DASDLL
Function Call Driver

USER'S GUIDE

DASDLL
Function Call Driver
User's Guide

Revision A – December 1994
Part Number: 86590

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

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 using the DASDLL Function Call Driver. The DASDLL Function Call Driver supports the following Windows™-based languages:

- Microsoft Visual C++™ (Version 1.0 and higher)
- Microsoft Visual Basic for Windows (Version 3.0 and higher)

The manual is intended for application programmers using one of the following boards in an IBM® PC AT® or compatible computer:

- DAS-8 Series
- DAS-16 Series
- DAS-20
- DAS-40 Series
- DAS-HRES
- DDA-06
- Series 500
- PIO Series
- PDMA Series

Throughout this manual, these boards are referred to as DASDLL-supported boards.

It is assumed that users

- have read the External DAS Driver user's guide and the user's guide for their particular board to familiarize themselves with the board's features.

- have completed the appropriate hardware installation and configuration.
- are experienced in programming in their selected language and are familiar with data acquisition principles.

The *DASDLL Function Call Driver User's Guide* is organized as follows:

- Chapter 1 provides an overview of the Function Call Driver and describes the installation procedure. Information is included on setting up the board and how to get help, if necessary.
- Chapter 2 describes the available operations and 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 and their usage, arranged in alphabetical order.
- Appendix A contains a list of the error codes returned by the Function Call Driver, along with specific causes and suggested solutions.
- Appendix B contains instructions for converting counts to voltage and for converting voltage to counts.
- Appendix C provides board-specific operating specifications on gains and channels.
- Appendix D includes instructions for installing the Keithley Memory Manager.

An index completes this manual.

Note: The DASDLL-supported boards vary in their features and operating parameters. Information presented in this manual is generic to cover every board's requirements. For board-specific information, refer to your board's user's guide and External DAS Driver user's guide. Your board's user's guide is shipped with your board; the External DAS Driver user's guide is shipped with the DASDLL software package.

Table of Contents

4 Function Reference

List of Figures

List of Tables

1

Getting Started

The DASDLL Function Call Driver is a library of data acquisition and control functions (referred to as the Function Call Driver or FCD functions). Table 1-1 lists the Keithley DAS boards supported by the DASDLL Function Call Driver.

Table 1-1. Boards Supported

Series	Boards
DAS-8	DAS-8, DAS-8LT, DAS-8PGA, DAS-8PGA-G2, DAS-8/AO
DAS-16	DAS-16, DAS-16F, DAS-16G1, DAS-16G2
DAS-20	DAS-20
DAS-40	DAS-40G1, DAS-40G2
DAS-HRES	DAS-HRES
DDA-06	DDA-06
500	AMM1A, AMM2, AIM2, AIM3A, AIM4, AIM6 AIM7, AIM8, AIM9
PIO	PIO-12, PIO-24, PIO-32, PIO-96, PIO-HV
PDMA	PDMA-16, PDMA-32

Throughout this manual, the boards in Table 1-1 are referred to as DASDLL-supported boards.

The DASDLL software package contains the following:

- Dynamic Link Libraries (DLLs) of FCD functions for Microsoft Visual C++ and Microsoft Visual Basic for Windows.
- Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.
- Language-specific example programs.

The following sections describe how to install the software, how to set up a board to use the DASDLL Function Call Driver, and how to get help, if necessary.

Installing the Software

To install the DASDLL software package, perform the following steps:

1. Make a backup copy of the supplied disks. Use the copies as your working disks and store the originals as backup disks.
2. Insert disk #1 into the disk drive.
3. Start Windows, if necessary.
4. From the Program Manager menu, choose File and then choose Run.
5. Assuming that you are using disk drive A, type the following at the command line in the Run dialog box, and then select OK:

   ```
   A:SETUP
   ```

   The installation program prompts you for your installation preferences, including the drive and directory you want to copy the software to. It also prompts you to insert additional disks, as necessary.

6. Continue to insert disks and respond to prompts, as appropriate.

   When the installation program prompts you for a drive designation, enter a designation of your choosing or accept the default drive C. When the installation program prompts you for a directory name, enter a name of your choosing or accept the default name.

   The installation program creates a directory on the specified drive and copies all files, expanding any compressed files.

   The installation program also creates a DASDLL family group; this group includes example Windows programs.

7. When the installation program notifies you that the installation is complete, review the following files:

   – FILES.TXT lists and describes all the files copied to the hard disk by the installation program.

   – README.TXT contains information that was not available when this manual was printed.

Setting Up the Board and the Driver

Before you use the DASDLL Function Call Driver, you must perform the following tasks:

1. Set up your board's hardware. Refer to your board's user's guide and your External DAS Driver user's guide for information.
2. Exit Windows and return to DOS.
3. Run the configuration program for your board from DOS. The configuration program is shipped with the External DAS Driver for your board. Refer to your External DAS Driver user's guide for information.

Note: You cannot run the configuration program or load the External DAS Driver from the MS-DOS Prompt when in Windows. You must exit Windows and return to DOS.

---

4. Load the External DAS Driver for your board from DOS. Refer to your External DAS Driver user's guide for information.
5. Load Windows.

Note: If you want to set up your AUTOEXEC.BAT file to automatically load the External DAS Driver, make sure that you include the line that loads the External DAS Driver before the line that loads Windows.

Getting Help

If you need help installing or using the DASDLL 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:

DASDLL-supported board configuration	Model	_____
	Serial #	_____
	Revision code	_____
	Base address setting	_____
	Interrupt level setting	_____
	Input configuration	single-ended, differential
	Input range type	unipolar, bipolar
	DMA channel	_____
	Other	_____
Computer	Manufacturer	_____
	CPU type	_____
	Clock speed (MHz)	_____
	Amount of RAM	_____
	Video system	_____
	BIOS type	_____
Operating system	DOS version	_____
	Windows version	3.0, 3.1
	Windows mode	Standard, Enhanced
Software package	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 DASDLL-supported boards. The supported operations are listed in Table 2-1.

Table 2-1. Supported Operations

Operation	Page Reference
System	page 2-2
Analog input	page 2-6
Analog output	page 2-17
Digital input and output (I/O)	page 2-25

Note: The DASDLL-supported boards vary in their features and operating parameters. Information presented in this chapter is generic to cover every board's requirements. For board-specific information, refer to your board's user's guide and External DAS Driver user's guide. Your board's user's guide is shipped with your board; the External DAS Driver user's guide is shipped with the DASDLL software package.

The following features are not supported by the DASDLL Function Call Driver, even though the External DAS Driver for your DASDLL may support them:

- More than two memory buffers per frame
- Simultaneous sample-and-hold (SSH)
- Programmable external pacer clock polarity
- About-trigger acquisition
- Hardware gate
- Counter/timer functions
- Timed interrupt functions
- Time of Day (TOD) functions

System Operations

This section describes the miscellaneous and general maintenance operations that apply to DASDLL-supported boards and to the DASDLL Function Call Driver. It includes information on the following operations:

- Initializing the driver
- Initializing a board
- Retrieving revision levels
- Handling errors

Initializing the Driver

You must initialize the DASDLL 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; the driver returns a unique identifier for the driver (this identifier is called the driver handle).

If a particular driver is no longer required and you want to free some memory, you can use the **K_CloseDriver** function to free a driver handle and close the associated driver. The driver is shut down and the DLLs associated with the driver are shut down and unloaded from memory.

Note: You can also use the **DASDLL_DevOpen** function to initialize the driver and determine the number of boards found by the DASDLL Function Call Driver.

Initializing a Board

The number of boards supported by the DASDLL Function Call Driver depends on the number of External DAS Drivers you loaded and the number of boards supported by each External DAS Driver. You must use the **K_GetDevHandle** function to specify the boards you want to use. The driver returns a unique identifier for each board; this identifier is called the board handle.

Board handles allow you to communicate with more than one board. You use the board handle returned by **K_GetDevHandle** in subsequent function calls related to the board.

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

Note: You can also use the **DASDLL_GetDevHandle** function to specify the boards you are using.

The board number you specify in **K_GetDevHandle** is a logical board number; it is determined by how you loaded your External DAS Drivers. For example, Figure 2-1 illustrates a system in which you first loaded the DAS-8 External DAS Driver (configured for two boards) and then loaded the DAS-16 External DAS Driver (configured for two boards).

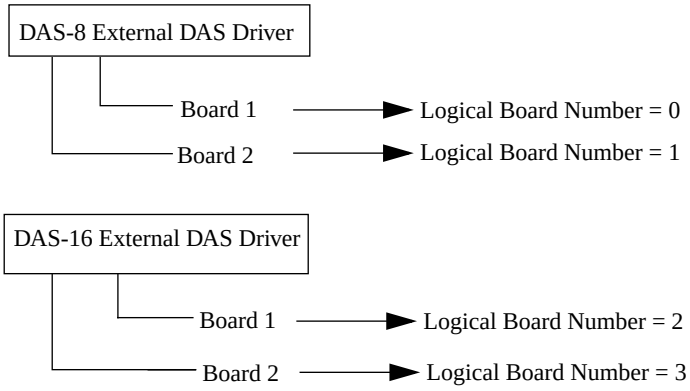


Figure 2-1. Logical Board Numbers

Note: The DASDLL Function Call Driver treats Series 500 modules as separate boards.

You can use the **DASDLL_GetBoardName** function to return information about the boards and drivers loaded in your system. When you enter a logical board number, the driver returns the name of the driver associated with the board. A NULL pointer is returned if no driver is associated with the board.

For example, if you set up a loop to return the names of the drivers associated with the boards shown in Figure 2-1, the driver returns four strings and a NULL pointer. The first two strings represent the DAS-8 External DAS Driver; the next two strings represent the DAS-16 External DAS Driver; the fifth string is a NULL pointer.

The returned strings indicate that your system contains four boards. The first two logical boards, 0 and 1, are DAS-8 Series boards; the next two, boards 2 and 3, are DAS-16 Series boards.

To reinitialize a board during an operation, use the **K_DASDevInit** function. **K_GetDevHandle**, **DASDLL_GetDevHandle**, and **K_DASDevInit** perform the following tasks:

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

Retrieving Revision Levels

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 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 is shared by all DASDLL-supported 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 programming language uses a different procedure for error checking. Refer to the following for information:

Visual C++	page 3-33
Visual Basic for Windows	page 3-37

For Visual C++ 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
  - Conversion mode
  - Clock source
  - Buffering mode
  - Trigger source

Note: The DASDLL-supported boards vary in their features and operating parameters. For board-specific information, such as voltage input ranges, refer to your board's user's guide and External DAS Driver user's guide.

Operation Modes

The operation mode determines which attributes you can specify for an analog input operation and how data is transferred from the board to computer memory. You can perform analog input operations in one of the following modes:

- **Single mode -** In single mode, the board 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 board 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 board acquires a single sample or multiple samples from one or more analog input channels. A hardware pacer clock initiates conversions. You cannot perform any other operation until the synchronous-mode operation is complete. After the driver transfers the specified number of samples to the host, the driver returns control to the application program, which reads the data.

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

- **Interrupt mode** - In interrupt mode, the board 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.

  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 2-14 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.

- **DMA mode** - In DMA mode, the board 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. DMA mode provides the fastest data transfer rates.

  Use the **K_DMAStart** function to start an analog input operation in DMA mode.

  You can specify either single-cycle or continuous buffering mode for DMA-mode operations. Refer to page 2-14 for more information on buffering modes. Use the **K_DMAStop** function to stop a continuous-mode DMA operation. Use the **K_DMAStatus** function to determine the current status of a DMA operation.

The converted data is stored as counts. For information on converting counts to voltage, refer to Appendix B.

Memory Allocation and Management

Interrupt-mode and DMA-mode analog input operations use one or two memory buffers to store acquired data; synchronous-mode analog input operations use one memory buffer to store acquired data. (You can use two memory buffers if your External DAS Driver supports double buffering; the driver automatically switches from the primary buffer to the secondary buffer when the primary buffer is full.)

Note: Except for DASDLL-40 Series boards, it is recommended that you always use a single memory buffer, particularly for analog input operations faster than 1 kHz.

Use one of the following functions to allocate memory:

- **K_SyncAlloc** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAAlloc** for DMA-mode operations.

You specify the following:

- Operation requiring the memory buffer.
- Number of samples to store in the memory buffer (up to 32,767).

The driver returns the starting address of the memory buffer and a unique identifier for the buffer (this identifier is called the memory handle).

When the memory buffer is no longer required, you can free the buffer for another use by specifying the memory handle in one of the following functions:

- **K_SyncFree** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAFree** for DMA-mode operations.

If you are using two memory buffers, you can work on data in the inactive buffer while the active buffer continues to collect data. To determine the active buffer, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode). Depending on the speed of your operation and the particular board you are using, data may be lost when the driver switches from one memory buffer to the other. To determine whether any data has been lost, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode).

Notes: For synchronous-mode and interrupt-mode operations and for DMA-mode operations on DAS-16 Series boards, memory is allocated from the first 1MB of DOS memory only; therefore, the amount of memory you can allocate may be limited.

For DAS-20 and DAS-HRES boards that run in DMA mode, it is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate large enough memory buffers. Refer to Appendix D for more information about the Keithley Memory Manager.

To eliminate page wrap conditions and to guarantee that memory is suitable for use by the computer's controller, **DASDLL_DMAAlloc** may allocate an area twice as large as actually needed. Once the data in this buffer is processed and/or saved elsewhere, use **DASDLL_DMAFree** to free the memory for other uses.

For Visual Basic for Windows, the program cannot transfer data directly from the memory buffer. You must use the **K_MoveBufToArray** function to move the data from the memory buffer to the program's local array; refer to page 4-81 for more information.

After you allocate your memory buffers, you must assign the starting address of the buffers and the number of samples to store in the buffers. Each supported programming language requires a particular procedure for allocating a memory buffer and assigning the starting address. Refer to the following for information:

Visual C++	page 3-33
Visual Basic for Windows	page 3-37

Channels and Gains

Analog input channels on DASDLL-supported boards measure signals in several analog input ranges. The analog input range for a particular channel depends on the gain of the channel. The driver uses gain codes to represent the gain.

For example, on a DAS-8PGA analog input board, an analog input range of 0 to 10 V translates to a gain of 1 and a gain code of 9. Refer to Appendix C for a summary of the gain codes used by DASDLL-supported boards.

For most DASDLL-supported boards, channels can be configured as single-ended or differential. The number of channels supported depends on which configuration you use.

If you require more than the supported number of channels, you can use expansion accessories to increase the number of available channels. Refer to your board's user's guide and to the appropriate expansion accessory documentation for more information.

Refer to Appendix C for a summary of the number of channels on DASDLL-supported boards.

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

Single 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, interrupt-mode, and DMA-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.

Multiple Channels Using a Group of Consecutive Channels

For synchronous-mode, interrupt-mode, and DMA-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.

Use the **K_SetG** function to specify the gain code for all 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.

Multiple Channels Using a Channel-Gain Queue

For synchronous-mode, interrupt-mode, and DMA-mode analog input operations, you can acquire samples from channels in a 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 a gain code for each channel.

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 required number of samples is read.

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 for information:

Visual C++	page 3-33
Visual Basic for Windows	page 3-37

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.

Note: You can use a channel-gain queue with DMA-mode operations on DAS-20 and DAS-40 Series boards only.

Pacer Clock

The pacer clock determines the period between the conversion of one channel and the conversion of the next channel. For synchronous-mode, interrupt-mode, and DMA-mode analog input operations, use the **K_SetClk** function to specify one of the following pacer clocks:

- **Internal pacer clock** - The internal pacer clock uses an onboard counter. You load a value into the counter to determine the period between conversions. Depending on the time base of the counter, each count represents a particular time period. Table 2-2 lists the time bases available on DASDLL-supported boards.

Table 2-2. Time Bases

Board	Time Base
DAS-8	Depends on PC bus clock frequency[1]
DAS-8LT DAS-8PGA DAS-8PGA-02 DAS-8/AO	1 MHz
DAS-16 Series	1 MHz or 10 MHz[1]
DAS-20	5 MHz
DAS-40 Series	4 MHz
DAS-HRES	1 MHz, 8 MHz, or 10 MHz[1]
DDA-06	Not applicable[2]
Series 500	1 MHz

Table 2-2. Time Bases (cont.)

Board	Time Base
PIO Series	Not applicable[2]
PDMA Series	10 MHz

Notes
[1] Specified in the External DAS Driver configuration.
[2] DDA-06 and PIO Series boards do not support an internal pacer clock.

Use the **K_SetClkRat**e function to specify the number of counts (clock ticks) between conversions. For example, if you are using a DAS-8PGA board (1 MHz time base), each count represents 1.0 µs. If you specify a count of 30, the period between conversions is 30 µs (33.33 ksamples/s).

