSamples*);

Visual Basic for Windows
Declare Function K_SetBuf Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *pBuf* As Long,
ByVal *dwSamples* As Long) As Integer

BASIC
DECLARE FUNCTION KSETBUF% Alias "K_SetBuf"
(BYVAL *hFrame* AS LONG, BYVAL *pBuf* AS LONG,
BYVAL *dwSize* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer.
dwSamples	Number of samples. Valid values:

Synchronous mode	**1** to **32767**
Interrupt mode	**1** to **5000000**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the starting address of a previously allocated buffer in *pBuf* and the number of samples to acquire in *dwSamples*.

For C/C++ application programs, use this function whether you dimensioned your array locally or allocated your buffer dynamically using **K_IntAlloc**.

For Visual Basic for Windows and BASIC, use this function only for buffers allocated dynamically using **K_IntAlloc**. For locally dimensioned arrays, use **K_SetBufI**.

Do not use this function if you are using multiple buffers. Use **K_BufListAdd** to specify the starting addresses of multiple buffers.

The *pBuf* variable sets the value of the Buffer element; the *dwSamples* variable sets the value of the Number of Samples element.

See Also K_IntAlloc, K_BufListAdd, K_SetBufI

Usage

C/C++

```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
void far *pBuf;   // Pointer to allocated buffer
...
wDasErr = K_IntAlloc (hAD, 1000, &pBuf, &hMem);
wDasErr = K_SetBuf (hAD, pBuf, 1000);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global pBuf As Long
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
wDasErr = K_SetBuf (hAD, pBuf, 1000)
```

BASIC

```
’ $INCLUDE: ’DASDECL.BI’
...
DIM pBuf AS LONG
...
wDasErr = KINTALLOC% (hAD, 1000, pBuf, hMem)
wDasErr = KSETBUF% (hAD, pBuf, 1000)
```

Purpose

Specifies the starting address of a locally dimensioned integer array and the number of samples to acquire.

Prototype

C/C++
Not supported

Visual Basic for Windows
Declare Function K_SetBufI Lib "DASSHELL.DLL" Alias "K_SetBuf" (ByVal *hFrame* As Long, *pBuf* As Integer, ByVal *dwSize* As Long) As Integer

BASIC
DECLARE FUNCTION KSETBUFI% Alias "K_SetBuf" (BYVAL *hFrame* AS LONG, SEG *pBuf* AS INTEGER, BYVAL *dwSize* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of the locally dimensioned integer array.
dwSize	Number of samples. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the starting address of a locally dimensioned integer array in *pBuf* and the number of samples to acquire in *dwSize*.

Do not use this function for C/C++; instead, use **K_SetBuf**.

For Visual Basic for Windows and BASIC, use this function only for locally dimensioned arrays. For buffers allocated dynamically using **K_IntAlloc**, use **K_SetBuf**.

Do not use this function if you are using multiple buffers. Instead, use **K_BufListAdd** to specify the starting addresses of multiple buffers.

The *pBuf* variable sets the value of the Buffer element; the *dwSize* variable sets the value of the Number of Samples element.

See Also

K_IntAlloc, K_BufListAdd, K_SetBuf

Usage

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
Dim ADData(2000) As Integer
...
wDasErr = K_SetBufI (hAD, ADData(0), 2000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ADData(2000) AS INTEGER
...
wDasErr = KSETBUFI% (hAD, ADData(0), 2000)
```

Purpose

Enables/disables automatic data correction.

Prototype

C/C++
DASErr far pascal K_SetCalMode (DWORD *hFrame*, short *nCalMode*);

Visual Basic for Windows
Declare Function K_SetCalMode Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nCalMode* As Integer) As Integer

BASIC
Not supported

Parameters

hFrame	Handle to the frame that defines the operation.
nCalMode	Automatic data correction mode. Valid values: **0** for Disabled **1** for Enabled

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies whether automatic data correction is enabled or disabled in *nCalMode*.

The *nCalMode* variable sets the Calibration Mode element.

Even though automatic data correction is disabled when *nCalMode* = 0, **K_GetADFrame** and **K_ClearFrame** enable automatic data correction (set *nCalMode* to 1); it is only necessary to call this function to disable automatic data correction.

Since automatic data correction is not supported in DOS, the Calibration Mode element is ignored when programming under DOS. It is not necessary to disable automatic data correction when programming under DOS.

See Also

K_CorrectData, K_GetCalData

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetCalMode (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetCalMode (hAD, 1)
```

Purpose

Specifies a single analog input channel.

Prototype

C/C++
DASErr far pascal K_SetChn (DWORD *hFrame*, short *nChan*);

Visual Basic for Windows
Declare Function K_SetChn Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nChan* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETCHN% ALIAS "K_SetChn"
(BYVAL *hFrame* AS LONG, BYVAL *nChan* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nChan	Channel on which to perform operation. Valid values: **0** to **255**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies a single analog input channel in *nChan*.

The *nChan* variable sets the Start Channel element and the Stop Channel element.

See Also

K_SetStartStopChn, K_SetStartStopG

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetChn (hAD, 2);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetChn (hAD, 2)
```

K_SetChn (cont.)

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETCHN% (hAD, 2)
```

Purpose

Specifies the starting address of a channel-gain queue.

