DAS-1600/1400/1200 Series Function Call Driver

USER'S GUIDE

DAS-1600/1400/1200 Series
Function Call Driver
User's Guide

Revision C – May 1996
Part Number: 80950

New Contact Information

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

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

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

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

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

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

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 programs for DAS-1600/1400/1200 Series boards using the DAS-1600/1400/1200 Series Function Call Driver. The DAS-1600/1400/1200 Series Function Call Driver supports the following DOS-based languages:

- Microsoft® QuickBasic™ (Version 4.5)
- Microsoft Professional Basic (Version 7.0 and higher)
- Microsoft Visual Basic® for DOS (Version 1.0)
- Microsoft C/C++ (Version 4.0 and higher)
- Borland® C/C++ (Version 1.0 and higher)
- Borland Turbo Pascal® for DOS (Version 7.0 and higher)

The DAS-1600/1400/1200 Series Function Call Driver also supports the following Windows™-based languages:

- Microsoft C/C++ (Version 7.0 and higher)
- Borland C/C++ (Version 4.0 and higher)
- Microsoft Visual Basic for Windows (Version 3.0 and higher)
- Microsoft Visual C++™ (Version 1.0 and higher)
- Borland Turbo Pascal for Windows (Version 1.0 and higher)

The manual is intended for programmers using a DAS-1600/1400/1200 Series board in an IBM® PC AT® or compatible computer. It is assumed that you have read the user's guide for your board to familiarize yourself with the board's features, and that you have completed the appropriate hardware installation and configuration.

It is also assumed that you are experienced in programming in your selected language and that you are familiar with data acquisition principles.

The *DAS-1600/1400/1200 Series Function Call Driver User's Guide* is organized as follows:

- Chapter 1 provides an overview of the Function Call Driver, a summary of the Function Call Driver functions, a series of flow diagrams illustrating the procedures used when programming each of the operations supported by the Function Call Driver, and information on how to get help.
- Chapter 2 contains the background information needed to use the functions included in the Function Call Driver.
- Chapter 3 contains a programming overview and language-specific information related to using the Function Call Driver.
- Chapter 4 contains detailed descriptions of the functions, arranged in alphabetical order.
- Appendix A contains a list of the error codes returned by the Function Call Driver.
- Appendix B contains instructions for converting counts to voltage and for converting voltage to counts.
- An index completes this manual.

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

- References to DAS-1600/1400/1200 Series boards apply to all members of the family. When a feature applies to a particular board, that board's name is used.
- References to BASIC apply to all DOS-based BASIC languages (Microsoft QuickBasic, Microsoft Professional Basic, and Microsoft Visual Basic for DOS). When a feature applies to a specific language, the complete language name is used. References to Visual Basic for Windows apply to Microsoft Visual Basic for Windows.
- Keyboard keys are represented in bold.

Table of Contents

List of Figures

List of Tables

1

Getting Started

This chapter contains the following sections:

- **Overview** - a description of the DAS-1600/1400/1200 Function Call Driver.
- **Summary of Functions** - a brief description of the DAS-1600/1400/1200 Function Call Driver functions.
- **Programming Flow Diagrams** - an illustration of the procedures to follow when programming a DAS-1600/1400/1200 Series board using the DAS-1600/1400/1200 Function Call Driver.
- **Getting Help** - information on how to get help when installing or using the DAS-1600/1400/1200 Function Call Driver.

Overview

The DAS-1600/1400/1200 Series Function Call Driver is a library of data acquisition and control functions (referred to as the Function Call Driver or FCD functions). It is part of the following two software packages:

- **DAS-1600/1400/1200 Series standard software package** - This is the software package that is shipped with DAS-1600/1400/1200 Series boards; it includes the following:
  - Libraries of FCD functions for Microsoft QuickBasic, Microsoft Professional Basic, and Microsoft Visual Basic for DOS.
  - Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.

  - Utility programs, running under DOS, that allow you to configure, calibrate, and test the features of DAS-1600/1400/1200 Series boards.
  - Language-specific example programs.

- **ASO-1600/1400/1200 software package** - This is the Advanced Software Option for DAS-1600/1400/1200 Series boards. You purchase the ASO-1600/1400/1200 software package separately from the board; it includes the following:
  - Libraries of FCD functions for Microsoft C/C++, Borland C/C++, and Borland Turbo Pascal.
  - Dynamic Link Libraries (DLLs) of FCD functions for Microsoft C/C++, Borland C/C++, Microsoft Visual C++, Microsoft Visual Basic for Windows, and Borland Turbo Pascal for Windows.
  - Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.
  - Utility programs, running under DOS and Windows, that allow you to configure, calibrate, and test the functions of DAS-1600/1400/1200 Series boards.
  - Language-specific example programs.