When using an internal pacer clock, use the following formula to determine the number of counts to specify:

$$\text{counts} = \frac{\text{time base}}{\text{conversion rate}}$$

For example, if you want a conversion rate of 10 ksamples/s on a DAS-8PGA board, specify a count of 100, as shown in the following equation:

$$\frac{1,000,000}{10,000} = 100$$

The internal pacer clock is the default pacer clock.

- **External pacer clock** - You connect an external pacer clock to the appropriate pin on the main I/O connector.

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

  Refer to your DAS board's user's guide to determine which edge (positive or negative) is the active edge supported for your board.

Notes: Make sure that the pacer clock initiates conversions at a rate that the analog-to-digital converter (ADC) can handle.

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

Buffering Modes

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

- **Single-cycle mode** - In single-cycle mode, after the board converts the specified number of samples and stores them in the buffer, the operation stops automatically. Single-cycle mode is the default buffering mode.
- **Continuous mode** - In continuous mode, the board 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.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For synchronous-mode, interrupt-mode, and DMA-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 immediately after you start the analog input operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**). The point at which conversions begin depends on the pacer clock; refer to page 2-12 for more information. The internal trigger is the default trigger source.

- **External trigger** - When you start the analog input operation (using **K_SyncStart**, **K_IntStart** or **K_DMAStart**), the application program waits until an external trigger event occurs. For Series 500 boards, the external trigger is an analog trigger; for DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, and DAS-HRES boards, the external trigger is a digital trigger. The point at which conversions begin depends on the pacer clock; refer to page 2-12 for more information.

Note: DDA-06 and PIO Series boards do not support an external trigger.

Analog and digital triggers are described in the following sections.

Analog Trigger

Only Series 500 boards support an external analog trigger. An analog trigger event occurs when a particular condition is met by the analog input signal on a specified analog trigger channel. Use the **K_SetADTrig** function to specify the following:

- Analog input channel to use as the trigger channel.
- Voltage level. You specify the voltage level as a count value between 0 and 8191, where 0 represents –10 V and 8191 represents +10 V.
- Trigger polarity and sensitivity. Depending on your board, the trigger event occurs when one of the following conditions is met:
  - **Positive-edge trigger** - The analog input signal rises above the specified voltage level.
  - **Negative-edge trigger** - The analog input signal falls below the specified voltage level.

  Figure 2-2 illustrates these analog trigger conditions, where the specified voltage level is +5 V.

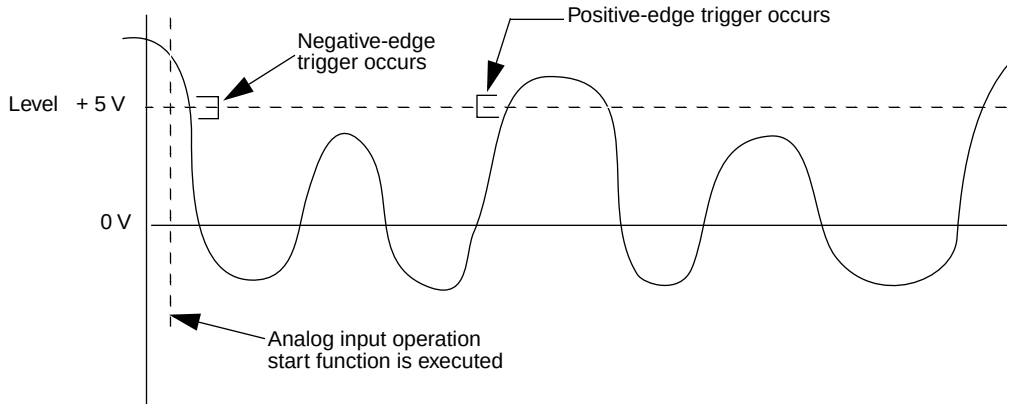


Figure 2-2. Analog Trigger Conditions

Digital Trigger

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, and DAS-HRES boards support an external digital trigger. A digital trigger event occurs when a particular condition is met by the digital trigger signal, which is connected to the appropriate pin on the main I/O connector. Depending on your board, the trigger event occurs when one of the following conditions is met:

- **Positive-edge trigger** - A rising edge occurs on the digital trigger signal.
- **Negative-edge trigger** - A falling edge occurs on the digital trigger signal.
- **Positive-level trigger** - The digital trigger signal is high.
- **Negative-level trigger** - The digital trigger signal is low.

Refer to your board's user's guide and External DAS Driver user's guide for information about the digital trigger conditions supported for your board.

Analog Output Operations

This section describes the following:

- Analog output operation modes available.
- How to allocate and manage memory for analog output operations.
- How to specify the following for an analog output operation:
  - Channel
  - Clock source
  - Buffering mode
  - Digital trigger condition

Operation Modes

The operation mode determines which attributes you can specify for an analog output operation. You can perform analog output operations in one of the following modes:

- **Single mode** - In single mode, the driver writes a single value to an analog output channel; you cannot perform any other operation until the single-mode operation is complete.

  Use the **K_DAWrite** function to start an analog output operation in single mode. You specify the board you want to use, the analog output channel, and the value you want to write.

- **Synchronous mode** - In synchronous mode, the driver writes a single value or multiple values to an analog output channel. A hardware pacer clock paces the updating of the channel. You cannot perform any other operation until the synchronous-mode operation is complete. After the driver writes the specified number of values, the driver returns control to the application program.

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

- **Interrupt mode** - In interrupt mode, the driver writes a single value or multiple values to an analog output channel. A hardware clock paces the updating of the channel. Once the analog output operation begins, control returns to your application program.

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

You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-23 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.

- **DMA mode** - In DMA mode, the driver writes a single value or multiple values to an analog output channel. A hardware clock paces the updating of the channel. Once the analog output operation begins, control returns to your application program. DMA mode provides the fastest data transfer rates.

  Use the **K_DMAStart** function to start an analog output operation in DMA mode.

  You can specify either single-cycle or continuous buffering mode for DMA-mode operations. Refer to page 2-23 for more information on buffering modes. Use the **K_DMAStop** function to stop a continuous-mode DMA operation. Use the **K_DMAStatus** function to determine the current status of a DMA operation.

For an analog output operation, the values are written as counts. For information on converting voltage to counts, refer to Appendix B.

Memory Allocation and Management

Interrupt-mode and DMA-mode analog output operations use one or two memory buffers to store acquired data; synchronous-mode analog output operations use one memory buffer to store acquired data. (You can use two memory buffers if your External DAS Driver supports double buffering; the driver automatically switches from the primary buffer to the secondary buffer when the primary buffer is empty.)

Note: It is recommended that you always use a single memory buffer, particularly for analog output operations faster than 1 kHz.

Use one of the following functions to allocate memory:

- **K_SyncAlloc** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAAlloc** for DMA-mode operations.

You specify the following:

- Operation requiring the memory buffer.
- Number of samples to store in the memory buffer (up to 32,767).

The driver returns the starting address of the memory buffer and a unique identifier for the buffer (this identifier is called the memory handle).

When the memory buffer is no longer required, you can free the buffer for another use by specifying the memory handle in one of the following functions:

- **K_SyncFree** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAFree** for DMA-mode operations.

If you are using two memory buffers, you can work on data in the inactive buffer while the active buffer continues to collect data. To determine the active buffer, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode). Depending on the speed of your operation and the particular board you are using, data may be lost when the driver switches from one memory buffer to the other. To determine whether any data has been lost, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode).

If you are using a group of analog output channels, when you start the analog output operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), the driver simultaneously writes one value to each channel in the group. The driver writes the first value in the memory buffer to the first channel, the second value in the buffer to the second channel, the third value in the buffer to the third channel, and so on. To ensure predictable results, make sure that the number of values stored in the memory buffer is an even multiple of the number of channels in the group.

Notes: For synchronous-mode and interrupt-mode operations, memory is allocated from the first 1MB of DOS memory only; therefore, the amount of memory you can allocate may be limited.

For DAS-20 boards that run in DMA mode, it is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate large enough memory buffers. Refer to Appendix D for more information about the Keithley Memory Manager.

To eliminate page wrap conditions and to guarantee that memory is suitable for use by the computer's controller, **DASDLL_DMAAlloc** may allocate an area twice as large as actually needed. Once the data in this buffer is processed and/or saved elsewhere, use **DASDLL_DMAFree** to free the memory for other uses.

For Visual Basic for Windows, the program cannot transfer data directly to the memory buffer. You must use the **K_MoveArrayToBuf** function to move the data from the program's local array to the memory buffer; refer to page 4-79 for more information.

After you allocate your memory buffers, you must assign the starting address of the buffers and the number of samples stored in the buffers. Each supported programming language requires a particular procedure for allocating a buffer. Refer to the following for information:

Visual C++	page 3-30
Visual Basic for Windows	page 3-34

Channels

DASDLL-supported boards that perform analog output operations contain one or more digital-to-analog converters (DACs). Each DAC is associated with an analog output channel. You can perform the analog output operation on a single channel or on a group of multiple channels. The following subsections describe how to specify the channels you are using.

Single Channel

For single-mode analog output operations, you can write a single value to a single analog output channel. Use the **K_DAWrite** function to specify the channel.

For synchronous-mode, interrupt-mode, and DMA-mode analog output operations, you can write a single value or multiple values to a single analog output channel. Use the **K_SetChn** function to specify the channel. At each pulse of the pacer clock, the driver updates all the analog output channels and then writes a new value to the specified channel only.

Multiple Channels

For synchronous-mode, interrupt-mode, and DMA-mode analog output operations, you can write a single value or multiple values to a group of consecutive analog output channels. Use the **K_SetStartStopChn** function to specify the first and last channels in the group. At each pulse of the pacer clock, the driver updates all the analog output channels and then writes new values to the channels in the group only.

For example, assume that the start channel is 0, the stop channel is 1, and your array contains two waveforms (0, 4095, 1, 4094, 2, 4093, . . 4095, 0). At the first pulse of the pacer clock, the driver updates all the analog output channels and then simultaneously writes 0 to channel 0 and 4095 to channel 1; at the next pulse of the pacer clock, the driver updates all the analog output channels and then simultaneously writes 1 to channel 0 and 4094 to channel 1.

Pacer Clock

The pacer clock determines the period between updates of an analog output channel. For synchronous-mode, interrupt-mode, or DMA-mode analog output operations, use the **K_SetClk** function to specify one of the following pacer clocks:

- **Internal pacer clock** - The internal pacer clock uses an onboard counter. You load a value into the counter to determine the period between updates. Depending on the time base of the counter, each count represents a particular time period. Refer to Table 2-2 on page 2-12 for a list of the time bases available on DASDLL-supported boards.

Use the **K_SetClkRat**e function to specify the number of counts (clock ticks) between updates. For example, if you are using a DAS-8/AO board (1 MHz time base), each count represents 1.0 µs. If you specify a count of 30, the period between updates is 30 µs (33.33 ksamples/s).

When using an internal pacer clock, use the following formula to determine the number of counts to specify:

$$\text{counts} = \frac{\text{time base}}{\text{update rate}}$$

For example, if you want an update rate of 10 ksamples/s on a DAS-8/AO board, specify a count of 100, as shown in the following equation:

$$\frac{1,000,000}{10,000} = 100$$

The internal pacer clock is the default pacer clock.

- **External pacer clock** - You connect an external pacer clock to the appropriate pin on the main I/O connector.

  When you start an analog output operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), conversions are armed. At the next active edge of the external pacer clock (and at every subsequent active edge of the external pacer clock), the analog output channel is updated.

  Refer to your DAS board's user's guide to determine which edge (positive or negative) is the active edge supported for your board.

Notes: At each pulse of the pacer clock, the driver updates all the analog output channels on the board and then writes new values to the channels specified in **K_SetChn** or **K_SetStartStopChn** only.

You cannot use the internal pacer clock or the external pacer clock for analog output operations if the clock is being used by another operation.

The actual update rate also depends on other factors, including your computer, the operating system/environment, and other software issues.

Buffering Modes

The buffering mode determines how the driver writes the values in the host buffer to the analog output channel. For interrupt-mode and DMA-mode analog output operations, you can specify one of the following buffering modes:

- **Single-cycle mode** - In single-cycle mode, after the driver writes the values stored in the buffer, the operation stops automatically. Single-cycle mode is the default buffering mode.
- **Continuous mode** - In continuous mode, the driver continuously writes values from the buffer until the application program issues a stop function; when all the values in the buffer have been written, the driver writes the values again. Use the **K_SetContRun** function to specify continuous buffering mode.

Note: Buffering modes are not meaningful for synchronous-mode operations.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For synchronous-mode, interrupt-mode, and DMA-mode analog output 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 immediately after you start the analog output operation (using **K_SyncStart**, **K_IntStart,** or **K_DMAStart**). The point at which the channel is updated depends on the pacer clock; refer to page 2-21 for more information. The internal trigger is the default trigger source.
- **External trigger** - DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, and DAS-HRES boards support an external trigger. An external trigger is a digital trigger signal connected to the appropriate pin on the main I/O connector. When you start the analog output operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), the application program waits until the trigger event occurs. Depending on your board, the trigger event occurs when one of the following conditions is met:
  - **Positive-edge trigger** - A rising edge occurs on the digital trigger signal.
  - **Negative-edge trigger** - A falling edge occurs on the digital trigger signal.
  - **Positive-level trigger** - The digital trigger signal is high.
  - **Negative-level trigger** - The digital trigger signal is low.

Refer to your board's user's guide and External DAS Driver user's guide for information about the digital trigger conditions supported for your board.

The point at which updates begin depends on the pacer clock; refer to page 2-21 for more information.

Digital I/O Operations

This section describes the following:

- Digital I/O operation modes available.
- How to allocate and manage memory for digital I/O operations.
- Digital I/O channels.
- How to specify the following for a digital I/O operation:
  - Clock source
  - Buffering mode
  - Digital trigger condition

Operation Modes

The operation mode determines which attributes you can specify for a digital I/O operation. You can perform digital I/O operations in one of the following modes:

- **Single mode** - In a single-mode digital input operation, the driver reads the value of a digital input channel once; in a single-mode digital output operation, the driver writes a value to a digital output channel once. You cannot perform any other operation until the single-mode operation is complete.

  Use the **K_DIRead** function to start a digital input operation in single mode; you specify the board you want to use, the digital input channel, and the variable in which to store the value. Use the **K_DOWrite** function to start a digital output operation in single mode; you specify the board you want to use, the digital output channel, and the digital output value.

- **Synchronous mode** - In a synchronous-mode digital input operation, the driver reads the value of a digital input channel multiple times; in a synchronous-mode digital output operation, the driver writes a single value or multiple values to a digital output channel multiple times. A hardware pacer clock paces the digital I/O operation. You cannot perform any other operation until the synchronous-mode operation is complete.

  Use the **K_SyncStart** function to start a digital I/O operation in synchronous mode.

- **Interrupt mode** - In an interrupt-mode digital input operation, the driver reads the value of a digital input channel multiple times; in an interrupt-mode digital output operation, the driver writes a single value or multiple values to a digital output channel multiple times.

  A hardware clock paces the digital I/O operation. Once the digital I/O operation begins, control returns to your application program.

  Use the **K_IntStart** function to start a digital I/O operation in interrupt mode.

  You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-31 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.

- **DMA mode** - In a DMA-mode digital input operation, the driver reads the value of a digital input channel multiple times; in a DMA-mode digital output operation, the driver writes a single value or multiple values to a digital output channel multiple times.

  A hardware clock paces the digital I/O operation. Once the digital I/O operation begins, control returns to your application program. DMA mode provides the fastest data transfer rates.

  Use the **K_DMAStart** function to start a digital I/O operation in DMA mode.

  You can specify either single-cycle or continuous buffering mode for DMA-mode operations. Refer to page 2-31 for more information on buffering modes. Use the **K_DMAStop** function to stop a continuous-mode DMA operation. Use the **K_DMAStatus** function to determine the current status of a DMA operation.

Memory Allocation and Management

Interrupt-mode and DMA-mode digital I/O operations use one or two memory buffers to store the data to be read or written; synchronous-mode digital I/O operations use one memory buffer to store the data to be read or written. (You can use two memory buffers if your External DAS Driver supports double buffering; the driver automatically switches from the primary buffer to the secondary buffer when the primary buffer is full or empty.)

Note: It is recommended that you always use a single memory buffer, particularly for digital I/O operations faster than 1 kHz.

Use one of the following functions to allocate memory:

- **K_SyncAlloc** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAAlloc** for DMA-mode operations.

You specify the following:

- Operation requiring the memory buffer.
- Number of samples to store in the memory buffer (up to 32,767).