Prototype

C/C++
DASErr far pascal K_SetChnGAry (DWORD *hFrame*, void far **pArray*);

Visual Basic for Windows
Declare Function K_SetChnGAry Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pArray* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETCHNGARY% ALIAS "K_SetChnGAry"
(BYVAL *hFrame* AS LONG, SEG *pArray* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the starting address of the channel-gain queue in *pArray.*

It is recommended that you use channel-gain queues in synchronous mode only.

The *pArray* variable sets the Channel-Gain Queue element.

Refer to page 2-14 for information on setting up a channel-gain queue.

If you created your channel-gain queue in BASIC or Visual Basic for Windows, you must use **K_FormatChnGAry** to convert the channel-gain queue before you specify the address with **K_SetChnGAry**.

See Also

K_FormatChnGAry, K_RestoreChnGAry

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
// DECLARE AND INITIALIZE CHAN/GAIN PAIRS
// (GainChanTable-TYPE IS DEFINED IN dasdecl.h)
GainChanTable ChanGainArray= {2,   // # of entries
   0, 0,    // chan 0, gain 1
   1, 1};   // chan 1, gain 2
...
wDasErr = K_SetChnGAry (hAD, &ChanGainArray);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global ChanGainArray(5) As Integer
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 0
ChanGainArray(3) = 1: ChanGainArray(4) = 1
wDasErr = K_FormatChnGAry (ChanGainArray(0))
wDasErr = K_SetChnGAry (hAD, ChanGainArray(0))
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ChanGainArray(5) AS INTEGER
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 0
ChanGainArray(3) = 1: ChanGainArray(4) = 1
wDasErr = KFORMATCHNGARY% (ChanGainArray(0))
wDasErr = KSETCHNGARY% (hAD, ChanGainArray(0))
```

Purpose

Specifies the pacer clock source.

Prototype

C/C++
DASErr far pascal K_SetClk (DWORD *hFrame*, short *nMode*);

Visual Basic for Windows
Declare Function K_SetClk Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETCLK% ALIAS "K_SetClk"
(BYVAL *hFrame* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nMode	Pacer clock source. Valid values: **0** for Internal **1** for External

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the pacer clock source in *nMode*.

The *nMode* variable sets the Clock Source element.

The internal clock source is the output of the counter/timer circuitry on the card; an external clock source is an external signal connected to the XCLK/PI0 line of the DASCard-1000 Series card.

Refer to page 2-15 for more information about pacer clock sources.

K_GetADFrame and **K_ClearFrame** specify internal as the default clock source.

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetClk (hAD, 1);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetClk (hAD, 1)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETCLK% (hAD, 1)
```

Purpose

Specifies the clock rate (number of clock ticks) used by the internal pacer clock.

Prototype

C/C++
DASErr far pascal K_SetClkRate (DWORD *hFrame*,
DWORD *dwDivisor*);

Visual Basic for Windows
Declare Function K_SetClkRate Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *dwDivisor* As Long) As Integer

BASIC
DECLARE FUNCTION KSETCLKRATE% ALIAS "K_SetClkRate"
(BYVAL *hFrame* AS LONG, BYVAL *dwDivisor* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
dwDivisor	Number of clock ticks between conversions. Valid values:

Card	Single Channel	Multiple Channels
DASCard-1001 DASCard-1002	**71** to **655350**	**294** to **655350**
DASCard-1003	**71** to **655350**	**71** to **655350**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the number of clock ticks between conversions in *dwDivisor.*

The hardware may not be able to convert the analog input channels at the exact rate determined by the number of clock ticks you specify. However, the driver calculates a rate that is as close as possible to the number you specify. To determine the actual number of clock ticks used by the internal pacer clock, use the **K_GetClkRate** function after you start the analog input operation.

The *dwDivisor* variable sets the Pacer Clock Rate element.

Refer to page 2-15 for more information about the internal pacer clock.

See Also K_GetClkRate

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD dwClkDiv;
...
dwClkDiv = 10000000 / 30000;
wDasErr = K_SetClkRate (hAD, dwClkDiv);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global dwClkDiv As Long
...
dwClkDiv = 10000000 / 30000
wDasErr = K_SetClkRate (hAD, dwClkDiv)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwClkDiv AS LONG
...
dwClkDiv = 10000000 / 30000
wDasErr = KSETCLKRATE% (hAD, dwClkDiv)
```

Purpose

Specifies continuous buffering mode.

Prototype

C/C++
DASErr far pascal K_SetContRun (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_SetContRun Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KSETCONTRUN% ALIAS "K_SetContRun"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function sets the buffering mode to continuous mode and sets the Buffering Mode element in the frame accordingly.

K_GetADFrame and **K_ClearFrame** specify single-cycle as the default buffering mode.

Refer to page 2-17 for a description of buffering modes.

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetContRun (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetContRun (hAD)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETCONTRUN% (hAD)
```

Purpose

Sets up a digital trigger.

Prototype

C/C++
DASErr far pascal K_SetDITrig (DWORD *hFrame*, short *nOpt*,
short *nChan*, DWORD *nPattern*);

Visual Basic for Windows
Declare Function K_SetDITrig Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nOpt* As Integer,
ByVal *nChan* As Integer, ByVal *nPattern* As Long) As Integer

BASIC
DECLARE FUNCTION KSETDITRIG% ALIAS "K_SetDITrig"
(BYVAL *hFrame* AS LONG, BYVAL *nOpt* AS INTEGER,
BYVAL *nChan* AS INTEGER, BYVAL *nPattern* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
nOpt	Trigger polarity and sensitivity.
nChan	Digital input channel.
nPattern	Trigger pattern.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies that you are using an external digital trigger.