Before you use the Function Call Driver, make sure that you have installed the software, set up the board, and created a configuration file using the setup and installation procedures described in Chapter 3 of the user's guide for your board.

Summary of Functions

Table 1-1 provides a brief description of the functions in the DAS-1600/1400/1200 Series Function Call Driver. For more detailed information about the functions, refer to Chapter 4.

Table 1-1. Summary of Functions

Type of Function	Name of Function	Description
Initialization	DAS1600_DevOpen	Initializes the DAS-1600/1400/1200 Series Function Call Driver.
	K_OpenDriver	Initializes any Function Call Driver.
	K_CloseDriver	Closes a Function Call Driver.
	DAS1600_GetDevHandle	Initializes a DAS-1600/1400/1200 Series board.
	K_GetDevHandle	Initializes any Keithley DAS board.
	K_FreeDevHandle	Frees a device handle.
	K_DASDevInit	Reinitializes a board.
Operation	K_ADRead	Reads a single analog input value.
	K_DAWrite	Writes a single analog output value.
	K_DIRead	Reads a single digital input value.
	K_DOWrite	Writes a single digital output value.
	K_DMAStart	Starts a DMA-mode operation.
	K_DMAStatus	Gets the status of a DMA-mode operation.
	K_DMAStop	Stops a DMA-mode operation.
	K_IntStart	Starts an interrupt-mode operation.
	K_IntStatus	Gets the status of an interrupt-mode operation.
	K_IntStop	Stops an interrupt-mode operation.
	K_SyncStart	Starts a synchronous-mode operation.
Frame management	K_GetADFrame	Accesses a frame for an analog input operation.
	K_GetDAFrame	Accesses a frame for an analog output operation.
	K_GetDIFrame	Accesses a frame for a digital input operation.
	K_GetDOFrame	Accesses a frame for a digital output operation.
	K_FreeFrame	Frees a frame.
	K_ClearFrame	Sets all frame elements to their default values.

Table 1-1. Summary of Functions (cont.)

Type of Function	Name of Function	Description
Memory management	K_DMAAlloc	Dynamically allocates a memory buffer for a DMA-mode operation.
	K_DMAFree	Frees a memory buffer that was dynamically allocated for a DMA-mode operation.
	K_IntAlloc	Dynamically allocates a memory buffer for an interrupt-mode or synchronous-mode operation.
	K_IntFree	Frees a memory buffer that was dynamically allocated for an interrupt-mode or synchronous-mode operation.
	KMakeDMABuf	Converts a local array to a buffer suitable for a DMA-mode operation.
	K_MoveArrayToBuf	Transfers data from a local integer array to a dynamically allocated memory buffer.
	K_MoveArrayToBufL	Transfers data from a local long array to a dynamically allocated memory buffer.
	K_MoveBufToArray	Transfers data from a dynamically allocated memory buffer to a local integer array.
	K_MoveBufToArrayL	Transfers data from a dynamically allocated memory buffer to a local long array.
	K_MoveDataBuf	Moves data from one memory area to another.
Buffer address	K_SetBuf	Specifies the address of a local array (C/C++ or Pascal) or a dynamically allocated memory buffer (C/C++, Pascal, Visual Basic for Windows, or BASIC) for an interrupt-mode or synchronous-mode operation.
	K_SetBufI	Specifies the address of a local integer array (BASIC or Visual Basic for Windows) for an interrupt-mode or synchronous-mode operation.
	K_SetBufL	Specifies the address of a local long array (BASIC or Visual Basic for Windows) for an interrupt-mode or synchronous-mode operation.
	K_SetDMABuf	Specifies the address of a dynamically allocated memory buffer for a DMA-mode operation.

Table 1-1. Summary of Functions (cont.)