The driver returns the starting address of the memory buffer and a unique identifier for the buffer (this identifier is called the memory handle).

When the memory buffer is no longer required, you can free the buffer for another use by specifying the memory handle in one of the following functions:

- **K_SyncFree** for synchronous-mode or interrupt-mode operations.
- **DASDLL_DMAFree** for DMA-mode operations.

If you are using two memory buffers, you can work on data in the inactive buffer while the active buffer continues to collect data. To determine the active buffer, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode). Depending on the speed of your operation and the particular board you are using, data may be lost when the driver switches from one memory buffer to the other. To determine whether any data has been lost, use the **K_IntStatus** function (for interrupt mode) or the **K_DMAStatus** function (for DMA mode).

Notes: For synchronous-mode and interrupt-mode operations, memory is allocated from the first 1MB of DOS memory only; therefore, the amount of memory you can allocate may be limited.

To eliminate page wrap conditions and to guarantee that memory is suitable for use by the computer's controller, **DASDLL_DMAAlloc** may allocate an area twice as large as actually needed. Once the data in this buffer is processed and/or saved elsewhere, use **DASDLL_DMAFree** to free the memory for other uses.

For Visual Basic for Windows, the data in the memory buffer is not directly accessible by your program. For digital input operations, you must use the **K_MoveBufToArray** function to move the data from the memory buffer to the program's local array; refer to page 4-81 for more information. For digital output operations, you must use the **K_MoveArrayToBuf** function to move the data from the program's local array to the memory buffer; refer to page 4-79 for more information.

After you allocate your memory buffers, you must assign the starting address of the buffers and the number of samples stored in the buffers. Each supported programming language requires a particular procedure for allocating a buffer. Refer to the following for information:

Visual C++	page 3-30
Visual Basic for Windows	page 3-34

Channels

You can read values from or write values to one or more of the digital I/O lines on your board. Refer to your board's user's guide and External DAS Driver user's guide for information about the number of digital I/O lines available on your board.

For Series 500 boards, the DASDLL Function Call Driver treats each 8-bit digital input port or 8-bit digital output port as a separate channel.

For DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, DDA-06, PIO Series, and PDMA Series boards, the DASDLL Function Call Driver supports one digital input channel and one digital output channel. When specifying your digital I/O ports in the External DAS Driver configuration, you must make sure that all the digital I/O lines can be accommodated on a single channel. For example, if you want to use all 24 bits on a PIO-12 board for digital output, you must configure a single 24-bit channel. You cannot configure three 8-bit channels.

For single-mode digital I/O operations, use the **K_DIRead** function to specify a single digital input channel; use **K_DOWrite** to specify a single digital output channel. For synchronous-mode, interrupt-mode, and DMA-mode digital I/O operations, use the **K_SetChn** function to specify a single digital I/O channel or the **K_SetStartStopChn** function to specify multiple digital I/O channels.

Each bit in a digital I/O channel corresponds to one of the digital I/O lines on the board. The bits can be configured as digital inputs or digital outputs. A value of 1 in a bit position indicates that the input or output is high; a value of 0 in a bit position indicates that the input or output is low. If no signal is connected to a digital input line, the input appears high (value is 1).

Notes: On some DASDLL-supported boards, a digital I/O line may also be used for another purpose, such as an external trigger. In these cases, you cannot use the digital I/O line for general-purpose digital I/O operations.

Pacer Clock

The pacer clock determines the period between reading the digital input channel or writing to the digital output channel. For synchronous-mode, interrupt-mode, and DMA-mode digital I/O operations, use the **K_SetClk** function to specify one of the following pacer clocks:

- **Internal pacer clock** - The internal pacer clock uses an onboard counter. You load a value into the counter to determine the period between reads/writes. Depending on the time base of the counter, each count represents a particular time period. Refer to Table 2-2 on page 2-12 for a list of the time bases available on DASDLL-supported boards.

  Use the **K_SetClkRat**e function to specify the number of counts (clock ticks) between reads/writes. For example, if you are using a DAS-8PGA board (1 MHz time base), each count represents 1.0 µs. If you specify a count of 30, the period between reads/writes is 30 µs (33.33 ksamples/s).

  When using an internal pacer clock, use the following formula to determine the number of counts to specify:

$$\text{counts} = \frac{\text{time base}}{\text{read/write rate}}$$

  For example, if you want a read/write rate of 10 ksamples/s on a DAS-8/AO board, specify a count of 100, as shown in the following equation:

$$\frac{1,000,000}{10,000} = 100$$

  The internal pacer clock is the default pacer clock.

- **External pacer clock** - You connect an external pacer clock to the appropriate pin on the main I/O connector.

  When you start a digital I/O operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), conversions are armed. At the next active edge of the external pacer clock (and at every subsequent active edge of the external pacer clock), a conversion is initiated. Refer to your board's user's guide to determine which edge (positive or negative) is the active edge supported for your board.

Notes: You cannot use the internal pacer clock or the external pacer clock for digital I/O operations if the clock is being used by another operation.

The actual read/write rate also depends on other factors, including your computer, the operating system/environment, and other software issues.

Buffering Modes

The buffering mode determines how the driver reads or writes the values in the buffer. For interrupt-mode and DMA-mode digital I/O operations, you can specify one of the following buffering modes:

- **Single-cycle mode** - In a single-cycle-mode digital input operation, after the driver fills the buffer, the operation stops automatically. In a single-cycle-mode digital output operation, after the driver writes the values stored in the buffer, the operation stops automatically. Single-cycle mode is the default buffering mode.
- **Continuous mode** - In a continuous-mode digital input operation, the driver continuously reads a digital input channel and stores the values in the buffer until the application program issues a stop function; any values already stored in the buffer are overwritten. In a continuous mode digital output operation, the driver continuously writes values from the buffer to a digital output channel until the application program issues a stop function; when all the values in the buffer have been written, the driver writes the values again. You use the **K_SetContRun** function to specify continuous buffering mode.

Note: Buffering modes are not meaningful for synchronous-mode operations.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For synchronous-mode and interrupt-mode digital I/O 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 immediately after you start the digital I/O operation (using **K_SyncStart** or **K_IntStart**). The point at which a value is read or written depends on the pacer clock; refer to page 2-30 for more information. The internal trigger is the default trigger source.
- **External trigger** - DAS-8 Series, DAS-16 Series, DAS-20, and DAS-HRES boards support an external trigger. An external trigger is a digital trigger signal connected to the appropriate pin on the main I/O connector. When you start the digital I/O (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), the application program waits until the trigger event occurs. Depending on your board, the trigger event occurs when one of the following conditions is met:
  - **Positive-edge trigger** - A rising edge occurs on the digital trigger signal.
  - **Negative-edge trigger** - A falling edge occurs on the digital trigger signal.
  - **Positive-level trigger** - The digital trigger signal is high.
  - **Negative-level trigger** - The digital trigger signal is low.

Refer to your board's user's guide and External DAS Driver user's guide for information about the digital trigger conditions supported for your board.

The point at which updates begin depends on the pacer clock; refer to page 2-30 for more information.

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

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**.

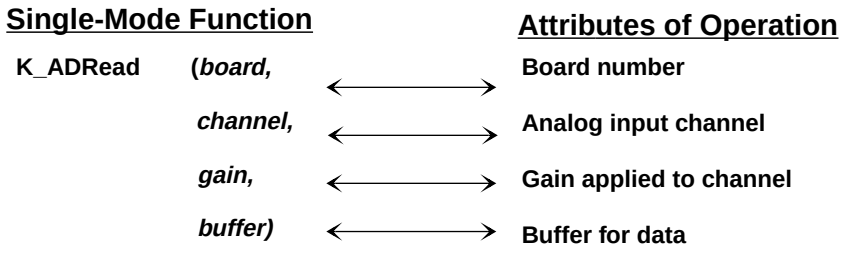


Figure 3-1. Single-Mode Function

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

Frames help you create structured application programs. You set up the attributes of the I/O 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 only available for I/O operations that use frames.

You indicate that you want to perform an I/O 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 I/O 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 I/O operation, you might use the **K_SetChn** setup function.

For each setup function, the Function Call Driver provides a readback function, which reads the current definition of a particular element. For example, the **K_GetChn** readback function reads the channel number specified for the I/O operation.

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

When you are ready to perform the I/O 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**.

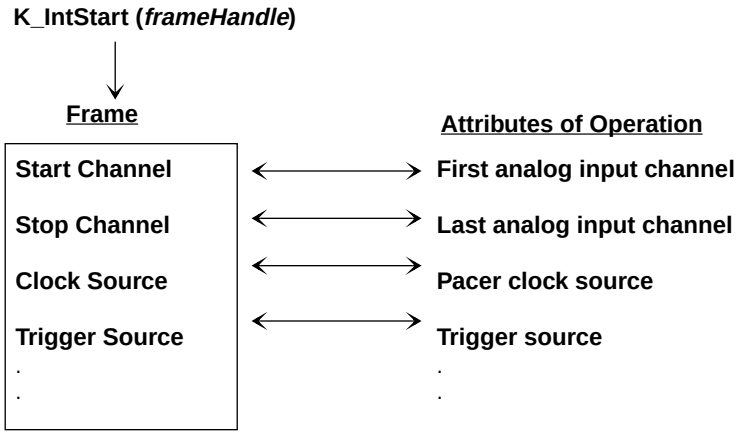


Figure 3-2. Interrupt-Mode Operation

Different I/O operations require different types of frames. For example, to perform a digital input operation, you use a digital input frame; to perform an analog output operation, you use an analog output frame.

For DASDLL-supported boards, synchronous-mode, interrupt-mode, and DMA-mode operations require frames. The DASDLL Function Call Driver provides the following types of frames:

- Analog input frames, called A/D (analog-to-digital) frames. You use the **K_GetADFrame** function to access an available A/D frame and a frame handle.
- Analog output frames, called D/A (digital-to-analog) frames. You use the **K_GetDAFrame** function to access an available D/A frame and a frame handle.
- Digital input frames, called DI frames. You use the **K_GetDIFrame** function to access an available DI frame and a frame handle.
- Digital output frames, called DO frames. You use the **K_GetDOFrame** function to access an available DO frame and a frame handle.

If you want to perform a synchronous-mode, interrupt-mode, or DMA-mode operation and all frames of a particular type 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.

For DASDLL-supported boards, the elements for each specific frame type are listed as follows:

- A/D frame elements - Table 3-1.
- D/A frame elements - Table 3-2 on page 3-6.
- DI frame elements - Table 3-3 on page 3-7.
- DO frame elements - Table 3-4 on page 3-8.

These tables also list the default values of each element, the setup functions used to define each element, and the readback functions used to read the current definition of the element.

Table 3-1. A/D Frame Elements

Element	Default Value	Setup Function	Readback Function
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Number of Samples	0	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[2]	K_GetContRun
Start Channel	0	K_SetChn K_SetStartStopChn K_SetStartStopG	K_GetChn K_GetStartStopChn K_GetStartStopG

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

Element	Default Value	Setup Function	Readback Function
Stop Channel	0	K_SetStartStopChn K_SetStartStopG	K_GetStartStopChn K_GetStartStopG
Gain	0	K_SetG K_SetStartStopG	K_GetG K_GetStartStopG
Channel-Gain Queue	0 (NULL)	K_SetChnGAry	K_GetChnGAry
Clock Source	Internal	K_SetClk	K_GetClk
Pacer Clock Rate[1]	0	K_SetClkRate	K_GetClkRate
Trigger Source	Internal	K_SetTrig	K_GetTrig
Trigger Type	Digital	K_SetADTrig K_SetDITrig	K_GetADTrig K_GetDITrig
Trigger Channel	0 (for analog trigger)	K_SetADTrig	K_GetADTrig
	0 (for digital trigger)	K_SetDITrig	K_GetDITrig
Trigger Polarity and Sensitivity	Positive edge (for analog trigger)	K_SetADTrig	K_GetADTrig
	Positive edge (for digital trigger)	K_SetDITrig	K_GetDITrig
Trigger Level	0	K_SetADTrig	K_GetADTrig
Trigger Hysteresis	0	K_SetTrigHyst	K_GetTrigHyst

Notes

[1] This element must be set.

[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.

Table 3-2. D/A Frame Elements

Element	Default Value	Setup Function	Readback Function
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Number of Samples	0	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[2]	K_GetContRun
Start Channel	0	K_SetChn K_SetStartStopChn	K_GetChn K_GetStartStopChn
Stop Channel	0	K_SetStartStopChn	K_GetStartStopChn
Clock Source	Internal	K_SetClk	K_GetClk
Pacer Clock Rate[1]	0	K_SetClkRate	K_GetClkRate
Trigger Source	Internal	K_SetTrig	K_GetTrig
Trigger Type	Digital	K_SetDITrig	K_GetDITrig
Trigger Channel	0 (for digital trigger)	K_SetDITrig	K_GetDITrig
Trigger Polarity and Sensitivity	Positive edge	K_SetDITrig	K_GetDITrig

Notes

[1] This element must be set.

[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.

Table 3-3. DI Frame Elements

Element	Default Value	Setup Function	Readback Function
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Number of Samples	0	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[2]	K_GetContRun
Start Channel	0	K_SetChn K_SetStartStopChn	K_GetChn K_GetStartStopChn
Stop Channel	0	K_SetStartStopChn	K_GetStartStopChn
Clock Source	Internal	K_SetClk	K_GetClk
Pacer Clock Rate[1]	0	K_SetClkRate	K_GetClkRate
Trigger Source	Internal	K_SetTrig	K_GetTrig
Trigger Type	Digital	K_SetDITrig	K_GetDITrig
Trigger Channel	0 (for digital trigger)	K_SetDITrig	K_GetDITrig
Trigger Polarity and Sensitivity	Positive edge	K_SetDITrig	K_GetDITrig

Notes

[1] This element must be set.

[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.

Table 3-4. DO Frame Elements

Element	Default Value	Setup Function	Readback Function
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf K_GetBufB
Number of Samples	0	K_SetBuf K_SetBufB K_SetDMABuf K_SetDMABufB	K_GetBuf
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[2]	K_GetContRun
Start Channel	0	K_SetChn K_SetStartStopChn	K_GetChn K_GetStartStopChn
Stop Channel	0	K_SetStartStopChn	K_GetStartStopChn
Clock Source	Internal	K_SetClk	K_GetClk
Pacer Clock Rate[1]	0	K_SetClkRate	K_GetClkRate
Trigger Source	Internal	K_SetTrig	K_GetTrig
Trigger Type	Digital	K_SetDITrig	K_GetDITrig
Trigger Polarity and Sensitivity	Positive edge	K_SetDITrig	K_GetDITrig

Notes

[1] This element must be set.

[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.

Note: The DASDLL 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, 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 DASDLL Function Call Driver, perform the following steps:

1. Define the application's requirements. Refer to Chapter 2 for a description of the board 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, describes the programming tasks that are common to all application programs.
   - Operation-Specific Programming Tasks, on page 3-10, describes operation-specific programming tasks and the sequence in which these tasks must be performed.
   - Chapter 4 contains detailed descriptions of the FCD functions.
   - The DASDLL software package contains several example programs. The FILES.TXT file in the installation directory lists and describes the example programs.
3. Compile and link the program. Refer to Language-Specific Programming Information, starting on page 3-29, for compile and link statements and other language-specific considerations for each supported language.

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. This file is included in the DASDLL software package.
2. Declare and initialize program variables.
3. Use the **K_DevOpen** function to initialize the driver.
4. Use the **K_GetDevHandle** function to specify the board you want to use and to initialize the board. If you are using more than one board, use the **K_GetDevHandle** function once for each board you are using.

Operation-Specific Programming Tasks

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

Note: Any FCD functions that are not mentioned in the operation-specific programming tasks can be used at any point in your application program.

Analog Input Operations

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

Single Mode

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

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

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the acquired data.
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 2-11 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 3-5.

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 3-5. Setup Functions for Synchronous-Mode Analog Input Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Start Channel	K_SetChn K_SetStartStopChn K_StartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStopG
Channel-Gain Queue	K_SetChnGAry
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig K_SetDITrig
Trigger Polarity and Sensitivity	K_SetADTrig K_SetDITrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst

Notes
[1] This element must be set.

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*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to the program's local array.
7. Use the **K_SyncFree** function to deallocate the buffers.
8. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt Mode

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

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the acquired data.
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 2-11 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 3-6.

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 3-6. Setup Functions for Interrupt-Mode Analog Input Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn K_StartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStop
Channel-Gain Queue	K_SetChnGAry
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig K_SetDITrig
Trigger Polarity and Sensitivity	K_SetADTrig K_SetDITrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst

Notes

[1] This element must be set.

[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.