It is not necessary to change the default values of the *nOpt*, *nChan*, and *nPattern* parameters. Since DASCard-1000 Series cards support a negative-edge digital trigger only, the value of *nOpt* is ignored. Since the external digital trigger must be connected to XTRIG/PI1 line of the DASCard-1000 Series card, the value of *nChan* is meaningless. Since the DASCard-1000 Series Function Call Driver does not currently support digital pattern triggering, the value of *nPattern* is meaningless.

K_SetDITrig does not affect the operation defined by *hFrame* unless the Trigger Source element is set to External (by a call to **K_SetTrig**) before *hFrame* is used as a calling argument to **K_SyncStart** or **K_IntStart**.

See Also

K_SetTrig

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetTrig (hAD, 1);
wDasErr = K_SetDITrig (hAD, 0, 0, 0);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetTrig (hAD, 1)
wDasErr = K_SetDITrig (hAD, 0, 0, 0)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETTRIG% (hAD, 1)
wDasErr = KSETDITRIG% (hAD, 0, 0, 0)
```

Purpose

Sets the gain.

Prototype

C/C++
DASErr far pascal K_SetG (DWORD *hFrame*, short *nGain*);

Visual Basic for Windows
Declare Function K_SetG Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nGain* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETG% ALIAS "K_SetG"
(BYVAL *hFrame* AS LONG, BYVAL *nGain* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nGain	Gain code. Valid values:

Card	**Gain**	**Gain Code**
DASCard-1001	1	0
	10	1
	100	2
	500	3
DASCard-1002	1	0
	2	1
	4	2
	8	3
DASCard-1003	1	0

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the gain code for a single channel or for a group of consecutive channels in *nGain*.

The *nGain* variable sets the Gain element.

K_GetADFrame and **K_ClearFrame** specify a gain of 1 (gain code 0) as the default gain.

See Also

K_SetStartStopG

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetG (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetG (hAD, 1)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETG% (hAD, 1)
```

K_SetStartStopChn

Purpose

Specifies the first and last channels in a group of consecutive channels.

Prototype

C/C++
DASErr far pascal K_SetStartStopChn (DWORD *hFrame*, short *nStart*, short *nStop*);

Visual Basic for Windows
Declare Function K_SetStartStopChn Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nStart* As Integer,
ByVal *nStop* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETSTARTSTOPCHN% ALIAS
"K_SetStartStopChn" (BYVAL *hFrame* AS LONG,
BYVAL *nStart* AS INTEGER, BYVAL *nStop* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels. Valid values: **0** to **255**
nStop	Last channel in a group of consecutive channels. Valid values: **0** to **255**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the first channel in a group of consecutive channels in *nStart* and the last channel in the group of consecutive channels in *nStop*.

The *nStart* variable sets the value of the Start Channel element; the *nStop* variable sets the value of the Stop Channel element.

K_GetADFrame and **K_ClearFrame** set the Start Channel and Stop Channel elements to 0.

See Also

K_SetStartStopG

K_SetStartStopChn (cont.)

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetStartStopChn (hAD, 0, 7);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetStartStopChn (hAD, 0, 7)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETSTARTSTOPCHN% (hAD, 0, 7)
```

Purpose

Specifies the first and last channels in a group of consecutive channels and sets the gain for all channels in the group.

Prototype

C/C++
DASErr far pascal K_SetStartStopG (DWORD *hFrame*, short *nStart*, short *nStop*, short *nGain*);

Visual Basic for Windows
Declare Function K_SetStartStopG Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *nStart* As Integer, ByVal *nStop* As Integer, ByVal *nGain* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETSTARTSTOPG% ALIAS "K_SetStartStopG" (BYVAL *hFrame* AS LONG, BYVAL *nStart* AS INTEGER, BYVAL *nStop* AS INTEGER, BYVAL *nGain* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels. Valid values: **0** to **255**
nStop	Last channel in a group of consecutive channels. Valid values: **0** to **255**

nGain

Gain code.
Valid values:

Card	Gain	Gain Code
DASCard-1001	1	0
	10	1
	100	2
	500	3
DASCard-1002	1	0
	2	1
	4	2
	8	3
DASCard-1003	1	0

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the first channel in a group of consecutive channels in *nStart*, the last channel in a group of consecutive channels in *nStop*, and the gain code for all channels in the group in *nGain*.

The *nStart* variable sets the value of the Start Channel element; the *nStop* variable sets the value of the Stop Channel element; the *nGain* variable sets the value of the Gain element.

K_GetADFrame and **K_ClearFrame** set the Start Channel, Stop Channel, and Gain elements to 0.

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetStartStopG (hAD, 0, 7, 0);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetStartStopG (hAD, 0, 7, 0)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETSTARTSTOPG% (hAD, 0, 7, 0)
```

K_SetTrig

Purpose

Specifies the trigger source.

Prototype

C/C++
DASErr far pascal K_SetTrig (DWORD *hFrame*, short *nMode*);

Visual Basic for Windows
Declare Function K_SetTrig Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nMode* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETTRIG% ALIAS "K_SetTrig"
(BYVAL *hFrame* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nMode	Trigger source. Valid values: **0** for Internal trigger **1** for External trigger

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the trigger source in *nMode*.

An internal trigger is a software trigger; the trigger event occurs when the operation is started. An external trigger is either an analog trigger or a digital trigger. Refer to page 2-18 for more information about internal and external trigger sources.