Type of Function	Name of Function	Description
Buffering mode	K_SetContRun	Specifies continuous mode.
	K_ClrContRun	Specifies single-cycle mode.
Conversion mode	K_SetADFreeRun	Specifies burst mode.
	K_ClrADFreeRun	Specifies paced mode.
	K_SetSSH	Specifies burst mode with SSH (simultaneous sample-and-hold).
Channel and gain	K_SetChn	Specifies a single channel.
	K_SetStartStopChn	Specifies the first and last channels in a group of consecutive channels.
	K_SetG	Specifies the gain for a group of consecutive channels.
	K_SetStartStopG	Specifies the first and last channels in a group of consecutive channels and the gain for all channels in the group.
	K_SetChnGAry	Specifies the starting address of a channel-gain queue.
	K_FormatChnGAry	Converts the format of a channel-gain queue.
	K_RestoreChnGAry	Restores a converted channel-gain queue.
	K_GetADConfig	Gets the input channel configuration (differential or single-ended).
	K_GetADMode	Gets the input range type (bipolar or unipolar).
Clock	K_SetClk	Specifies the pacer clock source.
	K_SetClkRate	Specifies the clock rate for the internal pacer clock.
	K_GetClkRate	Gets the clock rate for the internal pacer clock.
	K_SetBurstTicks	Specifies the count value used to adjust the settling time.

Table 1-1. Summary of Functions (cont.)

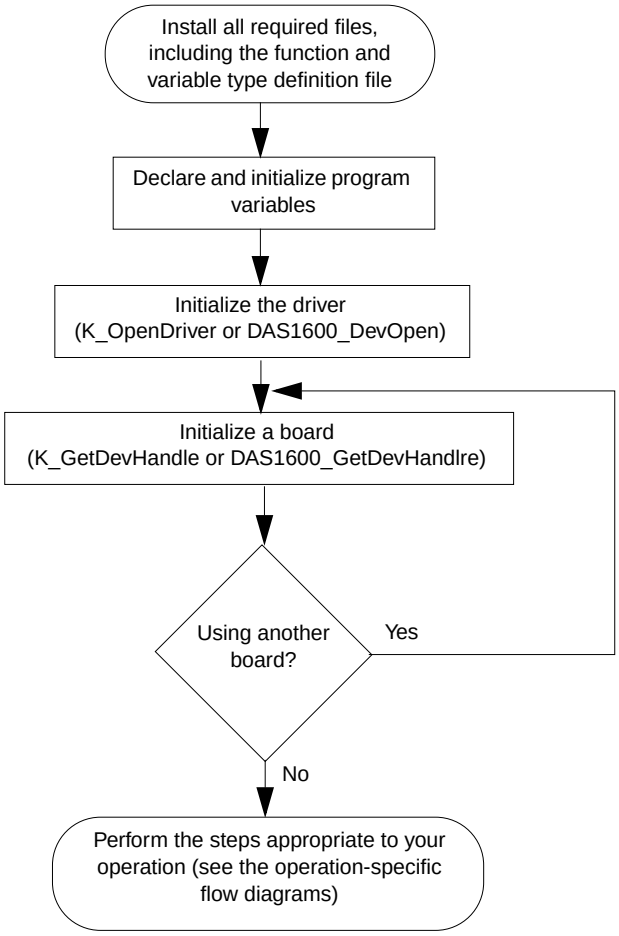
Type of Function	Name of Function	Description
Trigger	K_SetTrig	Specifies the trigger source.
	K_SetADTrig	Sets up an external analog trigger.
	K_SetTrigHyst	Specifies the hysteresis value.
	K_SetDITrig	Sets up an external digital trigger.
82C54 counter/timer[1]	DAS1600_8254Control	Writes data to the 82C54 counter/timer control register.
	DAS1600_8254SetCounter	Specifies the value of a counter.
	DAS1600_8254SetClk0	Specifies the clock source for counter 0.
	DAS1600_8254SetTrig0	Enables/disables the gate signal.
	DAS1600_8254GetCounter	Gets the value of a counter.
	DAS1600_8254GetClk0	Gets the clock source for counter 0.
	DAS1600_8254GetTrig0	Gets the status of the gate signal.
Miscellaneous	K_GetErrMsg	Gets the address of an error message string.
	K_GetVer	Gets revision numbers.
	K_GetShellVer	Gets the current DAS shell version.

Notes
[1] These functions allow you to program the 82C54 counter/timer on the DAS-1600/1400/1200 Series board. See Appendix E of your user's guide for more information.