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*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to the program's local array.
9. Use the **K_SyncFree** function to deallocate the buffers.
10. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

DMA Mode

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

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Use the **DASDLL_DMAAlloc** function to allocate the buffers in which to store the acquired data.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-7.

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 3-7. Setup Functions for DMA-Mode Analog Input Operations

Attribute	Setup Functions
Buffer[1]	K_SetDMABuf K_SetDMABufB
Number of Samples	K_SetDMABuf K_SetDMABufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn K_StartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStopG
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig K_SetDITrig
Trigger Polarity and Sensitivity	K_SetADTrig K_SetDITrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst

Notes

[1] This element must be set.

[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.

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

4. Use the **K_DMAStart** function to start the DMA-mode operation.
5. Use the **K_DMAStatus** function to monitor the status of the DMA-mode operation.
6. *If you specified continuous buffering mode*, use the **K_DMAStop** function to stop the DMA-mode operation when the appropriate number of samples has been acquired.
7. *If you are programming in Visual Basic for Windows*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to the program's local array.
8. Use the **DASDLL_DMAFree** function to deallocate the buffers.
9. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Analog Output Operations

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

Single Mode

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

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

Synchronous Mode

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

1. Use the **K_GetDAFrame** function to access a D/A frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the data to be written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-8.

Note: When you access a new D/A 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 3-2 on page 3-6 for a list of the default values of D/A frame elements.

Table 3-8. Setup Functions for Synchronous-Mode Analog Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig

Table 3-8. Setup Functions for Synchronous-Mode Analog Output Operations (cont.)

Attribute	Setup Functions
Trigger Type	K_SetDITrig
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes
[1] This element must be set.

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

4. *If you are programming in Visual Basic for Windows*, use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
5. Use the **K_SyncStart** function to start the synchronous-mode operation.
6. Use the **K_SyncFree** function to deallocate the buffer.
7. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt Mode

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

1. Use the **K_GetDAFrame** function to access a D/A frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the data to be written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-9.

Note: When you access a new D/A 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 3-2 on page 3-6 for a list of the default values of D/A frame elements.

Table 3-9. Setup Functions for Interrupt-Mode Analog Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetDITrig
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes
[1] This element must be set.
[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.

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

4. *If you are programming in Visual Basic for Windows*, use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
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 written.
8. Use the **K_SyncFree** function to deallocate the buffers.
9. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

DMA Mode

For a DMA-mode analog output operation, perform the following tasks:

1. Use the **K_GetDAFrame** function to access a D/A frame.
2. Use the **DASDLL_DMAAlloc** function to allocate the buffers in which to store the data to be written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-10.

Note: When you access a new D/A 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 3-2 on page 3-6 for a list of the default values of D/A frame elements.

Table 3-10. Setup Functions for DMA-Mode Analog Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetDMABuf K_SetDMABufB
Number of Samples	K_SetDMABuf K_SetDMABufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetDITrig
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes
[1] This element must be set.
[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.

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

4. *If you are programming in Visual Basic for Windows*, use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
5. Use the **K_DMAStart** function to start the DMA-mode operation.

6. Use the **K_DMAStatus** function to monitor the status of the DMA-mode operation.
7. *If you specified continuous buffering mode*, use the **K_DMAStop** function to stop the DMA-mode operation when the appropriate number of samples has been written.
8. Use the **DASDLL_DMAFree** function to deallocate the buffers.
9. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Digital I/O Operations

The following subsections describe the operation-specific programming tasks required to perform single-mode, synchronous-mode, interrupt-mode, and DMA-mode digital I/O operations.

Single Mode

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.

Synchronous Mode

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

1. Use the **K_GetDIFrame** function to access a DI frame; use the **K_GetDOFrame** function to access a DO frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the data to be read or written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-7.

Note: When you access a new DI or DO 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 3-3 on page 3-7 for a list of the default values of DI frame elements. Refer to Table 3-4 on page 3-8 for a list of the default values of DO frame elements.

Table 3-11. Setup Functions for Synchronous-Mode Digital Input and Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Clock Source	K_SetClk
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetDITrig

Table 3-11. Setup Functions for Synchronous-Mode Digital Input and Output Operations (cont.)

Attribute	Setup Functions
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes
[1] This element must be set.

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

4. *If you are performing a digital output operation and you are programming in Visual Basic for Windows,* use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
5. Use the **K_SyncStart** function to start the synchronous-mode operation.
6. *If you are performing a digital input operation and you are programming in Visual Basic for Windows,* use the **K_MoveBufToArray** function to transfer the data from the allocated buffer to the program's local array.
7. Use the **K_SyncFree** function to deallocate the buffers.
8. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt Mode

For an interrupt-mode digital I/O operation, perform the following tasks:

1. Use the **K_GetDIFrame** function to access a DI frame; use the **K_GetDOFrame** function to access a DO frame.
2. Use the **K_SyncAlloc** function to allocate the buffers in which to store the data to be read or written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-12.

Note: When you access a new DI or DO 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 3-3 on page 3-7 for a list of the default values of DI frame elements. Refer to Table 3-4 on page 3-8 for a list of the default values of DO frame elements.

Table 3-12. Setup Functions for Interrupt-Mode Digital Input and Digital Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetBuf K_SetBufB
Number of Samples	K_SetBuf K_SetBufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetDITrig
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes

[1] This element must be set.

[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.

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

4. *If you are performing a digital output operation and you are programming in Visual Basic for Windows,* use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
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 written.
8. *If you are performing a digital input operation and you are programming in Visual Basic for Windows,* use the **K_MoveBufToArray** function to transfer the data from the allocated buffer to the program's local array.
9. Use the **K_SyncFree** function to deallocate the buffers.
10. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

DMA Mode

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

1. Use the **K_GetDIFrame** function to access a DI frame; use the **K_GetDOFrame** function to access a DO frame.
2. Use the **DASDLL_DMAAlloc** function to allocate the buffers in which to store the data to be read or written.
3. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-13.

Note: When you access a new DI or DO 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 3-3 on page 3-7 for a list of the default values of DI frame elements. Refer to Table 3-4 on page 3-8 for a list of the default values of DO frame elements.

Table 3-13. Setup Functions for DMA-Mode Digital Input and Digital Output Operations

Attribute	Setup Functions
Buffer[1]	K_SetDMABuf K_SetDMABufB
Number of Samples	K_SetDMABuf K_SetDMABufB
Buffering Mode	K_SetContRun K_ClrContRun[2]
Start Channel	K_SetChn K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Pacer Clock Rate[1]	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetDITrig
Trigger Channel	K_SetDITrig
Trigger Polarity and Sensitivity	K_SetDITrig

Notes

[1] This element must be set.

[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.

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

4. *If you are performing a digital output operation and you are programming in Visual Basic for Windows,* use the **K_MoveArrayToBuf** function to transfer the data from the program's local array to the allocated buffer.
5. Use the **K_DMAStart** function to start the DMA-mode operation.
6. Use the **K_DMAStatus** function to monitor the status of the DMA-mode operation.
7. *If you specified continuous buffering mode*, use the **K_DMAStop** function to stop the DMA-mode operation when the appropriate number of samples has been written.
8. *If you are performing a digital input operation and you are programming in Visual Basic for Windows,* use the **K_MoveBufToArray** function to transfer the data from the allocated buffer to the program's local array.
9. Use the **DASDLL_DMAFree** function to deallocate the buffers.
10. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Language-Specific Programming Information

This section provides programming information for each of the supported languages. Note that the compilation procedures for each language assumes that the paths and/or environment variables are set correctly.

Microsoft Visual C++ Language

The following sections contain information you need to allocate and assign memory buffers and to create a channel-gain queue when programming in Microsoft Visual C++, as well as language-specific information for Microsoft Visual C++.

Note: When programming in Microsoft Visual C++, proper typecasting may be required to avoid C++ type-mismatch warnings.

Allocating and Assigning Memory Buffers

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

Note: The code fragments assume that you are using DMA mode; the code for synchronous-mode and interrupt mode is identical, except that you use the appropriate synchronous-mode or interrupt-mode functions instead of the DMA-mode functions.

Allocating the Memory Buffers

You can use a single memory buffer or two memory buffers for synchronous-mode, interrupt-mode, and DMA-mode analog I/O and digital I/O operations.

The following code fragment illustrates how to use **DASDLL_DMAAlloc** to allocate two buffers of size Samples for the frame defined by hFrame and how to use **K_SetDMABuf** and **K_SetDMABufB** to assign the starting addresses of the buffers.

```
. . .
void far *AcqBufA;          //Declare pointer to first buffer
void far *AcqBufB;          //Declare pointer to second buffer
WORD hMemA;                 //Declare word for first memory handle
WORD hMemB;                 //Declare word for second memory handle
. . .
wDasErr = DASDLL_DMAAlloc (hFrame, Samples, &AcqBufA, &hMemA);
wDasErr = K_SetDMABuf (hFrame, AcqBufA, Samples);
wDasErr = DASDLL_DMAAlloc (hFrame, Samples, &AcqBufB, &hMemB);
wDasErr = K_SetDMABufB (hFrame, AcqBufB, Samples);
. . .
```

The following code illustrates how to use **DASDLL_DMAFree** to later free the allocated buffers, using the memory handles stored by **DASDLL_DMAAlloc**.

```
. . .
wDasErr = DASDLL_DMAFree (hMemA);
wDasErr = DASDLL_DMAFree (hMemB);
. . .
```

Accessing the Data

You access the data stored in an allocated buffer through pointer indirection. For example, assume that you want to display the first 10 samples of the first buffer described in the previous section (AcqBufA). The following code fragment illustrates how to access and display the data.

```
. . .
int far *pData;            //Declare a pointer called pData
. . .
pData = (int far *) AcqBufA; //Assign pData to 1st buffer
for (i = 0; i < 10; i++)
   printf ("Sample #%d %X", i, *(pData+i));
. . .
```

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 DAS-40G2 board 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**, **K_IntStart,** or **K_DMAStart**), 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.

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

Programming in Microsoft Visual C++

To program in Microsoft Visual C++, you need the following files; these files are provided in the DASDLL software package.

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DASDLL.DLL	Dynamic Link Library
DASDECL.H	Include file for C
DASDLL.H	Include file for C
DASDECL.HPP	Include file for C++
DASDLL.HPP	Include file for C++
DASIMP.LIB	DAS Shell Imports
DASDLL.LIB	DASDLL Imports

To create an executable file in Visual C++, perform the following steps:

1. Create a project file by choosing New from the Project menu. The project file should contain all necessary files, including *filename*.c, *filename*.rc, *filename*.def, DASIMP.LIB, and DASDLL.LIB, where *filename* indicates the name of your application program.
2. From the Project menu, choose Rebuild All FILENAME.EXE to create a stand-alone executable file (.EXE) that you can execute from within Windows.

Microsoft Visual Basic for Windows

The following sections contain information you need to allocate and assign memory buffers and to create a channel-gain queue when programming in Microsoft Visual Basic for Windows, as well as language-specific information for Microsoft Visual Basic for Windows.

Allocating and Assigning Memory Buffers

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

Note: The code fragments assume that you are using DMA mode; the code for synchronous-mode and interrupt mode is identical, except that you use the appropriate synchronous-mode or interrupt-mode functions instead of the DMA-mode functions.

Allocating the Memory Buffers

You can use a single memory buffer or two memory buffers for synchronous-mode, interrupt-mode, and DMA-mode analog I/O and digital I/O operations.

The following code fragment illustrates how to use **DASDLL_DMAAlloc** to allocate two buffers of size Samples for the frame defined by hFrame and how to use **K_SetDMABuf** and **K_SetDMABufB** to assign the starting addresses of the buffers.

```
. . .
Global AcqBufA As Long     ' Declare pointer to first buffer
Global AcqBufB As Long     ' Declare pointer to second buffer
Global hMemA As Integer    ' Declare integer for first memory handle
Global hMemB As Integer    ' Declare integer for second memory handle
. . .
wDasErr = DASDLL_DMAAlloc (hFrame, Samples, AcqBufA, hMemA)
wDasErr = K_SetDMABuf (hFrame, AcqBufA, Samples)
wDasErr = DASDLL_DMAAlloc (hFrame, Samples, AcqBufB, hMemB)
wDasErr = K_SetDMABuf (hFrame, AcqBufB, Samples)
. . .
```

The following code illustrates how to use **DASDLL_DMAFree** to later free the allocated buffers, using the memory handles stored by **DASDLL_DMAAlloc**.

```
. . .
wDasErr = DASDLL_DMAFree (hMemA)
wDasErr = DASDLL_DMAFree (hMemB)
. . .
```

Accessing the Data

In Microsoft Visual Basic for Windows, you cannot directly access samples stored in an allocated memory buffer. For analog input operations, you must use **K_MoveBufToArray** to move a subset of the data into the program's local array as required. The following code fragment illustrates how to move the first 100 samples of the first buffer described in the previous section (AcqBufA) to the program's local array.

```
. . .
Dim Buffer(1000) As Integer   ' Declare local memory buffer
. . .
wDasErr = K_MoveBufToArray (Buffer(0), AcqBufA, 100)
. . .
```

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 DASDLL Function Call Driver.

The following code fragment illustrates how to create a four-entry channel-gain queue called MyChanGainQueue for a DAS-16G2 board and how to use **K_SetChnGAry** to assign the starting address of MyChanGainQueue to the frame defined by hFrame.

```
. . .
Global MyChanGainQueue(9) As Integer 'Maximum # of entries
. . .
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 the channel-gain queue is formatted, your Visual Basic for Windows program can no longer read it. 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**, **K_IntStart,** or **K_DMAStart**), 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.

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.

```
. . .
DASErr = K_GetDevHandle (hDrv, BoardNum, hDev)
If (DASErr <> 0) Then
   MsgBox “K_GetDevHandle Error: “ + Hex$ (DASErr),
         MB_ICONSTOP, “DASDLL 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 DASDLL software package.

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DASDLL.DLL	Dynamic Link Library
DASDECL.BAS	Include file; must be added to the Project List
DASDLL.BAS	Include file; must be added to the Project List

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

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
- 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	K_OpenDriver	page 4-83
	K_CloseDriver	page 4-18
	DASDLL_DevOpen	page 4-7
	K_GetDevHandle	page 4-54
	K_FreeDevHandle	page 4-35
	DASDLL_GetDevHandle	page 4-13
	DASDLL_GetBoardName	page 4-12
	K_DASDevInit	page 4-21
Operation	K_ADRead	page 4-15
	K_DAWrite	page 4-22
	K_DIRead	page 4-24
	K_DOWrite	page 4-32
	K_DMAStart	page 4-26
	K_DMAStatus	page 4-27
	K_DMAStop	page 4-30
	K_IntStart	page 4-73
	K_IntStatus	page 4-74
	K_IntStop	page 4-77
	K_SyncStart	page 4-115
Frame Management	K_GetADFrame	page 4-37
	K_GetDAFrame	page 4-52
	K_GetDIFrame	page 4-56
	K_GetDOFrame	page 4-58
	K_FreeFrame	page 4-36
	K_ClearFrame	page 4-17

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Memory Management	DASDLL_DMAAlloc	page 4-9
	DASDLL_DMAFree	page 4-11
	K_SyncAlloc	page 4-112
	K_SyncFree	page 4-114
	K_MoveArrayToBuf	page 4-79
	K_MoveBufToArray	page 4-81
Buffer Address	K_SetBuf	page 4-88
	K_SetBufB	page 4-90
	K_GetBuf	page 4-40
	K_GetBufB	page 4-42
	K_SetDMABuf	page 4-101
	K_SetDMABufB	page 4-103
Buffering Mode	K_SetContRun	page 4-99
	K_ClrContRun	page 4-19
	K_GetContRun	page 4-50

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Channel and Gain	K_SetChn	page 4-92
	K_SetStartStopChn	page 4-106
	K_SetG	page 4-105
	K_SetStartStopG	page 4-108
	K_SetChnGAry	page 4-93
	K_FormatChnGAry	page 4-34
	K_RestoreChnGAry	page 4-85
	K_GetChn	page 4-44
	K_GetStartStopChn	page 4-65
	K_GetG	page 4-61
	K_GetStartStopG	page 4-67
	K_GetChnGAry	page 4-45
Clock	K_SetClk	page 4-95
	K_SetClkRate	page 4-97
	K_GetClk	page 4-46
	K_GetClkRate	page 4-48
Trigger	K_SetTrig	page 4-110
	K_SetADTrig	page 4-86
	K_GetTrig	page 4-69
	K_GetADTrig	page 4-38
Miscellaneous	K_GetErrMsg	page 4-60
	K_GetVer	page 4-71
	K_GetShellVer	page 4-63

Keep the following conventions in mind throughout this chapter:

- If DAS-8 Series, DAS-16 Series, DAS-40 Series, PIO Series, or PDMA Series is listed in the Boards Supported section, all boards in the series are supported. For Series 500, refer to your Series 500 documentation for information on which specific Series 500 modules are supported for a particular function.
- The data types DWORD, WORD, and BYTE are defined in the language-specific include files.
- Variable names are shown in italics.
- For valid value and value stored information, refer to the board's user's guide and the External DAS Driver user's guide for that board.
- 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 they are defined as shown in the syntax. 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.