If *nMode* = **1**, an external digital trigger (positive edge) is assumed. Use **K_SetDITrig** to specify a negative edge for an external digital trigger (DASCard-1000 Series cards support a negative-edge digital trigger only). Use **K_SetADTrig** to specify the conditions for an external analog trigger.

K_GetADFrame and **K_ClearFrame** set the trigger source to internal.

See Also

K_SetADTrig, K_SetDITrig

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetTrig (hAD, 1);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetTrig (hAD, 1)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETTRIG% (hAD, 1)
```

K_SetTrigHyst

Purpose

Specifies the hysteresis value.

Prototype

C/C++
DASErr far pascal K_SetTrigHyst (DWORD *hFrame*, short *nHyst*);

Visual Basic for Windows
Declare Function K_SetTrigHyst Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nHyst* As Integer) As Integer

BASIC
DECLARE FUNCTION KSETTRIGHYST% ALIAS "K_SetTrigHyst"
(BYVAL *hFrame* AS LONG, BYVAL *nHyst* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nHyst	Hysteresis value. Valid values:

Bipolar	**0** to **2047**
Unipolar	**0** to **4095**

Return Value
Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the hysteresis value used for an analog trigger in *nHyst*.

You specify the hysteresis value in counts; refer to Appendix B for information on converting the hysteresis voltage to a count.

The *nHyst* variable sets the Trigger Hysteresis element.

K_SetTrigHyst does not affect the operation defined by *hFrame* unless the Trigger Source element is set to External (by a call to **K_SetTrig**) and the trigger conditions are set to either postive or negative edge (by a call to **K_SetADTrig**) before *hFrame* is used as a calling argument to **K_SyncStart** or **K_IntStart**.

Refer to page 2-18 for more information about analog triggers.

See Also K_SetTrig, K_SetADTrig

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetTrig (hAD, 1);
wDasErr = K_SetADTrig (hAD, 0, 0, 2047);
wDasErr = K_SetTrigHyst (hAD, 50);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SetTrig (hAD, 1)
wDasErr = K_SetADTrig (hAD, 0, 0, 2047)
wDasErr = K_SetTrigHyst (hAD, 50)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETTRIG% (hAD, 1)
wDasErr = KSETADTRIG% (hAD, 0, 0, 2047)
wDasErr = KSETTRIGHYST% (hAD, 50)
```

K_SyncStart

Purpose

Starts a synchronous-mode operation.

Prototype

C/C++
DASErr far pascal K_SyncStart (DWORD *hFrame*);

Visual Basic for Windows
Declare Function K_SyncStart Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long) As Integer

BASIC
DECLARE FUNCTION KSYNCSTART% ALIAS "K_SyncStart"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function starts the synchronous-mode operation defined by *hFrame*.

Refer to page 3-8 for information on the programming tasks associated with synchronous-mode analog input operations.