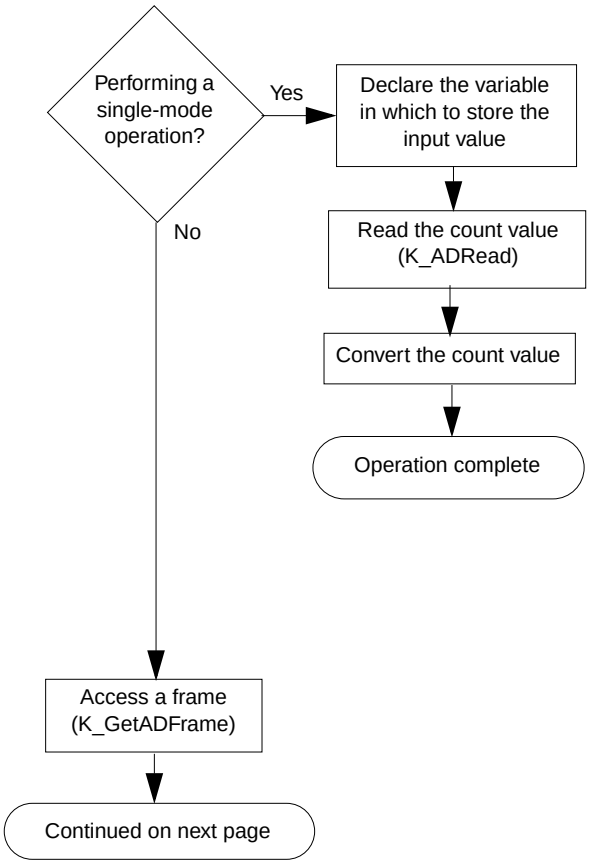
Programming Flow Diagrams

This section contains a series of programming flow diagrams illustrating the procedures used when programming each of the operations supported by the DAS-1600/1400/1200 Series Function Call Driver. Although error checking is not shown in the flow diagrams, it is recommended that you check the error/status code returned by each function used in your program.