DASDLL_DevOpen

Boards Supported

All

Purpose

Opens the driver and returns the number of boards found.

Prototype

Visual C++

DASErr far pascal DASDLL_DevOpen (char far * *szCfgFile*, char far ***pBoards*);

Visual Basic for Windows

Declare Function DASDLL_DevOpen Lib "DASDLL.DLL" (ByVal *szCfgFile* As String, *pBoards* As Integer) As Integer

Parameters

szCfgFile	Driver configuration file.
pBoards	Number of boards found.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function opens the DASDLL Function Call Driver and stores the number of boards found in *pBoards*.

The DASDLL Function Call Driver does not use a configuration file. It is recommended that you enter a NULL string for *szCfgFile.*

See Also

K_OpenDriver

DASDLL_DevOpen (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
#include "DASDLL.H"    // Use DASDLL.HPP for C++
...
char nBoards;
...
wDasErr = DASDLL_DevOpen ("", &nBoards);
```

Visual Basic for Windows

(Include DASDECL.BAS and DASDLL.BAS in your program make file)

```
...
DIM nBoards AS INTEGER
...
wDasErr = DASDLL_DevOpen ("", nBoards)
```

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Allocates a buffer for a DMA-mode operation.

Prototype

Visual C++

DASErr far pascal DASDLL_DMAAlloc (DWORD *hFrame*, DWORD *dwSamples*, void far * far **pBuf*, WORD far * *phMem*);

Visual Basic for Windows

Declare Function DASDLL_DMAAlloc Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *dwSamples* As Long, *pBuf* As Long, *phMem* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
dwSamples	Number of samples. Valid values: **1** to **32767**
pBuf	Starting address of the allocated buffer.
phMem	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 memory block (a buffer of the size *dwSamples*) from the available memory heap. On return, *pBuf* contains the address of a buffer that is suitable for a DMA-mode operation and *phMem* contains the handle associated with the allocated buffer.

Use **K_SetDMABuf** or **K_SetDMABufB** to assign *pBuf* to a frame. You can use *phMem* to free the allocated memory block by calling **DASDLL_DMAFree**.

See Also

DASDLL_DMAFree, K_SetDMABuf, K_SetDMABufB

DASDLL_DMAAlloc (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
Global hMem As Integer
...
wDasErr = DASDLL_DMAAlloc (hFrame, dwSamples, pBuf, hMem)
```

DASDLL_DMAFree

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Frees a buffer allocated for a DMA-mode operation.

Prototype

Visual C++

DASErr far pascal DASDLL_DMAFree (WORD *hMem*);

Visual Basic for Windows

Declare Function DASDLL_DMAFree Lib "DASSHELL.DLL"
(ByVal *hMem* As Integer) As Integer

Parameters

hMem	Handle to DMA buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

DASDLL_DMAAlloc, K_SetDMABuf, K_SetDMABufB

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = DASDLL_DMAFree (hMem)
```

DASDLL_GetBoardName

Boards Supported

All

Purpose

Returns information about the boards and drivers loaded in your system.

Prototype

Visual C++

DASErr far pascal DASDLL_GetBoardName (WORD *nBrdNum*, char far *far* *pDrvName*);

Visual Basic for Windows

Not supported

Parameters

nBrdNum	Logical board number.
pDrvName	Driver associated with board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function gets the name of the driver associated with the board specified by *nBrdNum* and stores the name in *pDrvName.*

See Also

K_GetDevHandle, DASDLL_GetDevHandle

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
char *pDrvName;
...
wDasErr = DASDLL_GetBoardName (0, &pDrvName);
```

DASDLL_GetDevHandle

Boards Supported

All

Purpose

Initializes a DASDLL-supported board.

Prototype

Visual C++

DASErr far pascal DASDLL_GetDevHandle (WORD *nBrdNum*, DWORD far * *phDev*);

Visual Basic for Windows

Declare Function DASDLL_GetDevHandle Lib "DASDLL.DLL" (ByVal *nBrdNum* As Integer, *phDev* As Long) As Integer

Parameters

nBrdNum	Logical board number.
phDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the board specified by *nBrdNum*, and stores the board handle of the specified board in *phDev*.

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

See Also

K_GetDevHandle, DASDLL_GetBoardName, K_DASDevInit

DASDLL_GetDevHandle (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
#include "DASDLL.H"    // Use DASDLL.HPP for C++
...
void far *phDev;
...
wDasErr = DASDLL_GetDevHandle (0, &phDev);
```

Visual Basic for Windows

(Include DASDECL.BAS and DASDLL.BAS in your program make file)

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

K_ADRead

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20 , DAS-40 Series, DAS-HRES, Series 500

Purpose

Reads a single analog input value.

Prototype

Visual 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

Parameters

hDev	Handle associated with the board.
nChan	Analog input channel.
nGain	Gain code.
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 board specified by *hDev* at the gain represented by *nGain*, and stores the count in *pData*.

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

Refer to your External DAS Driver user's guide for a description of the data that can be stored in *pData.*

Refer to Appendix C for board-specific operating specifications on gains and channels.

See Also

K_DMAStart, K_IntStart, K_SyncStart

K_ADRead (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Sets the elements of a frame to their default values.

Prototype

Visual 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

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.

See Also

K_GetADFrame, K_GetDAFrame, K_GetDIFrame, K_GetDOFrame

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_ClearFrame (hFrame)
```

K_CloseDriver

Boards Supported

All

Purpose

Closes a previously initialized Keithley DAS Function Call Driver.

Prototype

Visual 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

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 board 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 and unloaded.

See Also

K_OpenDriver

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_CloseDriver (hDrv)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Enables single-cycle buffering mode.

Prototype

Visual 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

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, **K_GetDAFrame**, **K_GetDIFrame**, **K_GetDOFrame,** and **K_ClearFrame** also enable single-cycle buffering mode.

This function is not meaningful for synchronous-mode operations.

For more information on buffering modes, refer to page 2-14 (for analog input operations), page 2-23 (for analog output operations), and page 2-31 (for digital I/O operations).

See Also

K_SetContRun, K_GetContRun

K_ClrContRun (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_ClrContRun (hFrame)
```

Boards Supported

All

Purpose

Reinitializes a board.

Prototype

Visual 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

Parameters

hDev Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stops all current operations and resets the board specified by *hDev* and the driver to their power-up states.

See Also

K_GetDevHandle, DASDLL_GetDevHandle

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_DASDevInit (hDev)
```

K_DAWrite

Boards Supported

DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, DDA-06, Series 500

Purpose

Writes a single analog output value.

Prototype

Visual C++

DASErr far pascal K_DAWrite (DWORD *hDev*, BYTE *nChan*, DWORD *dwData*);

Visual Basic for Windows

Declare Function K_DAWrite Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, ByVal *nChan* As Integer,
ByVal *dwData* As Long) As Integer

Parameters

hDev	Handle associated with the board.
nChan	Analog output channel.
dwData	Analog output value.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function writes the value *dwData* to the analog output channel specified by *nChan* on the board specified by *hDev.*

Refer to Table C-3 in Appendix C for supported channels.

Refer to Appendix B for information on converting data for analog output operations.

Refer to your External DAS Driver user's guide for a description of the data that can be stored in *dwData.*

Refer to page 2-17 for more information on analog output operations.

See Also

K_DMAStart, K_IntStart, K_SyncStart

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD dwDAValue;
...
dwDAValue = ((DWORD) (5.0 * 4096 / 20) + 2048) << 4;
wDasErr = K_DAWrite (hDev, 0, &dwDAValue);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global dwDAValue As Long
...
dwDAValue = (INT (5.0 * 4096! / 20!) + 2048) * 16
wDasErr = K_DAWrite (hDev, 0, dwDAValue)
```

K_DIRead

Boards Supported

All

Purpose

Reads a single digital input value.

Prototype

Visual 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

Parameters

hDev	Handle associated with the board.
nChan	Digital input channel.
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 channel specified by *nChan* on the board specified by *hDev* and stores the value in *pData*.

Refer to your External DAS Driver user's guide for a description of the data that can be stored in *pData*.

See Also

K_IntStart, K_SyncStart

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

K_DMAStart

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Starts a DMA-mode operation.

Prototype

Visual C++

DASErr far pascal K_DMAStart (DWORD *hFrame*);

Visual Basic for Windows

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

Parameters

hFrame Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function starts the DMA operation defined by *hFrame*.

For a discussion of the programming tasks associated with DMA-mode operations, refer to page 3-15 (for analog input operations), page 3-21 (for analog output operations), and page 3-27 (for digital I/O operations).

See Also

K_SyncStart, K_DMAStatus, K_DMAStop

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_DMAStart (hFrame)
```

K_DMAStatus

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Gets status of a DMA-mode operation.

Prototype

Visual C++

DASErr far pascal K_DMAStatus (DWORD *hFrame*, short far **pStatus*, DWORD far **pCount*);

Visual Basic for Windows

Declare Function K_DMAStatus Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pStatus* As Integer, *pCount* As Long)
As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of DMA-mode operation; see **Remarks** for value stored.
pCount	Number of samples in the current buffer.

Return Value

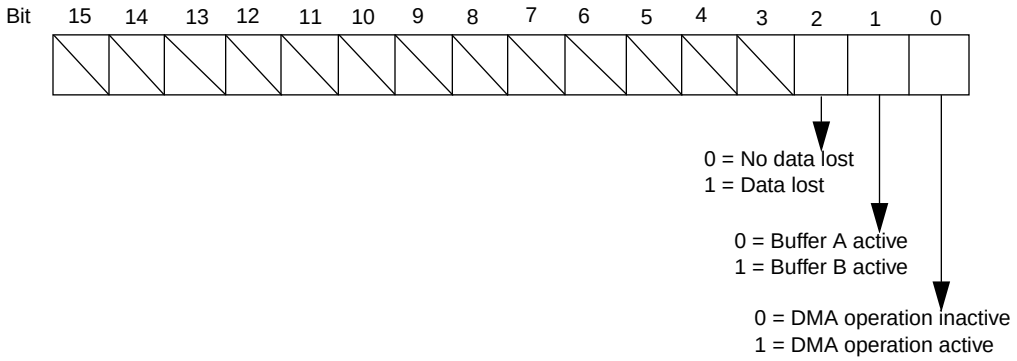
Error/status code. Refer to Appendix A.

K_DMAStatus (cont.)

Remarks

For the DMA operation defined by *hFrame*, this function stores the status in *pStatus* and the number of samples acquired in *pCount*.

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 a DMA-mode operation is in progress.
- Bit 1: If you are using two buffers, indicates which buffer is active. If you are using one buffer, this bit is always 0.
- Bit 2: If you are using two buffers, indicates whether data was lost when switching from one buffer to the other.
- Bits 3 through 15: Not used.

See Also

K_DMAStart, K_DMAStop

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wStatus As Integer
Global dwCount As Long
...
wDasErr = K_DMAStatus (hFrame, wStatus, dwCount)
```

K_DMAStop

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Stops a DMA-mode operation.

Prototype

Visual C++

DASErr far pascal K_DMAStop (DWORD *hFrame*, short far **pStatus*, DWORD far **pCount*);

Visual Basic for Windows

Declare Function K_DMAStop Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pStatus* As Integer, *pCount* As Long)
As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of DMA-mode operation; see **Remarks** for **K_DMAStatus** on page 4-28 for the value stored.
pCount	Number of samples in the current buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stops the DMA operation defined by *hFrame* and stores the status of the DMA operation in *pStatus* and the number of samples acquired in *pCount*.

If a DMA operation is not in progress, **K_DMAStop** is ignored.

See Also

K_DMAStart, K_DMAStatus

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wStatus As Integer
Global dwCount As Long
...
wDasErr = K_DMAStop (hFrame, wStatus, dwCount)
```

K_DOWrite

Boards Supported

All

Purpose

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

Prototype

Visual 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

Parameters

hDev	Handle associated with the board.
nChan	Digital output channel.
dwData	Digital output value.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function writes the value *dwData* to the digital output lines on the channel specified by *nChan* on the board specified by *hDev.*

Refer to your External DAS Driver user's guide for a description of the data that can be stored in *dwData.*

See Also

K_IntStart, K_SyncStart

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

K_FormatChnGAry

Boards Supported
DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES

Purpose
Converts the format of a channel-gain queue.

Prototype

Visual C++
Not supported

Visual Basic for Windows
Declare Function K_FormatChnGAry Lib "DASSHELL.DLL" (*pArray* As Integer) 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 using double-byte (16-bit) values to a channel-gain queue of single-byte (8-bit) values that the **K_SetChnGAry** function can use.

After you use this function, your program can no longer read the converted list. You must use the **K_RestoreChnGAry** function to return the list to its original format.

See Also
K_SetChnGAry, K_RestoreChnGAry

Usage

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)

```
...
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))
```

Boards Supported

All

Purpose

Frees a previously specified board handle.

Prototype

Visual 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

Parameters

hDev Board handle you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_GetDevHandle

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_FreeDevHandle (hDev)
```

K_FreeFrame

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Frees a frame.

Prototype

Visual 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

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, K_GetDAFrame, K_GetDIFrame, K_GetDOFrame

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_FreeFrame (hFrame)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

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

Prototype

Visual C++

DASErr far pascal K_GetADFrame (DWORD *hDev*,
DWORD far * *pFrame*);

Visual Basic for Windows

Declare Function K_GetADFrame Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *pFrame* As Long) As Integer

Parameters

hDev	Handle associated with the board.
pFrame	Handle to the frame that defines the operation.

Remarks

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

See Also

K_ClearFrame, K_FreeFrame

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

K_GetADTrig

Boards Supported

Series 500

Purpose

Gets the current analog trigger conditions.

Prototype

Visual C++

DASErr far pascal K_GetADTrig (DWORD *hFrame*, short far **pOpt*, short far **pChan*, DWORD far **pLevel*);

Visual Basic for Windows

Declare Function K_GetADTrig Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pOpt* As Integer, *pChan* As Integer, *pLevel* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pOpt	Analog trigger polarity and sensitivity. Valid values: **0** for Positive edge **2** for Negative edge
pChan	Analog input channel used as trigger channel.
pLevel	Level at which the trigger event occurs.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the channel used for an analog trigger in *pChan*, the level used for the analog trigger in *pLevel*, and the trigger polarity and sensitivity in *pOpt*.

The *pOpt* variable contains the value of the Trigger Polarity element; the *pChan* variable contains the value of the Trigger Channel element; the *pLevel* variable contains the value of the Trigger Level element.

The value of *pLevel* is represented as a count value between 0 and 8191, where 0 represents –10 V and 8181 represents +10 V.

See Also

K_SetADTrig, K_GetTrig

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
int nOpt, nChan;
DWORD dwLevel;
...
wDasErr = K_GetADTrig (hFrame, &nOpt, &nChan, &dwLevel);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global nOpt As Integer
Global nChan As Integer
Global dwLevel As Long
...
wDasErr = K_GetADTrig (hFrame, nOpt, nChan, dwLevel)
```

K_GetBuf

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Gets the address and size of the first memory buffer assigned to a frame.

Prototype

Visual C++

DASErr far pascal K_GetBuf (DWORD *hFrame*, void far * far **pBuf*, DWORD far **pSamples*);

Visual Basic for Windows

Declare Function K_GetBuf Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pBuf* As Long, *pSamples* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer.
pSamples	Number of samples.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation specified by *hFrame*, this function stores the address of the first memory buffer in *pBuf* and the number of samples stored in that buffer in *pSamples*.

Use this function to get the address of a synchronous-mode, interrupt-mode, or DMA-mode memory buffer whose address was specified by **K_SetBuf** or **KSetDMABuf**.

The *pBuf* variable contains the value of the Buffer element; the *pSamples* variable contains the value of the Number of Samples element.