See Also

K_IntStart

Usage

C/C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SyncStart (hAD);
```

Visual Basic for Windows
(Add DASDECL.BAS to your project)
```
...
wDasErr = K_SyncStart (hAD)
```

BASIC
```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSYNCSTART% (hAD)
```

A

Error/Status Codes

Table A-1 lists the error/status codes that are returned by the DASCard-1000 Series Function Call Driver, possible causes for error conditions, and possible solutions for resolving error conditions.

If you cannot resolve an error condition, contact the Keithley MetraByte Applications Engineering Department.

Table A-1. Error/Status Codes

Error Code			
Hex	**Decimal**	**Cause**	**Solution**
0	0	No error has been detected.	Status only; no action is necessary.
6000	24576	**Error in configuration file:** The configuration file you specified in the driver initialization function is corrupt, does not exist, or contains one or more undefined keywords.	Check that the file exists at the specified path. Check for illegal keywords in file; you can avoid illegal keywords by using the configuration utility to create and modify configuration files.
6001	24577	**Illegal base address in configuration file:** The base I/O address of the card/board in the configuration file is illegal and/or does not match the base address switches on the card/board.	Use the configuration utility to change the base I/O address of the card/board to one that matches the base address switches on the card/board, if applicable.
6002	24578	**Illegal IRQ level in configuration file:** The interrupt level in the configuration file is illegal.	Use the configuration utility to change the interrupt level to a legal one for your card/board. Refer to the user's guide for legal interrupt levels.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	Decimal	Cause	Solution
6003	24579	**Illegal DMA channel in configuration file:** The DMA channel in the configuration file is illegal.	Use the configuration utility to change the DMA channel to a legal one for your card/board. Refer to the user's guide for legal DMA channels.
6005	24581	**Illegal channel number:** The specified channel number is illegal for the card/board and/or for the range type (unipolar or bipolar).	Specify a legal channel number. Refer to the user's guide or to the description of **K_SetStartStopChn** in Chapter 4 for legal channel numbers.
6006	24582	**Illegal gain code:** The specified analog I/O channel gain code is illegal for this card/board.	Specify a legal gain code. Refer to the user's guide or to the description of **K_SetG** in Chapter 4 for a list of legal gain codes.
6007	24583	**Illegal DMA address:** An FCD function specified a buffer address that is not suitable for a DMA operation for the number of samples required.	Use the **K_DMAAlloc** function to allocate dynamic buffers for DMA operations. In Windows, make sure that the Keithley Memory Manager is installed; refer to the user's guide for information.
6008	24584	**Illegal number in configuration file:** The configuration file contains one or more numeric values that are illegal.	Use the configuration utility to check and then change the configuration file.
600A	24586	**Configuration file not found:** The driver cannot find the configuration file specified as an argument to the driver initialization function.	Check that the file exists at the specified path. Check that the file name is spelled correctly in the driver initialization function parameter list.
600B	24587	**Error returning DMA buffer:** DOS returned an error in INT 21H function 49H during the execution of **K_DMAFree** .	Check that the memory handle passed as an argument to **K_DMAFree** was previously obtained using **K_DMAAlloc** .
600C	24588	**Error returning interrupt buffer:** The memory handle specified in **K_IntFree** is invalid.	Check the memory handle stored by **K_IntAlloc** and make sure that it was not modified.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	Decimal	Cause	Solution
600D	24589	**Illegal frame handle:** The specified frame handle is not valid for this operation.	Check that the frame handle exists. Check that you are using the appropriate frame handle.
600E	24590	**No more frame handles:** No frames are left in the pool of available frames.	Use **K_FreeFrame** to free a frame that the application is no longer using.
600F	24591	**Requested buffer size too large:** The requested buffer cannot be dynamically allocated because of its size.	Specify a smaller buffer size; refer to the description of **K_IntAlloc** in Chapter 4 for the legal range. If in Windows Enhanced mode with the Keithley Memory Manager (VDMAD.386) installed, use KMMSETUP.EXE to increase the reserved buffer heap size.
6010	24592	**Cannot allocate interrupt buffer:** (Windows-based languages only) **K_IntAlloc** failed because there was not enough available DOS memory.	Remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
6012	24594	**Interrupt buffer deallocation error:** (Windows-based languages only) An error occurred when **K_IntFree** attempted to free a memory handle.	Make sure that the memory handle passed as an argument to **K_IntFree** was previously obtained using **K_IntAlloc** .
6015	24597	**DMA Buffer too large:** The number of samples specified in **K_DMAAlloc** is too large.	Refer to the description of **K_DMAAlloc** in Chapter 4 for the buffer size range.
6016	24598	**VDS - Region not contiguous:** An error occurred while using Windows Virtual DMA Services. You tried to use **K_DMAAlloc** in Windows Enhanced mode and the Keithley Memory Manager (VDMAD.386) was not installed.	Refer to the user's guide for information on how to install and set up the Keithley Memory Manager (VDMAD.386).

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	**Decimal**	**Cause**	**Solution**
6017	24599	**VDS - DMA wraparound:** See error 6016.	See error 6016.
6018	24600	**VDS - Unable to lock region:** See error 6016.	See error 6016.
6019	24601	**VDS - No buffer available:** See error 6016.	See error 6016.
601A	24602	**VDS - Region too large:** See error 6016.	See error 6016.
601B	24603	**VDS - Buffer in use:** See error 6016.	See error 6016.
601C	24604	**VDS - Illegal region:** See error 6016.	See error 6016.
601D	24605	**VDS - Region not locked:** See error 6016.	See error 6016.
601E	24606	**VDS - Illegal page:** See error 6016.	See error 6016.
601F	24607	**VDS - Illegal buffer:** See error 6016.	See error 6016.
6020	24608	**VDS - Copy out of range:** See error 6016.	See error 6016.
6021	24609	**VDS - Illegal DMA channel:** See error 6016.	See error 6016.
6022	24610	**VDS - Count overflow:** See error 6016.	See error 6016.
6023	24611	**VDS - Count underflow:** See error 6016.	See error 6016.
6024	24612	**VDS - Function not supported:** See error 6016.	See error 6016.