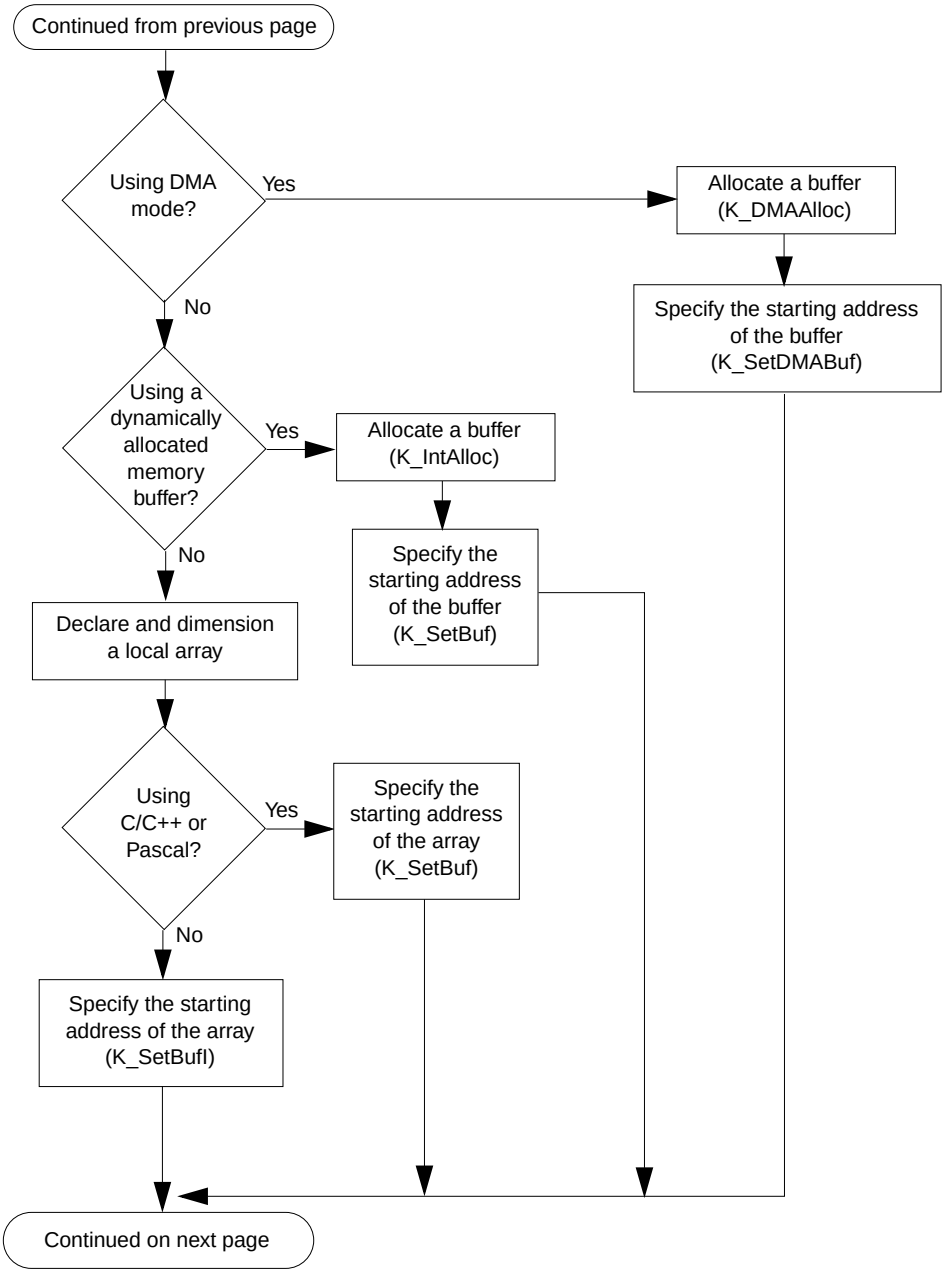
Preliminary Steps for All Operations



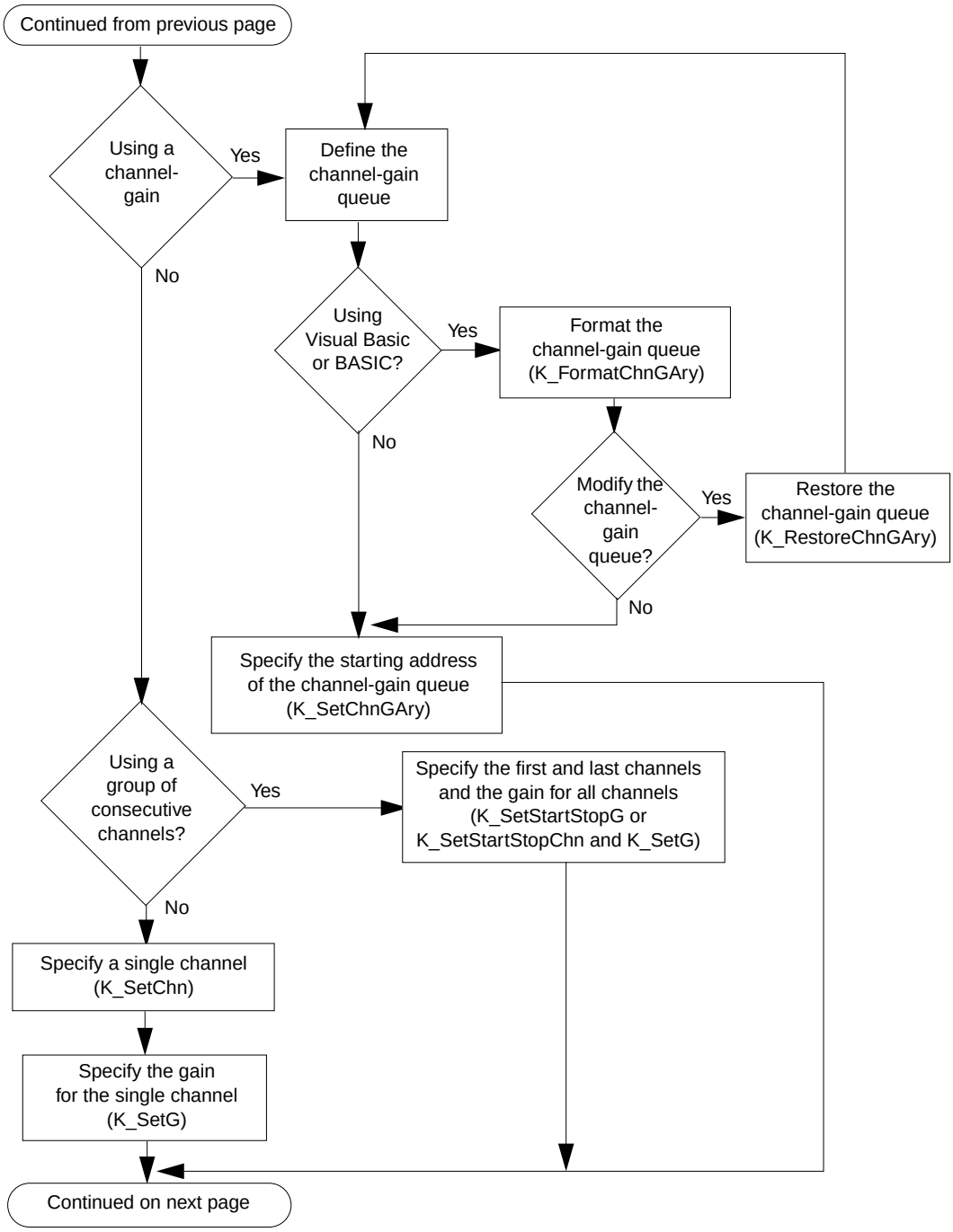