See Also

K_GetBufB, K_SetBuf

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pADBuffer;
DWORD dwSamples;
...
wDasErr = K_GetBuf (hFrame, &pADBuffer, &dwSamples);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Dim pADBuffer As Long
...
wDasErr = K_GetBuf (hFrame, pADBuffer, dwSamples)
```

K_GetBufB

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Gets the address and size of the second memory buffer assigned to a frame.

Prototype

Visual C++

DASErr far pascal K_GetBufB (DWORD *hFrame*, void far * far **pBuf*, DWORD far **pSamples*);

Visual Basic for Windows

Declare Function K_GetBufB Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pBuf* As Long, *pSamples* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer.
pSamples	Number of samples.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation specified by *hFrame*, this function stores the address of the second memory buffer in *pBuf* and the number of samples stored in that buffer in *pSamples*.

Use this function to get the address of an interrupt-mode or DMA-mode memory buffer whose address was specified by **K_SetBufB** or **K_SetDMABufB**. (Syncronous-mode operations do not support a second memory buffer.)

The *pBuf* variable contains the value of the Buffer element; the *pSamples* variable contains the value of the Number of Samples element.

See Also

K_GetBuf, K_SetBufB

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pADBuffer;
DWORD dwSamples;
...
wDasErr = K_GetBufB (hFrame, &pADBuffer, &dwSamples);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Dim pADBuffer As Long
...
wDasErr = K_GetBufB (hFrame, pADBuffer, dwSamples)
```

K_GetChn

Boards Supported
DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose
Gets a single channel number.

Prototype

Visual C++
DASErr far pascal K_GetChn (DWORD *hFrame*, short far **pChan*);

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

Parameters

hFrame	Handle to the frame that defines the operation.
pChan	Channel on which to perform the operation.

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

Remarks
For the operation defined by *hFrame*, this function stores the channel number in *pChan*.

The *pChan* variable contains the value of the Start Channel element.

See Also
K_SetChn, K_GetStartStopChn, K_GetStartStopG

Usage

Visual C++
```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
short nChan;
...
wDasErr = K_GetChn (hFrame, &nChan);
```

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)
```
...
Global nChan AS Integer
...
wDasErr = K_GetChn (hFrame, nChan)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES

Purpose

Gets the starting address of a channel-gain queue.

Prototype

Visual C++

DASErr far pascal K_GetChnGAry (DWORD *hFrame*,
void far * far **pArray*);

Visual Basic for Windows

Not supported

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 stores the starting address of the channel-gain queue in *pArray*.

The *pArray* variable contains the value of the Channel-Gain Queue element.

Refer to page 2-11 for information on setting up a channel-gain queue.

See Also

K_SetChnGAry

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pArray;
...
wDasErr = K_GetChnGAry (hFrame, &pArray);
```

K_GetClk

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PDMA Series

Purpose

Gets the pacer clock source.

Prototype

Visual C++

DASErr far pascal K_GetClk (DWORD *hFrame*, short far **pMode*);

Visual Basic for Windows

Declare Function K_GetClk Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pMode* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pMode	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 stores the pacer clock source in *pMode*.

An internal clock source is the output of the onboard counter; an external clock source is an external signal connected to the appropriate pin.

For more information about pacer clock sources, refer to page 2-6 (for analog input operations), page 2-17 (for analog output operations), and page 2-25 (for digital I/O operations).

The *pMode* variable contains the value of the Clock Source element.

See Also

K_SetClk, K_GetClkRate

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
Word wMode;
...
wDasErr = K_GetClk (hFrame, &wMode);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wMode As Integer
...
wDasErr = K_GetClk (hFrame, wMode)
```

K_GetClkRate

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PDMA Series

Purpose

Gets the number of clock ticks used by the internal pacer clock.

Prototype

Visual 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

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 used by the internal pacer clock in *pRate*.

After a synchronous-mode, interrupt-mode, or DMA-mode operation, 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, K_GetClk

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

K_GetContRun

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Gets the buffering mode.

Prototype

Visual C++

DASErr far pascal K_GetContRun (DWORD *hFrame*, short far **pMode*);

Visual Basic for Windows

Declare Function K_GetContRun Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pMode* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pMode	Buffering mode. Valid values: **0** for Single-cycle mode **1** for Continuous mode

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the buffering mode in *pMode*.

For a description of buffering modes, refer to page 2-14 (for analog input operations), page 2-23 (for analog output operations), and page 2-31 (for digital I/O operations).

The *pMode* variable contains the value of the Buffering Mode element.

See Also

K_SetContRun, K_ClrContRun

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wMode;
...
wDasErr = K_GetContRun (hFrame, &wMode);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wMode As Integer
...
wDasErr = K_GetContRun (hFrame, wMode)
```

K_GetDAFrame

Boards Supported

DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Accesses a D/A frame for an analog output operation.

Prototype

Visual C++

DASErr far pascal K_GetDAFrame (DWORD *hDev*,
DWORD far * *pFrame*);

Visual Basic for Windows

Declare Function K_GetDAFrame Lib "DASSHELL.DLL"
(ByVal *hDev* As Long, *pFrame* As Long) As Integer

Parameters

hDev	Handle associated with the board.
pFrame	Handle to the frame that defines the D/A operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function specifies that you want to perform a synchronous-mode, interrupt-mode, or DMA-mode analog output operation on the board specified by *hDev*, and accesses an available D/A frame with the handle *pFrame*.

See Also

K_ClearFrame, K_FreeFrame

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global hDA As Long
...
wDasErr = K_GetDAFrame (hDev, hDA)
```

K_GetDevHandle

Boards Supported

All

Purpose

Initializes any Keithley DAS board.

Prototype

Visual C++

DASErr far pascal K_GetDevHandle (DWORD *hDrv*, WORD *nBrdNum*, DWORD far * *pDev*);

Visual Basic for Windows

Declare Function K_GetDevHandle Lib "DASSHELL.DLL" (ByVal *hDrv* As Long, ByVal *nBrdNum* As Integer, *pDev* As Long) As Integer

Parameters

hDrv	Driver handle of the associated Function Call Driver.
nBrdNum	Logical board number.
pDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the board associated with *hDrv* and specified by *nBrdNum*, and stores the board handle of the specified board in *pDev*.

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

See Also

K_FreeDevHandle, DASDLL_GetDevHandle, DASDLL_GetBoardName, K_DASDevInit

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

K_GetDIFrame

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Accesses a DI frame for a digital input operation.

Prototype

Visual C++

DASErr far pascal K_GetDIFrame (DWORD *hDev*, DWORD far * *pFrame*);

Visual Basic for Windows

Declare Function K_GetDIFrame Lib "DASSHELL.DLL" (ByVal *hDev* As Long, *pFrame* As Long) As Integer

Parameters

hDev	Handle associated with the board.
pFrame	Handle to the frame that defines the digital input operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function specifies that you want to perform a synchronous-mode, interrupt-mode, or DMA-mode digital input operation on the board specified by *hDev*, and accesses an available digital input frame with the handle *pFrame*.

See Also

K_ClearFrame, K_FreeFrame

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global hDI As Long
...
wDasErr = K_GetDIFrame (hDev, hDI)
```

K_GetDOFrame

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Accesses a DO frame for a digital output operation.

Prototype

Visual C++

DASErr far pascal K_GetDOFrame (DWORD *hDev*, DWORD far * *pFrame*);

Visual Basic for Windows

Declare Function K_GetDOFrame Lib "DASSHELL.DLL" (ByVal *hDev* As Long, *pFrame* As Long) As Integer

Parameters

hDev	Handle associated with the board.
pFrame	Handle to the frame that defines the digital output operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function specifies that you want to perform a synchronous-mode, interrupt-mode, or DMA-mode digital output operation on the board specified by *hDev*, and accesses an available digital output frame with the handle *pFrame*.

See Also

K_ClearFrame, K_FreeFrame

Usage

Visual C++

```
#include “DASDECL.H”   // Use DASDECL.HPP for C++
...
DWORD hDO;
...
wDasErr = K_GetDOFrame (hDev, &hDO);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global hDO As Long
...
wDasErr = K_GetDOFrame (hDev, hDO)
```

K_GetErrMsg

Boards Supported

All

Purpose

Gets the address of an error message string.

Prototype

Visual C++

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

Visual Basic for Windows

Not supported

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board specified by *hDev*, this function stores the address of the string corresponding to error message number *nDASErr* in *pErrMsg*.

Refer to page 2-5 for more information about error handling. Refer to Appendix A for a list of error codes and their meanings.

Usage

Visual C++

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

Boards Supported

DAS-8PGA, DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Gets the gain.

Prototype

Visual C++

DASErr far pascal K_GetG (DWORD *hFrame*, short far **pGain*);

Visual Basic for Windows

Declare Function K_GetG Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pGain* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pGain	Gain code.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the gain code for a single channel or for a group of consecutive channels in *pGain*.

Refer to Appendix C for specific operating specifications on gains and channels.

See Also

K_SetG, K_GetStartStopG

K_GetG (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wGain;
...
wDasErr = K_GetG (hFrame, &wGain);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wGain As Integer
...
wDasErr = K_GetG (hFrame, wGain)
```

K_GetShellVer

Boards Supported
All

Purpose
Gets the current DAS shell version.

Prototype

Visual 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

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
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).

See Also
K_GetVer

K_GetShellVer (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Gets the first and last channels in a group of consecutive channels.

Prototype

Visual C++

DASErr far pascal K_GetStartStopChn (DWORD *hFrame*,
short far **pStart*, short far **pStop*);

Visual Basic for Windows

Declare Function K_GetStartStopChn Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pStart* As Integer, *pStop* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pStart	First channel in a group of consecutive channels.
pStop	Last channel in a group of consecutive channels.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the first channel in a group of consecutive channels in *pStart* and the last channel in the group of consecutive channels in *pStop*.

The *pStart* variable contains the value of the Start Channel element; the *pStop* variable contains the value of the Stop Channel element.

See Also

K_SetStartStopChn, K_GetChn, K_GetStartStopG

K_GetStartStopChn (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
short nStart, nStop;
...
wDasErr = K_GetStartStopChn (hFrame, &nStart, &nStop);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global nStart As Integer
Global nStop As Integer
...
wDasErr = K_GetStartStopChn (hFrame, nStart, nStop)
```

Boards Supported

DAS-8PGA, DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Gets the first and last channels in a group of consecutive channels and the gain for all channels in the group.

Prototype

Visual C++

DASErr far pascal K_GetStartStopG (DWORD *hFrame*, short far **pStart*, short far **pStop*, short far **pGain*);

Visual Basic for Windows

Declare Function K_GetStartStopG Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, *pStart* As Integer, *pStop* As Integer, *pGain* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pStart	First channel in a group of consecutive channels.
pStop	Last channel in a group of consecutive channels.
pGain	Gain code.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the first channel in a group of consecutive channels in *pStart*, the last channel in the group of consecutive channels in *pStop*, and the gain code for all channels in the group in *pGain*.

Refer to Appendix C for the gain associated with the gain code.

The *pStart* variable contains the value of the Start Channel element; the *pStop* variable contains the value of the Stop Channel element; the *pGain* variable contains the value of the Gain element.

See Also

K_SetStartStopG, K_GetChn, K_GetStartStopChn

K_GetStartStopG (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
short nStart, nStop, nGain;
...
wDasErr = K_GetStartStopG (hFrame, &nStart, &nStop, &nGain);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global nStart As Integer
Global nStop As Integer
Global nGain As Integer
...
wDasErr = K_GetStartStopG (hFrame, nStart, nStop, nGain)
```

K_GetTrig

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Gets the trigger source.

Prototype

Visual C++

DASErr far pascal K_GetTrig (DWORD *hFrame*, short far **pMode*);

Visual Basic for Windows

Declare Function K_GetTrig Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, *pMode* As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pMode	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 stores the trigger source in *pMode*.

The *pMode* variable contains the value of the Trigger Source element.

An internal trigger is a software trigger. An external trigger is either an analog trigger or a digital trigger. For more information about trigger sources, refer to page 2-14 (for analog input operations), page 2-24 (for analog output operations), and page 2-32 (for digital I/O operations.

See Also

K_SetTrig

K_GetTrig (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
WORD wMode;
...
wDasErr = K_GetTrig (hFrame, &wMode);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global wMode As Integer
...
wDasErr = K_GetTrig (hFrame, wMode)
```

Boards Supported

All

Purpose

Gets revision numbers.

Prototype

Visual 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

Parameters

hDev	Handle associated with the board.
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 board 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 2.1, the major revision level is 2 and the minor revision level is 1; therefore, the high byte of *pDrvVer* contains the value of **2** (512) and the low byte of *pDrvVer* contains the value of **1**; the value of both bytes is 513.

To extract the major and minor revision levels from the value stored in *pDrvVer* or *pSpecVer*, use the following equations:

$$\text{major revision level} = \text{Integer portion of } \left(\frac{\text{returned value}}{256}\right)$$

$$\text{minor revision level} = \text{returned value MOD } 256$$

See Also K_GetShellVer

Usage

Visual 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

(Include DASDECL.BAS in your program make file)

```
...
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$
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Starts an interrupt-mode operation.

Prototype

Visual 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

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*.

For a discussion of the programming tasks associated with interrupt-mode operations, refer to page 3-13 (for analog input operations), page 3-19 (for analog output operations), and page 3-25 (for digital I/O operations).

See Also

K_IntStatus, K_IntStop

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_IntStart (hFrame)
```

K_IntStatus

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Gets status of interrupt-mode operation.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt-mode operation; see **Remarks** for value stored.
pCount	Number of samples that were acquired.

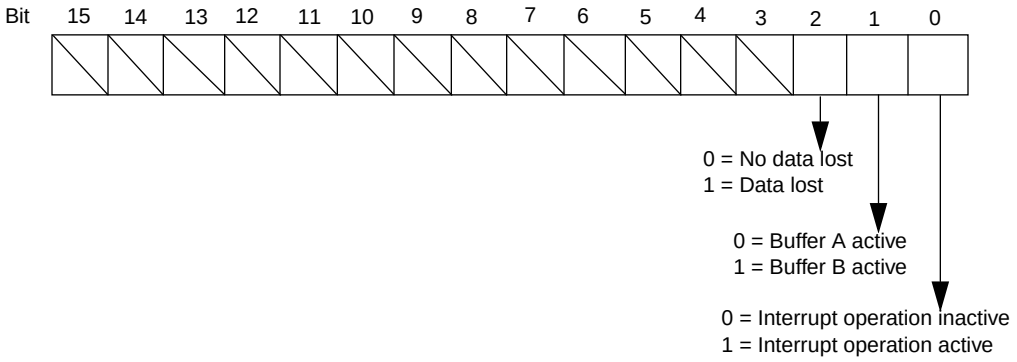
Return Value

Error/status code. Refer to Appendix A.

Remarks

For the interrupt operation defined by *hFrame*, this function stores the status in *pStatus* and the number of samples acquired in *pCount*.

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: If you are using two buffers, indicates which buffer is active. If you are using one buffer, this bit is always 0.
- Bit 2: If you are using two buffers, indicates whether data was lost when switching from one buffer to the other.
- Bits 3 through 15: Not used.

See Also

K_IntStart, K_IntStop

K_IntStatus (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Stops an interrupt-mode operation.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt operation; see **Remarks** for **K_IntStatus** on page 4-75 for the value stored.
pCount	Number of samples that were acquired.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stops the interrupt operation defined by *hFrame* and stores the status of the interrupt operation in *pStatus* and the number of samples acquired in *pCount*.

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

See Also

K_IntStart, K_IntStatus

K_IntStop (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

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

Boards Supported

DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Transfers data from the program's local array to a buffer allocated through **K_SyncAlloc** or **DASDLL_DMAAlloc**.

Prototype

Visual C++
Not supported

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

Parameters

pDest	Address of destination buffer.
pSource	Source array.
nCount	Number of samples to transfer. Valid values: **1** to **32767** (**0** = 32768)

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 buffer at address *pDest*.

The buffer used to store output data for your program is not accessible to the program; you must use this function to move the data from the program's local array to the allocated buffer.

See Also

DASDLL_DMAAlloc, K_SyncAlloc

K_MoveArrayToBuf (cont.)

Usage

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SyncAlloc ( hDA, dwSamples, pBuf, hMem )
...
wDasErr = K_MoveArrayToBuf ( pBuf, DACArray[0], dwSamples )
```

K_MoveBufToArray

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Transfers data from a buffer allocated through **K_SyncAlloc** or **DASDLL_DMAAlloc** to your program's local array.

Prototype

Visual 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

Parameters

pDest	Destination array.
pSource	Address of source buffer.
nCount	Number of samples to transfer. Valid values: **1** to **32767** (**0** = 32768)

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*.

The buffer used to store acquired data for your program is not accessible to your program; you must use this function to move the data from the allocated buffer to your program's local array.

See Also

DASDLL_DMAAlloc, K_SyncAlloc

K_MoveBufToArray (cont.)

Usage

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SyncAlloc ( hAD, dwSamples, pBuf, hMem )
...
wDasErr = K_MoveBufToArray ( ADArray[0], pBuf, dwSamples)
```

Boards Supported

All

Purpose

Initializes any Keithley DAS Function Call Driver.