6025	24613	**Illegal OBM mode:** The mode number specified in **K_SetOBMMode** is illegal.	Refer to the description of **K_SetOBMMode** in Chapter 4 for legal mode values.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	**Decimal**	**Cause**	**Solution**
6026	24614	**Illegal DMA structure:** An error occurred during the execution of **K_DMAFree**.	Try using **K_DMAFree** again. If the error continues, contact the Keithley MetraByte Applications Engineering Department.
6027	24615	**DMA allocation error:** See error 6026.	See error 6026.
6028	24616	**NULL DMA handle:** See error 6026.	See error 6026.
6029	24617	**DMA unlock error:** See error 6026.	See error 6026.
602A	24618	**DMA free error:** See error 6026.	See error 6026.
602B	24619	**Not enough memory to accommodate request:** The number of samples you requested in the Keithley Memory Manager is greater than the largest contiguous block available in the reserved heap.	Specify a smaller number of samples. Free a previously allocated buffer. Use the KMMSETUP utility to expand the reserved heap.
602C	24620	**Requested buffer size exceeds maximum:** The number of samples you requested from the Keithley Memory Manager is greater than the allowed maximum.	Specify a value within the legal range when calling **K_DMAAlloc** or **K_IntAlloc** in Windows Enhanced mode. Refer to Chapter 4 for legal values.
602D	24621	**Illegal device handle:** A bad device handle was passed to a function such as **K_GetADFrame**. The handle used was not initialized through a call to **K_GetDevHandle** or **DAS1000_GetDevHandle**, or it was corrupted by your program.	Check the device handle value.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
602E	24622	**Illegal Setup option:** An illegal option was specified to a function that accepts a user option, such as **K_SetDITrig**.	Check the option value passed to the function where the error occurred.
6030	24624	**DMA word-page wrap:** During **K_DMAAlloc**, a DMA word-page wrap condition occurred and the allocation attempt failed since there is not enough free memory to accommodate the allocation request.	Reduce the number of samples and retry. If in Windows Enhanced mode, install and configure VDMAD.386.
6031	24625	**Illegal memory handle:** A bad memory handle was passed to **K_IntFree** or **K_DMAFree**. The handle used was not initialized through a call to **K_IntAlloc** or **K_DMAAlloc**, or it was corrupted by your program.	Restart your program and monitor the memory handle value(s).
6032	24626	**Out of memory handles:** An attempt to allocate a memory block using **K_IntAlloc** or **K_DMAAlloc** failed because the maximum number of handles has already been assigned.	Use **K_IntFree** or **K_DMAFree** to free previously allocated memory blocks before allocating again.
6034	24628	**Memory corrupted:** Int 21H function 48H, used to allocate a memory block from the DOS far heap, returned the DOS error 7; this means that memory is corrupted. It is likely that you stored data (through a DMA-mode or interrupt-mode operation) into an illegal area of DOS memory.	Recheck the parameters set by **K_DMAAlloc** and **K_SetDMABuf**. If a fatal system error, restart your computer.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	Decimal	Cause	Solution
6035	24629	**Driver in use:** You attempted to initialize a driver that was already initialized by a call to **K_OpenDriver**. (This can occur since, under Windows, it is possible to open the same driver from multiple programs that are running simultaneously.)	To continue using the driver with the same configuration, pass a null string as the second argument to **K_OpenDriver**. To use the driver with a different configuration, close any application programs currently accessing the driver, and then open the driver again (using **K_OpenDriver**).
6036	24630	**Illegal driver handle:** The specified driver handle is not valid.	Someone may have closed the driver; if so, use **K_OpenDriver** to reopen the driver with the desired driver handle. Try again using another driver handle.
6037	24631	**Driver not found:** The specified driver cannot be found.	Check your link statement to make sure the specified driver is included. Make sure that the device name string is entered correctly in **K_OpenDriver**.
6038	24632	**Invalid source pointer:** (Windows-based languages only) The pointer to the source buffer that you passed as an argument to **K_MoveBufToArray** is invalid for the specified count. (The source pointer, when added to the number of samples, exceeds the programmed addressing range of that pointer.)	Check the pointer to the source buffer and the number of samples to transfer that you specified in **K_MoveBufToArray**.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
6039	24633	**Invalid destination pointer:** (Windows-based languages only) The pointer to the destination buffer (local array) that you passed as an argument to **K_MoveBufToArray** is invalid for the specified count. (The destination pointer, when added to the number of samples, exceeds the dimension of the local array.)	Check the dimension of the local array and the number of samples to transfer that you specified in **K_MoveBufToArray** .
603A	24634	**Illegal setup value:** An illegal value was passed to the function in which the error occurred.	Check the legal ranges of all parameters passed to this function.
603B	24635	**Error freeing buffer selector: K_DMAFree** or **K_IntFree** failed because one or more of the selectors that reference the memory buffer could not be freed.	Check that the memory buffer being freed was previously obtained through **K_DMAAlloc** or **K_IntAlloc.**
603C	24636	**Error allocating buffer selector: K_DMAAlloc** or **K_IntAlloc** failed because a selector could not be allocated from Window's Local Descriptor Table.	Close all applications and restart Windows. If the error continues, contact the Keithley MetraByte Applications Engineering Department.
603D	24637	**Error allocating memory buffer: K_DMAAlloc** or **K_IntAlloc** failed because a necessary internal buffer could not be allocated to complete the operation.	Close all applications and restart Windows. If the error continues, contact the Keithley MetraByte Applications Engineering Department.
7000	28672	**No board name** : The driver initialization function did not find a card name in the specified configuration file.	Specify a legal card name: DASCard-1001, DASCard-1002, DASCard-1003.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	Decimal	Cause	Solution
7001	28673	**Illegal board name**: The card name in the specified configuration file is illegal.	Specify a legal card name: DASCard-1001, DASCard-1002, DASCard-1003.
7002	28674	**Illegal board number**: The driver initialization function found an illegal card number in the specified configuration file.	Specify a legal card number: 0 or 1.