Steps for an Analog Input Operation



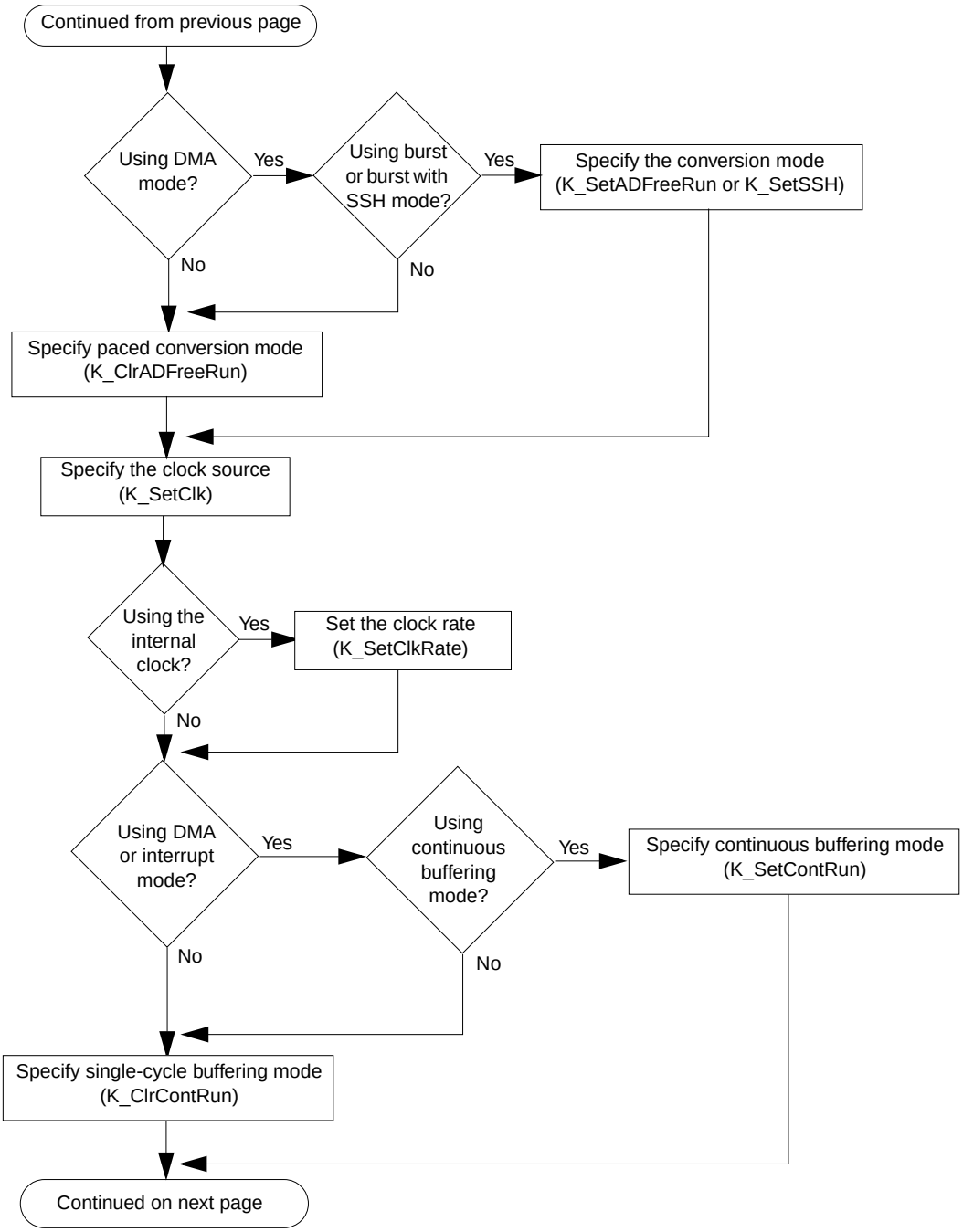
Steps for an Analog Input Operation (cont.)