Prototype

Visual 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

Parameters

szDrvName	Driver name. Valid value: **"DASDLL"** (for DASDLL-supported boards)
szCfgName	Driver configuration file.
pDrv	Handle associated with the driver.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DASDLL Function Call Driver and stores the driver handle in *pDrv*.

The DASDLL Function Call Driver does not use a configuration file. It is recommended that you enter a NULL string for *szCfgName.*

You can use this function to initialize the Function Call Driver associated with any Keithley MetraByte DAS board. For DASDLL-supported boards, the string stored in *szDrvName* must be DASDLL. Refer to other Function Call Driver user's guides for the appropriate string to store in *szDrvName* for other Keithley MetraByte DAS boards.

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.*

K_OpenDriver (cont.)

See Also K_CloseDriver, DASDLL_DevOpen

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
DIM hDrv As Long
...
wDasErr = K_OpenDriver ("DASDLL", "", hDrv)
```

Boards Supported DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES

Purpose Restores a converted channel-gain queue.

Prototype

Visual C++
Not supported

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

Parameters

pArray	Channel-gain queue starting address.

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

Remarks This function restores a channel-gain queue that was converted using **K_FormatChnGAry** to its original format so that it can be used by your Visual Basic for Windows program.

See Also K_FormatChnGAry, K_SetChnGAry

Usage

Visual Basic for Windows

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
Global ChanGainArray (16) As Integer
...
wDasErr = K_RestoreChnGAry (ChanGainArray (0))
```

K_SetADTrig

Boards Supported

Series 500

Purpose

Sets up an analog trigger.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
nOpt	Analog trigger polarity and sensitivity. Valid values: **0** for Positive edge **2** for Negative edge
nChan	Analog input channel used as trigger channel.
dwLevel	Level at which the trigger event occurs. Valid values: **0** to **8191**

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* as a count value between 0 and 8191, where 0 represents –10 V and 8191 represents +10 V.

Refer to Appendix C for board-specific operating specifications on channels.

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**, **K_IntStart**, or **K_DMAStart**.

See Also K_GetADTrig, K_SetTrig

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetADTrig (hFrame, 0, 0, 2047)
```

K_SetBuf

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Specifies the starting address of the first memory buffer used in synchronous mode or interrupt mode.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer.
dwSamples	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 the first memory buffer in *pBuf* and the number of samples (the size of the buffer) in *dwSamples*.

Use this function for synchronous mode and interrupt mode only. For DMA mode, use **K_SetDMABuf**.

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_SetBufB, K_GetBuf

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
...
wDasErr = K_SyncAlloc (hFrame, dwSamples, pBuf, hMem)
wDasErr = K_SetBuf (hFrame, pBuf, dwSamples)
```

K_SetBufB

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Specifies the starting address of the second memory buffer used in interrupt mode.

Prototype

Visual C++

DASErr far pascal K_SetBufB (DWORD *hFrame*, void far **pBuf*, DWORD *dwSamples*);

Visual Basic for Windows

Declare Function K_SetBufB Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *pBuf* As Long,
ByVal *dwSamples* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer.
dwSamples	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 the second memory buffer in *pBuf* and the number of samples (the size of the buffer) in *dwSamples*.

Use this function for interrupt mode only. For DMA mode, use **K_SetDMABufB**. (Syncronous-mode operations do not support a second memory buffer.)

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_SetBuf, K_GetBufB

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pBuf;   // Pointer to allocated buffer
...
wDasErr = K_SyncAlloc (hFrame, dwSamples, &pBuf, &hMem);
wDasErr = K_SetBufB (hFrame, pBuf, dwSamples);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
...
wDasErr = K_SyncAlloc (hFrame, dwSamples, pBuf, hMem)
wDasErr = K_SetBufB (hFrame, pBuf, dwSamples)
```

K_SetChn

Boards Supported
DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose
Specifies a single channel.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
nChan	Channel on which to perform operation.

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

Remarks
For the operation defined by *hFrame*, this function specifies the single channel used in *nChan*.

Refer to Appendix C for board-specific operating specifications on channels.

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

See Also
K_GetChn, K_SetStartStopChn, K_SetStartStopG

Usage

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

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)
```
...
wDasErr = K_SetChn (hFrame, 2)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES

Purpose

Specifies the starting address of a channel-gain queue.

Prototype

Visual 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

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*.

The *pArray* variable sets the value of the Channel-Gain Queue element.

Refer to page 2-11 for information on setting up a channel-gain queue.

Refer to Appendix C for board-specific information on channels and gains.

If you created your channel-gain queue in 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

K_SetChnGAry (cont.)

Usage

Visual 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 (hFrame, &ChanGainArray);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global ChanGainArray(16) 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 (hFrame, ChanGainArray(0))
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PDMA Series

Purpose

Specifies the pacer clock source.

Prototype

Visual 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

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 value of the Clock Source element.

The internal clock source is the output of the onboard counter; an external clock source is an external signal connected to the appropriate pin on the main I/O connector.

For more information about pacer clock sources, refer to page 2-6 (for analog input operations), page 2-17 (for analog output operations), and page 2-25 (for digital I/O operations).

K_GetADFrame, **K_GetDAFrame**, **K_GetDIFrame**, **K_GetDOFrame,** and **K_ClearFrame** specify internal as the default clock source.

See Also

K_GetClk, K_SetClkRate

K_SetClk (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetClk (hFrame, 1)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PDMA Series

Purpose

Specifies the number of clock ticks used by the internal pacer clock.

Prototype

Visual 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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the number of clock ticks used by the internal pacer clock in *dwDivisor.*

The *dwDivisor* variable sets the value of the Pacer Clock Rate element.

For more information about the pacer clock, refer to page 2-12 (for analog input operations), page 2-21 (for analog output operations), and page 2-30 (for digital I/O operations).

See Also

K_GetClkRate, K_SetClk

K_SetClkRate (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
DWORD dwClkDiv;
...
dwClkDiv = 1000000 / 10000;
wDasErr = K_SetClkRate (hFrame, dwClkDiv);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global dwClkDiv As Long
...
dwClkDiv = 1000000 / 10000
wDasErr = K_SetClkRate (hFrame, dwClkDiv)
```

K_SetContRun

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Enables continuous buffering mode.

Prototype

Visual 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

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, **K_GetDAFrame**, **K_GetDIFrame**, **K_GetDOFrame**, and **K_ClearFrame** enable single-cycle buffering mode.

For a description of buffering modes, refer to page 2-6 (for analog input operations), page 2-17 (for analog output operations) section, and page 2-25 (for digital I/O operations).

See Also

K_ClrContRun, K_GetContRun

K_SetContRun (cont.)

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetContRun (hFrame)
```

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, PDMA Series

Purpose

Specifies the starting address of the first memory buffer used in DMA mode.

Prototype

Visual C++

DASErr far pascal K_SetDMABuf (DWORD *hFrame*, void far **pBuf*, DWORD *dwSamples*);

Visual Basic for Windows

Declare Function K_SetDMABuf Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *pBuf* As Long,
ByVal *dwSamples* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the DMA-mode operation.
pBuf	Starting address of buffer.
dwSamples	Number of samples. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation specified by *hFrame*, this function stores the address of the first memory buffer in *pBuf* and the number of samples stored in the buffer in *dwSamples*.

Use this function for DMA mode only. For synchronous mode and interrupt mode, use **K_SetBuf**.

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

See Also

DASDLL_DMAAlloc, K_SetDMABufB

K_SetDMABuf (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pBuf;   // Pointer to allocated buffer
...
wDasErr = DASDLL_DMAAlloc (hFrame, dwSamples, &pBuf, &hMem);
wDasErr = K_SetDMABuf (hFrame, pBuf, dwSamples);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
...
wDasErr = DASDLL_DMAAlloc (hFrame, dwSamples, pBuf, hMem)
wDasErr = K_SetDMABuf (hFrame, pBuf, dwSamples)
```

Boards Supported

DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES

Purpose

Specifies the starting address of the second memory buffer used in DMA mode.

Prototype

Visual C++

DASErr far pascal K_SetDMABufB (DWORD *hFrame*, void far **pBuf*, DWORD *dwSamples*);

Visual Basic for Windows

Declare Function K_SetDMABufB Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *pBuf* As Long,
ByVal *dwSamples* As Long) As Integer

Parameters

hFrame	Handle to the frame that defines the DMA-mode operation.
pBuf	Starting address of buffer.
dwSamples	Number of samples. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation specified by *hFrame*, this function stores the address of the second memory buffer in *pBuf* and the number of samples stored in the buffer in *dwSamples*.

Use this function for DMA mode only. For interrupt mode, use **K_SetBufB**. (Syncronous-mode operations do not support a second memory buffer.)

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

See Also

DASDLL_DMAAlloc, K_SetDMABuf

K_SetDMABufB (cont.)

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
void far *pBuf;   // Pointer to allocated buffer
...
wDasErr = DASDLL_DMAAlloc (hFrame, dwSamples, &pBuf, &hMem);
wDasErr = K_SetDMABufB (hFrame, pBuf, dwSamples);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
...
wDasErr = DASDLL_DMAAlloc (hFrame, dwSamples, pBuf, hMem)
wDasErr = K_SetDMABufB (hFrame, pBuf, dwSamples)
```

Boards Supported
DAS-8PGA, DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose
Sets the gain.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
nGain	Gain code.

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*.

Refer to Appendix C for board-specific operating specifications on gains.

The *nGain* variable sets the value of the Gain element.

K_GetADFrame and **K_ClearFrame** specify a gain of 1 (gain code 0) as the default gain.

See Also
K_GetG, K_SetStartStopG

Usage

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

Visual Basic for Windows
(Include DASDECL.BAS in your program make file)
```
...
wDasErr = K_SetG (hFrame, 1)
```

K_SetStartStopChn

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Specifies the first and last channels in a group of consecutive channels.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels.
nStop	Last channel in a group of consecutive channels.

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*.

Refer to Appendix C for board-specific operating specifications on channels.

The *nStart* variable sets the value of the Start Channel element; the *nStop* variable sets the value of the Stop Channel element.

K_GetADFrame, **K_GetDAFrame**, **K_GetDIFrame**, **K_GetDOFrame** and **K_ClearFrame** set the Start Channel and Stop Channel elements to 0.

See Also

K_GetStartStopChn, K_SetChn, K_SetStartStopG

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetStartStopChn (hFrame, 0, 7);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetStartStopChn (hFrame, 0, 7)
```

K_SetStartStopG

Boards Supported

DAS-8PGA, DAS-8/AO, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Specifies the first and last channels in a group of consecutive channels and sets the gain for all channels in the group.

Prototype

Visual 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

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels.
nStop	Last channel in a group of consecutive channels.
nGain	Gain code.

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.

Refer to Appendix C for board-specific operating specifications on gains and channels.

K_GetADFrame and **K_ClearFrame** set the Start Channel, Stop Channel, and Gain elements to 0.

See Also K_GetStartStopG, K_SetChn, K_SetStartStopChn

Usage

Visual C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetStartStopG (hFrame, 0, 7, 0);
```

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetStartStopG (hFrame, 0, 7, 0)
```

K_SetTrig

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500

Purpose

Specifies the trigger source.

Prototype

Visual 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

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. An external trigger is either an analog trigger or a digital trigger. For more information about trigger sources, refer to page 2-14 (for analog input operations), page 2-24 (for analog output operations), and page 2-32 (for digital I/O operations).

For DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, and DAS-HRES boards, if *nMode* = **1**, the external trigger is a digital trigger. For Series 500 boards, if *nMode* = **1**, the external trigger is an analog trigger; use **K_SetADTrig** to specify the conditions for an external analog trigger.

K_GetADFrame, **K_GetDAFrame**, **K_GetDIFrame**, **K_GetDOFrame**, and **K_ClearFrame** set the trigger source to internal.

See Also

K_GetTrig

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SetTrig (hFrame, 1)
```

K_SyncAlloc

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

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

Prototype

Visual C++

DASErr far pascal K_SyncAlloc (DWORD *hFrame*,
DWORD *dwSamples*, void far * far **pBuf*, WORD far **pMem*);

Visual Basic for Windows

Declare Function K_SyncAlloc Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *dwSamples* As Long, *pBuf* As Long,
pMem As Integer) As Integer

Parameters

hFrame	Handle to the frame that defines the operation.
dwSamples	Number of samples. Valid values: **1** to **32767**
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*.

See Also

K_SyncFree, K_SetBuf, K_SetBufB

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
Global pBuf As Long
Global hMem As Integer
...
wDasErr = K_SyncAlloc (hFrame, dwSamples, pBuf, hMem)
```

K_SyncFree

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

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

Prototype

Visual C++

DASErr far pascal K_SyncFree (WORD *hMem*);

Visual Basic for Windows

Declare Function K_SyncFree Lib "DASSHELL.DLL"
(ByVal *hMem* As Integer) As Integer

Parameters

hMem Handle to memory buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_SyncAlloc

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SyncFree (hMem)
```

Boards Supported

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, Series 500, PIO Series, PDMA Series

Purpose

Starts a synchronous-mode operation.

Prototype

Visual 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

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 operation defined by *hFrame*.

For a discussion of the programming tasks associated with synchronous-mode operations, refer to page 3-11 (for analog input operations), page 3-18 (for analog output operations), and page 3-24 (for digital I/O operations).

See Also

K_DMAStart

Usage

Visual C++

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

Visual Basic for Windows

(Include DASDECL.BAS in your program make file)

```
...
wDasErr = K_SyncStart (hFrame)
```

A

Error/Status Codes

Error and status codes may be returned by either the DASDLL Function Call Driver or your External DAS Driver. Table A-1 lists the error/status codes that are returned by the DASDLL Function Call Driver, as well as possible causes for errors and possible solutions for resolving errors. Refer to your External DAS Driver user's guide for a list of the error/status codes returned by the External DAS Driver.

If you cannot resolve an error, contact the Keithley MetraByte Applications Engineering Department.

Table A-1. Error/Status Codes

Error Code		Cause	Solution
Hex	**Decimal**		
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 board's base I/O address in the configuration file is illegal and/or does not match the base address switches on the board.	Use the configuration utility to change the base I/O address to one that matches the base address switches on the board.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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 board. Refer to the External DAS Driver user's guide for the board for legal interrupt levels.
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 board. Refer to the External DAS Driver user's guide for legal DMA channels.
6005	24581	**Illegal channel number:** The specified channel number is illegal for the board and/or for the range type (unipolar or bipolar).	Specify a legal channel number. Refer to the External DAS Drivers user's guide or to Appendix C for legal channel numbers.
6006	24582	**Illegal gain code:** The specified analog I/O channel gain code is illegal for this board.	Specify a legal gain code. Refer to the External DAS Driver user's guide or to Appendix C 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 Appendix D 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.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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.
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 allocated because of its size.	Specify a smaller buffer size. 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.	Specify a smaller buffer size.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	**Decimal**		
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 Appendix D for information on how to install and set up the Keithley Memory Manager (VDMAD.386).
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.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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** for legal mode values.
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** in Windows Enhanced mode.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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 **DASDLL_GetDevHandle**, or it was corrupted by your program.	Check the device handle value.
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. Refer to Appendix D.
6031	24625	**Illegal memory handle:** A bad memory handle was passed to **K_IntFree, K_SyncFree,** or **K_DMAFree**. The handle used was not initialized through a call to **K_IntAlloc, K_SyncAlloc,** or **K_DMAAlloc**, or it was corrupted by you program.	Restart your program and monitor the memory handle value.
6032	24626	**Out of memory handles:** An attempt to allocate a memory block using **K_IntAlloc, K_SyncAlloc,** or **K_DMAAlloc** failed because the maximum number of handles has already been assigned.	Use **K_IntFree, K_SyncFree,** or **K_DMAFree** to free previously allocated memory blocks before allocating again.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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.
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.)	Make sure that you initialize a driver only once during a single Windows session. To continue using the driver with its current configuration, pass a null string as the second argument to **K_OpenDriver**. To use the driver with a different configuration, close the driver (using **K_CloseDriver**) 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**.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
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**.
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.
8001	32769	**Function not supported:** You have attempted to use a function not supported by the Function Call Driver.	Make sure that the function is supported by the board you are using. Contact the Keithley MetraByte Applications Engineering Department.
8003	32771	**Illegal board number:** An illegal board number was specified in the board initialization function.	Specify a legal board number.
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.

Table A-1. Error/Status Codes (cont.)