7006	28678	**Illegal error number**: The 7000 Series error number passed to **K_GetErrMsg** was invalid.	Check the 7000 Series error number passed to **K_GetErrMsg**.
7007	28679	**Illegal ADC channel mode**: The driver initialization function found an illegal input range type in the specified configuration file.	Specify a legal input range type: bipolar or unipolar.
7008	28680	**Illegal ADC channel configuration**: The driver initialization function found an illegal input configuration in the specified configuration file.	Specify a legal input configuration: single-ended or differential.
7009	28681	**Illegal number of EXP-1600 boards**: The driver initialization function found an illegal number of EXP-1600 expansion accessories in the specified configuration file.	Specify a legal number of EXP-1600 accessories: 1 through 16.
700A	28682	**Bad EXP-1600 number specified**: The driver initialization function found an illegal number assigned to one of the EXP-1600 expansion accessories in the specified configuration file.	Specify a legal number for each EXP-1600 expansion accessory: 0 to 15

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	Decimal	Cause	Solution
700B	28683	**Bad EXP-1600 gain specified**: The driver initialization function found an illegal gain assigned to one of the EXP-1600 expansion accessories in the specified configuration file.	Specify a legal gain value for each EXP-1600 expansion accessory: 0.5, 1, 5, 10, 50, 100, 250, 500
700E	28686	**KMENABLE is not loaded**: You used **K_GetDevHandle**, **DAS1000_GetDevHandle**, or **K_DASDevInit** to initialize a card, but the Enabler was not yet loaded.	Return to DOS and load the Enabler. Refer to Chapter 3 of the *DASCard-1000 Series User's Guide* for information.
700F	28687	**No card inserted in socket(s)**: You used **K_GetDevHandle**, **DAS1000_GetDevHandle**, or **K_DASDevInit** to initialize a card, but no cards were installed.	Make sure that your DASCard-1000 Series card is installed properly. Refer to Chapter 3 of the *DASCard-1000 Series User's Guide* for information. On some computers, you may have to turn computer power OFF and then ON after you install a card in order for the computer to recognize the card.
7010	28688	**Card is not enabled**: You used **K_GetDevHandle**, **DAS1000_GetDevHandle**, or **K_DASDevInit** to initialize a card, but the card was not enabled.	Run the KMINFOW.EXE utility to check card information. Make sure that PCMCIA card and socket services, the Enabler, and the card are installed. Make sure that the specified base address, interrupt level, and memory segment address are not being used by another device in your system. Make sure that no errors occurred during the allocation of system resources.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	**Decimal**	**Cause**	**Solution**
7011	28689	**Card in socket differs from config file**: You used **K_GetDevHandle**, **DAS1000_GetDevHandle**, or **K_DASDevInit** to initialize a card, but the card is not the same as the card specified in the associated configuration file.	Check the configuration file specified in **K_OpenDriver** or **DAS1000_DevOpen** and make sure that the card name specified matches the card you are trying to initialize.
7012	28690	**Illegal DAS specification revision number:** The revision of the driver you are using does not match the revision of the Keithley DAS Driver Specification.	Make sure that you are using the appropriate driver.
7013	28691	**Resource busy:** The application program attempted to start an operation while a similar operation was in progress.	An attempt was made to execute interrupt-mode operations simultaneously. Only one of these operations is allowed at one time. Use **K_IntStop** to stop the in-progress operation before initiating the second operation.
7014	28692	**Clock rate specified exceeds maximum:** The number of clock ticks you specified in **K_SetClkRate** is too low.	Enter a valid number of clock ticks: 294 to 655,350 (for DASCard-1001 or DASCard-1002); 71 to 655,350 (for DASCard-1003).
7015	28693	**Buffer size exceeds 32K for synchronous mode**: The number of samples you specified for the buffer or array used for a synchronous-mode operation is too large.	For a synchronous-mode operation, make sure that the number of samples you specify in **K_IntAlloc** or **K_SetBuf** does not exceed 32,767.
7016	28694	**Cannot acquire cyclic data in sync mode**: You specified continuous buffering mode for a synchronous-mode operation.	For a synchronous-mode operation, either use the default setting of the frame (single-cycle) or use **K_ClrContRun** to return the buffering mode to single-cycle.

Table A-1. Error/Status Codes (cont.)

Error Code			
Hex	**Decimal**	**Cause**	**Solution**
7017	28695	**Data has already been corrected by driver**: You called **K_CorrectData** while automatic data correction was enabled.	If you want to use **K_CorrectData** to correct your data, make sure that you disable automatic data correction in **K_SetCalMode**.
7018	28696	**Cannot set to differential with EXP-1600**: You used **K_SetADConfig** to set the input channel configuration to differential when either an EXP-1600 expansion accessory was connected to your card or the current configuration file indicates that an EXP-1600 is being used.	Make sure that you are using the appropriate configuration file. If you are not using any EXP-1600 expansion accessories, make sure that none are connected to your card and that the configuration file indicates zero expansion accessories. Use **K_SetADConfig** to set the input channel configuration to single-ended.
7019	28697	**Bad A/D clock pulse**: Either the external clock rate is too fast or a noisy external clock signal is causing unwanted pulses.	Adjust the external clock source to slow down the rate at which the card acquires data. Make sure that your external clock signal is stable.
8001	32769	**Function not supported:** You have attempted to use a function not supported by the Function Call Driver.	Contact the Keithley MetraByte Applications Engineering Department.
8003	32771	**Illegal board number:** An illegal card/board number was specified in the card/board initialization function.	Refer to the description of **K_GetDevHandle** or **DASxxxx_GetDevHandle** in Chapter 4 for legal card/board numbers.
8004	32772	**Illegal error number:** The error message number specified in **K_GetErrMsg** is invalid.	The error number must be one the error numbers listed in this appendix.
8005	32773	**Board not found at configured address:** The card/board initialization function does not detect the presence of a card/board.	If applicable, make sure that the base address setting of the switches on the card/board matches the base address setting in the configuration file.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
8006	32774	**A/D not initialized:** You attempted to start a frame-based analog input operation without the A/D frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8007	32775	**D/A not initialized:** You attempted to start a frame-based analog output operation without the D/A frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8008	32776	**Digital input not initialized:** You attempted to start a frame-based digital input operation without the DI frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8009	32777	**Digital output not initialized:** You attempted to start a frame-based digital output operation without the DO frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
800B	32779	**Conversion overrun:** The conversion rate is too fast or the time required to service an interrupt is too long.	Adjust the clock source to slow down the rate at which the card/board acquires data. Remove other application programs that are running and using computer resources. Try performing the operation in synchronous mode instead of interrupt mode.
8016	32790	**Interrupt overrun**: The card/board communicated a hardware event to the software by generating a hardware interrupt, but the software was still servicing a previous interrupt. This is usually caused by a pacer clock rate that is too fast.	Check the maximum throughput rate for your computer's programming environment and use **K_SetClkRate** to specify an appropriate rate.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	**Decimal**		
801A	32794	**Interrupts already active:** You have attempted to start an operation whose interrupt level is being used by another system resource.	Use **K_IntStop** to stop the first operation before starting the second operation.
801B	32795	**DMA already active**: You attempted to start a DMA-mode operation using a DMA channel that is currently used by another active operation.	Use **K_DMAStop** to stop the first operation before starting the second operation.
8020	32800	**FIFO Overflow event detected:** During data acquisition, the temporary oncard/onboard data storage (FIFO) overflowed.	The conversion rate is too fast for your computer's programming environment; use **K_SetClkRate** to reduce the conversion rate. If you are using DMA-mode and your card/board supports dual-DMA, use the configuration utility to reconfigure your card/board to use dual-DMA.
8021	32801	**Illegal clock sync mode:** The two operations you are trying to synchronize cannot be synchronized on your card/board.	Check the synchronizing clock source that you specified in **K_SetSync**.
FFFF	65535	**User aborted operation:** You pressed **Ctrl** +**Break** during a synchronous-mode operation or while waiting for an analog trigger event to occur.	Start the operation again, if desired.

B

Data Formats

The DASCard-1000 Series Function Call Driver can read and write counts only. When writing a value (as in **K_SetADTrig**), you must convert the voltage value to a count; when reading a value (as in **K_ADRead**), you may want to convert the count to a more meaningful voltage value.

To ensure valid results when converting a count value to voltage, the count value must be the corrected count value. If you have disabled automatic data correction and you are not using **K_CorrectData** to correct the data, the application program must correct the data using the calibration factors and the appropriate formula. For more information on correcting data, refer to page 2-22.

This appendix contains instructions for converting corrected counts to voltage and for converting voltage to counts.

Note: The DASCard-1000 Series Function Call Driver provides the **K_GetADMode** function, which gets the analog input range type (bipolar or unipolar). You may find this function useful when converting values.

Converting Corrected Counts to Voltage

You may want to convert corrected counts to voltage when reading an analog input value. To convert a corrected count to an analog input voltage, use the following equation, where *count* is the corrected count value and *span* is the span of the analog input range. Refer to Table B-1 for a list of span values.

$$\text{Voltage} = \frac{\text{count} \times \text{span}}{4096}$$

Table B-1. Span Values for A/D Conversion Equations

Card	A/D Mode	Gain	Input Range	Span
DASCard-1001	Unipolar	1	0 to 5 V	5 V
		10	0 V to 0.5 V	0.5 V
		100	0 to 50 mV	0.05 V
		1000	0 to 5 mV	0.005 V
	Bipolar	1	–5 V to +5 V	10 V
		10	–0.5 V to 0.5 V	1.0 V
		100	–50 mV to +50 mV	0.10 V
		1000	–5 mV to +5 mV	0.01 V

Table B-1. Span Values for A/D Conversion Equations (cont.)

Card	A/D Mode	Gain	Input Range	Span
DASCard-1002	Unipolar	1	0 to 5 V	5 V
		2	0 V to 2.5 V	2.5 V
		4	0 to 1.25 V	1.25 V
		8	0 V to 0.625 V	0 .625 V
	Bipolar	1	–5.0 V to +5.0 V	10 V
		2	–2.5 V to +2.5 V	5 V
		4	–1.25 V to +1.25 V	2.5 V
		8	–0.625 V to +0.625 V	1.25 V
DASCard-1003	Unipolar	1	0 to 5 V	5 V
	Bipolar	1	–5.0 V to +5.0 V	10 V

For example, assume that you want to read analog input data from a channel on a DASCard-1001 card configured for the unipolar input range and a gain of 1. The count value is 3072. The voltage is determined as follows:

$$\frac{3072 \times 5}{4096} = 3.75\text{ V}$$

As another example, assume that you want to read the analog input data from a channel on a DASCard-1002 card configured for a bipolar input range and a gain of 4. The count value is 1024. The voltage is determined as follows:

$$\frac{1024 \times 2.5}{4096} = 0.625\text{ V}$$

Converting Voltage to Counts

You must convert voltage to counts when specifying an analog trigger level or hysteresis value. The following sections describe how to convert voltage to counts for each of these situations.

Specifying a Trigger Level

To convert a voltage value to a count when specifying an analog trigger level, use the equation that is appropriate for your analog input range type, substituting the desired voltage for V_{trig}.

Bipolar

$$\text{Count} = \frac{V_{trig} \times 4096}{10}$$

Unipolar

$$\text{Count} = \frac{V_{trig} \times 4096}{5}$$

For example, assume that you want to specify an analog trigger level of 2.5 V for a channel on a DASCard-1001 card configured for a bipolar input range. The count is determined as follows:

$$\frac{2.5 \times 4096}{10} = 1024$$

Specifying a Hysteresis Value

To convert a voltage value to a count when specifying a hysteresis value, use the equation that is appropriate for your analog input range type, substituting the desired voltage for V_{hyst}.

Bipolar

$$\text{Count} = \frac{V_{hyst} \times 4096}{10}$$

Unipolar

$$\text{Count} = \frac{V_{hyst} \times 4096}{5}$$

For example, assume that you want to specify a hysteresis value of 0.05 V for a channel on a DASCard-1001 card configured for a unipolar input range. The count is determined as follows:

$$\frac{0.05 \times 4096}{5} = 41$$

Index