Error Code		Cause	Solution
Hex	Decimal		
8005	32773	**Board not found at configured address:** The board initialization function does not detect the presence of a board.	Make sure that the base address setting of the switches on the board matches the base address setting in the configuration file.
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:** Data was overwritten before it was transferred to the computer's memory.	Adjust the clock source to slow down the rate at which the board acquires data. Remove other application programs that are running and using computer resources.
8016	32790	**Interrupt overrun**: The 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 on-board 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 board supports dual-DMA, use the configuration utility to reconfigure your board to use dual-DMA.
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 DASDLL Function Call Driver can read and write counts only. When writing a value (as in **K_DAWrite**), 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.

This appendix contains instructions for converting counts to voltage and for converting voltage to counts.

Converting Counts to Voltage

You may want to convert counts to voltage when reading an analog input value.

Perform the following steps to convert a count value to voltage when reading an analog input value:

1. Unpack the count, if necessary. The way you unpack the count depends on the board you are using. Table B-1 lists the data format supported and the location of the data for each DASDLL-supported board.

Table B-1. Data Formats (Analog Input)

Board	Data Format	Location of Data
DAS-8 Series	Straight binary	Lower 12 bits
DAS-16 Series	Straight binary	Upper 12 bits
DAS-20	Bipolar: twos complement Unipolar: straight binary	Upper 12 bits
DAS-40 Series	Switch-configurable	Lower 12 bits
DAS-HRES	Straight binary	All 16 bits
Series 500: AMM1A	Straight binary	Upper 12 bits
Series 500: AMM2	Straight binary	All 16 bits

For example, if you are using a DAS-16 Series board (12-bit board), use the following equation to produce a count value that ranges from 0 through 4095.

count = (right-shift data four bits) bit-wise AND with 0FFF

2. Use the equation that is appropriate for the analog input range type, substituting the count value for *count* and the span of the analog input range for *span.* The *full scale value* depends on the number of bits supported by the board; refer to Table B-2.

Bipolar

$$\text{Voltage} = \frac{(\text{count} - \text{half full scale value}) \times \text{span}}{\text{full scale value}}$$

Unipolar

$$\text{Voltage} = \frac{\text{count} \times \text{span}}{\text{full scale value}}$$

Table B-2. Full Scale Values

Number of Bits	Full Scale Value
8	256
12	4096
16	65536

For example, assume that you are using a DAS-16 Series board (12-bit board) and want to read analog input data from a channel configured for a span of 10 V and a unipolar input range. The count value is 3072. The voltage is determined as follows:

$$\frac{3072 \times 10}{4096} = 7.5 \text{ V}$$

As another example, assume that you are using a DAS-16 Series board and want to read the analog input data from a channel configured for a span of 10 V and a bipolar input range. The count value is 1024. The voltage is determined as follows:

$$\frac{(1024 - 2048) \times 10}{4096} = -2.5 \text{ V}$$

Converting Voltage to Counts

You must convert voltage to counts when specifying an analog output value.

Perform the following steps to convert a voltage value to a count when specifying an analog output value:

1. Use the equation that is appropriate for the analog output range type, substituting the desired voltage for V_{out} and the span of the analog output range for *span*. The *full scale value* depends on the number of bits supported by the board; refer to Table B-2 on page B-3.

   Bipolar

   $$\text{Count} = \frac{V_{out} \times \text{full scale value}}{\text{span}} + \text{half full scale value}$$

   Unipolar

   $$\text{Count} = \frac{V_{out} \times \text{full scale value}}{\text{span}}$$

   For example, assume that you are using a DAS-16 Series board (12-bit board) and want to specify an analog output of 3 V for a channel configured for a span of 10 V and a bipolar output range. The count is determined as follows:

   $$\frac{3 \times 4096}{10} + 2048 = 3277$$

2. Pack the count into a variable, if necessary. The way you pack the count depends on the board you are using. Table B-1 lists the data format supported and the location of the data for each DASDLL-supported board.

Table B-3. Data Formats (Analog Output)

Board	Data Format	Location of Data
DAS-8/AO	Straight binary	Lower 12 bits
DAS-16 Series	Straight binary	Upper 12 bits
DAS-20	Twos complement	Lower 12 bits
DAS-40 Series	Straight binary	Lower 12 bits
DAS-HRES	Straight binary	All 16 bits
DDA-06	Straight binary	Lower 12 bits
Series 500: AOM1/2	Straight binary	Lower 12 bits
Series 500: AOM1/5	Straight binary	Lower 12 bits
Series 500: AOM2/1	Straight binary	Lower 12 bits
Series 500: AOM2/2	Straight binary	All 16 bits
Series 500: AOM3	Straight binary	Lower 12 bits
Series 500: AOM4	Straight binary	Lower 12 bits
Series 500: AOM5	Sign and magnitude	Lower 12 bits = magnitude MSB = sign bit

For example, if you are using a DAS-16 Series board (12-bit board), use the following equation:

variable data = (left-shift count four bits) bit-wise AND with FFF0

C

Operating Specifications

This appendix provides board-specific operating specifications on gains and channels.

Gains

DASDLL FCD functions use gain codes to represent the gain assigned to a particular channel on a DASDLL-supported board. These gain codes are listed in the following tables:

- Table C-1 on page C-2 lists analog input ranges, gains, and corresponding gain codes for DASDLL-supported boards that support analog input operations.
- Table C-2 on page C-6 lists the gains and gain codes for Series 500 boards. Note that some Series 500 boards combine the use of local and global gains to determine the total gain assigned to a channel.

Refer to your External DAS Driver board's user's guide for more information.

.

Table C-1. Gain Codes for DASDLL-Supported Boards

Series	Board	A/D Mode	Gain	Input Range	Gain Code
DAS-8[1]	DAS-8PGA DAS-8/AO	Unipolar	1	0 to 10 V	9
			10	0 to 1 V	11
			100	0 to 100 mV	13
			500	0 to 20 mV	15
		Bipolar	1	±10 V	8
			2	±5 V	0
			20	±500 mV	10
			200	±50 mV	12
			1000	±10 mV	14
	DAS-8PGA-G2	Unipolar	1	0 to 10 V	9
			2	0 to 5 V	11
			4	0 to 2.5 V	13
			8	0 to 1.25 V	15
		Bipolar	1	±10 V	8
			2	±5 V	0
			4	±2.5 V	10
			8	±1.25 V	12
			16	±0.625 V	14

Table C-1. Gain Codes for DASDLL-Supported Boards (cont.)

Series	Board	A/D Mode	Gain	Input Range	Gain Code
DAS-16[2]	DAS-16G1	Unipolar	1	0 to 10 V	0
			10	0 to 1 V	1
			100	0 to 100 mV	2
			500	0 to 20 mV	3
		Bipolar	1	±10 V	0
			10	±1 V	1
			100	±100 mV	2
			500	±20 mV	3
	DAS-16G2	Unipolar	1	0 to 10 V	0
			2	0 to 5 V	1
			4	0 to 2.5 mV	2
			8	0 to 1.25 mV	3
		Bipolar	1	±10 V	0
			2	±5 V	1
			4	±2.5 V	2
			8	±1.25 V	3
DAS-20	DAS-20	Unipolar	1	0 to 10 V	0 or 2
			10	0 to 1 V	4
			100	0 to 100 mV	6
		Bipolar	0.5	±10 V	1
			1	±5 V	3
			10	±0.5 V	5
			100	±50 mV	7

Table C-1. Gain Codes for DASDLL-Supported Boards (cont.)

Series	Board	A/D Mode	Gain	Input Range	Gain Code
DAS-40	DAS-40G1	0 to 10 V[3]	1	0 to 10 V	0
			10	0 to 1 V	1
			100	0 to 100 mV	2
			500	0 to 20 mV	3
		±10 V[3]	1	±10 V	0
			10	±1 V	1
			100	±100 mV	2
			500	±20 mV	3
		±5 V[3]	1	±5 V	0
			10	± 500 mV	1
			100	± 50 mV	2
			500	±10 mV	3
	DAS-40G2	0 to 10 V[3]	1	0 to 10 V	0
			2	0 to 5 V	1
			4	0 to 2.5 V	2
			8	0 to 1.25 V	3
		±10 V[3]	1	±10 V	0
			2	±5 V	1
			4	±2.5 V	2
			8	±1.25 V	3
		±5 V[3]	1	±5 V	0
			2	±2.5 V	1
			4	±1.25 V	2
			8	±625 mV	3

Table C-1. Gain Codes for DASDLL-Supported Boards (cont.)

Series	Board	A/D Mode	Gain	Input Range	Gain Code
DAS-HRES	DAS-HRES	Unipolar	1	0 to 10 V	0
			2	0 to 5 V	1
			4	0 to 2.5 V	2
			8	0 to 1.25 V	3
		Bipolar	1	±10 V	0
			2	±5 V	1
			4	±2.5 V	2
			8	±1.25 V	3

Notes

[1] The DAS-8 and the DAS-8LT do not have programmable gains. The analog input range for both boards is always ±5 V.

[2] Gains on the DAS-16 and DAS-16F boards are switch-selectable.

[3] Analog input range is switch-selectable.

Table C-2. Gain Codes for Series 500 Boards

Module[1]	Local Gain	Global Gain	Total Gain	Gain Code
AMM1A AMM2	1	1	1	0
	1	2	2	1
	1	5	5	2
	1	10	10	3
	10	1	10	4
	10	2	20	5
	10	5	50	6
	10	10	100	7
AIM2 AIM4 AIM9	--	--	1	0
	--	--	2	1
	--	--	5	2
	--	--	10	3
AIM3A	1	1	1	0
	1	2	2	1
	1	5	5	2
	1	10	10	3
	10	1	10	4
	10	2	20	5
	10	5	50	6
	10	10	100	7
	100	1	100	8
	100	2	200	9
	100	5	500	10
	100	10	1000	11

Table C-2. Gain Codes for Series 500 Boards (cont.)

Module[1]	Local Gain	Global Gain	Total Gain	Gain Code
AIM6	--	--	50	0
	--	--	100	1
	--	--	250	2
	--	--	500	3
AIM7	--	--	100	0
	--	--	200	1
	--	--	500	2
	--	--	1,000	3
AIM8	1	1	1	0
	1	2	2	1
	1	5	5	2
	1	10	10	3
	10	1	10	4
	10	2	20	5
	10	5	50	6
	10	10	100	7
	100	1	100	8
	100	2	200	9
	100	5	500	10
	100	10	1000	11
	1000	1	1000	12
	1000	2	2000	13
	1000	5	5000	14
	1000	10	10000	15

Notes

[1] Series 500 modules not listed in this table do not have programmable gains.

Channels

Table C-3 lists the number of available analog input and analog output channels on DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, DDA-06, PIO Series, and PDMA Series boards.

Notes: For information on the number of analog input and analog output channels supported on Series 500 boards, refer to the *Keithley Instruments 500/575 External DAS Drivers* user's guide.

DAS-8 Series, DAS-16 Series, DAS-20, DAS-40 Series, DAS-HRES, DDA-06, PIO Series, and PDMA Series boards support one digital input channel (channel 0) and one digital output channel (channel 0). Series 500 boards treat each 8-bit digital input port or 8-bit digital output port as a separate channel. For information on the number of available digital I/O channels on Series 500 boards, refer to the *Keithley Instruments 500/575 External DAS Drivers* user's guide.

Table C-3. Channels Available

Series	Board Type	A/D Channels		D/A Channels
		Onboard	Expansion	
DAS-8	DAS-8	8 single-ended	128	0
	DAS-8LT	8 single-ended	128	0
	DAS-8PGA	8 single-ended or 8 differential[1]	128	0
	DAS-8PGA-G2	8 single-ended or 8 differential[1]	128	0
	DAS-8/AO	8 differential	128	2
DAS-16	DAS-16	16 single-ended or 8 differential[1]	256	2
	DAS-16F	16 single-ended or 8 differential[1]	256	2
	DAS-16G1	16 single-ended or 8 differential[1]	256	2
	DAS-16G2	16 single-ended or 8 differential[1]	256	2

Table C-3. Channels Available (cont.)

Series	Board Type	A/D Channels		D/A Channels
		Onboard	Expansion	
DAS-20	DAS-20	16 single-ended or 8 differential[1]	256	2
DAS-40	DAS-40	16 single-ended or 8 differential[1]	Not supported	2
DAS-HRES	DAS-HRES	8 differential	Not supported	2
DDA	DDA-06	0	0	6

Notes
[1] Switch-selectable.

D

Keithley Memory Manager

The process that Windows uses to allocate memory can limit the amount of memory available to Keithley DAS boards operating in Windows Enhanced mode. To reserve a memory heap large enough for the needs of your application, use the Keithley Memory Manager (KMM) that is included in the DASDLL software package.

The reserved memory heap is part of the total physical memory available in your system. When you start up Windows, the KMM reserves the memory heap. Then, whenever your application program requests memory, the memory buffer is allocated from the reserved memory heap instead of from the Windows global heap. The KMM is DAS board independent and can be used by multiple Keithley DAS Windows application programs simultaneously.

Note: The memory allocated with the KMM can be used by any DMA controller, if applicable.

The following are supplied with the KMM:

- **VDMAD.386 -** Customized version of Microsoft's Virtual DMA Driver. This file consists of a copy of Microsoft's Virtual DMA Driver and a group of functions that are added to perform the KMM functions. When you use the KMM to reserve a memory heap, Microsoft's Virtual DMA Driver is replaced by the VDMAD.386 file.

  Note: If you have multiple versions of VDMAD.386, it is recommended that you install the latest version; to determine which version is the latest version, refer to the time stamp of the file.

- **KMMSETUP.EXE** - Windows program that helps you set up the VDMAD.386 parameters and then modifies your SYSTEM.INI file accordingly.

Installing and Setting Up the KMM

To install and set up the KMM whenever you start up Windows, you must modify the SYSTEM.INI file. You can modify the SYSTEM.INI file using either the KMMSETUP.EXE program or a text editor.

Using KMMSETUP.EXE

Using the KMMSETUP.EXE program, you can modify your Windows SYSTEM.INI file as follows:

1. Invoke KMMSETUP.EXE in one of the following ways:
   - From the Program Manager menu, choose File and then Run, and then type the complete path and program name for KMMSETUP.
   - Select the KMMSETUP icon, if installed.
2. In the New VDMAD.386 box, enter the path and name of the VDMAD.386 file, as follows:
   `C:\WINDOWS\VDMAD.386`

   The string you enter replaces `*vdmad` in the `device=*vdmad` line in your SYSTEM.INI file.

   Note: Normally, the VDMAD.386 file is stored in the WINDOWS directory. If it is stored elsewhere, enter the correct path and name or use the Browse button to find the file.

   ---

3. Notice the Current Setting box. The value specified reflects the current size of the reserved memory heap in kilobytes.
4. In the Desired Setting box, enter the desired size of the reserved memory heap in kilobytes.

   The value you enter replaces the `KEIDMAHEAPSIZE=` line in the [386Enh] section of your SYSTEM.INI file.

Note: The memory size you specify is no longer available to Windows. For example, if your computer has 8 MBytes of memory installed and you specify `KEIDMAHEAPSIZE=1000` (1 MByte), Windows can only see and use 7 MBytes.

If you specify a value less than 128, a 128K byte minimum heap size is assumed. The maximum heap size is limited only by the physical memory installed in your system and by Windows itself.

5. Select the Update button to update the SYSTEM.INI file with the changes you have made.
6. Restart Windows to ensure that the system changes take effect.

Using a Text Editor

Using a text editor, you can modify your Windows SYSTEM.INI file in the [386Enh] section, as follows:

1. Replace the line `device=*vdmad` with the following:
   ```
   device=c:\windows\vdmad.386
   ```

Note: Normally, the VDMAD.386 file is stored in the WINDOWS directory. If it is stored elsewhere, enter the correct path and name.

2. Add the following line:
   `KEIDMAHEAPSIZE=<`*size*`>`

   where *size* indicates the desired size of the reserved memory heap in kilobytes.

Note: The memory size you specify is no longer available to Windows. For example, if your computer has 8 MBytes of memory installed and you specify `KEIDMAHEAPSIZE=1000` (1 MByte), Windows can only see and use 7 MBytes.

If you do not add the `KEIDMAHEAPSIZE` keyword or if the size you specify is less than 128, a 128K byte minimum heap size is assumed. The maximum heap size is limited only by the physical memory installed in your system and by Windows itself.

3. Restart Windows to ensure that the system changes take effect.

Removing the KMM

If you make changes to the SYSTEM.INI file, you can always remove the updated information from the SYSTEM.INI file and return all previously reserved memory to Windows.

If you are using KMMSETUP.EXE, select the Remove button to remove the updated information. If you are using a text editor, modify and/or delete the appropriate lines in SYSTEM.INI. In both cases, make sure that you restart Windows to ensure that the system changes take effect.

Index

A

B

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V