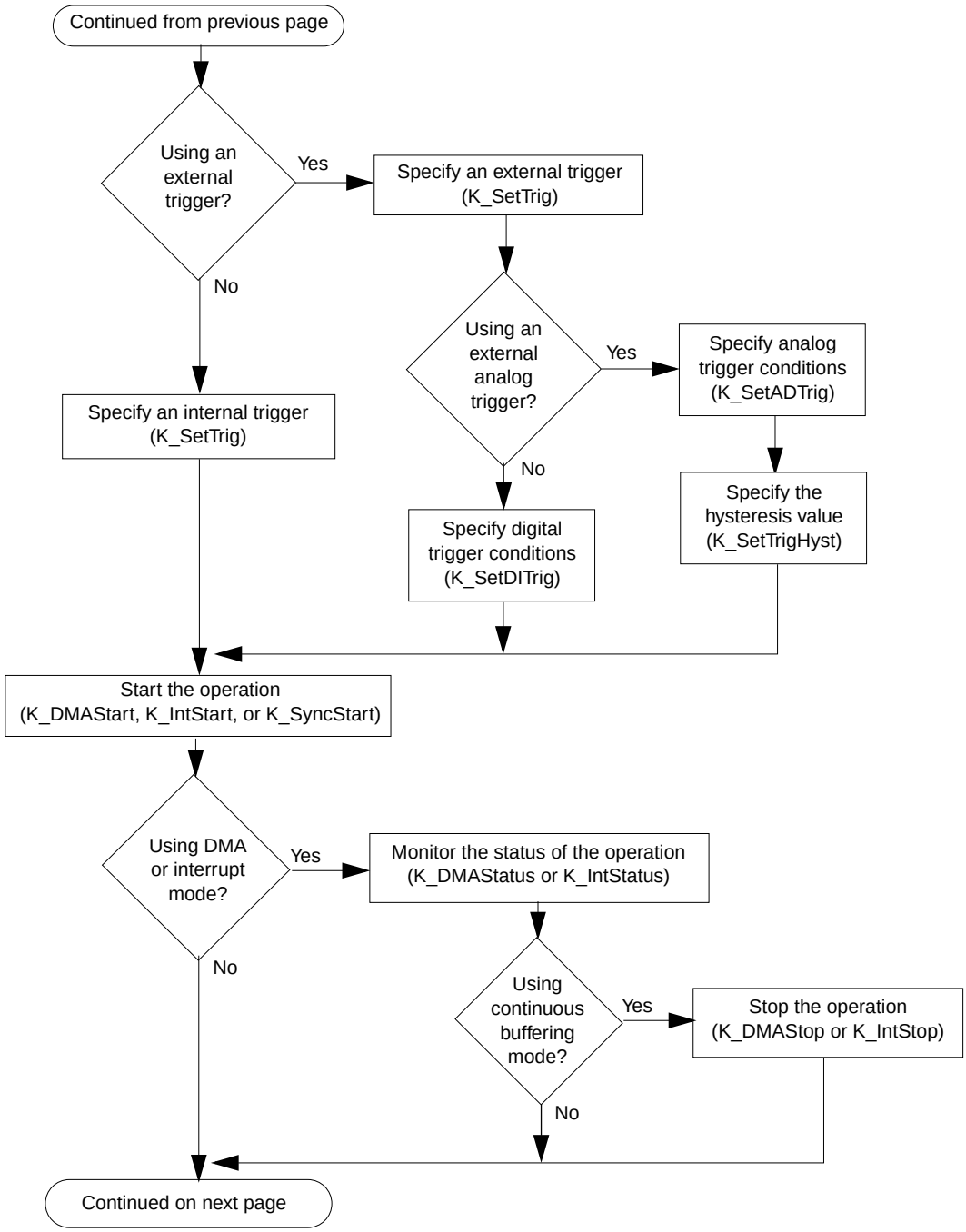
Steps for an Analog Input Operation (cont.)



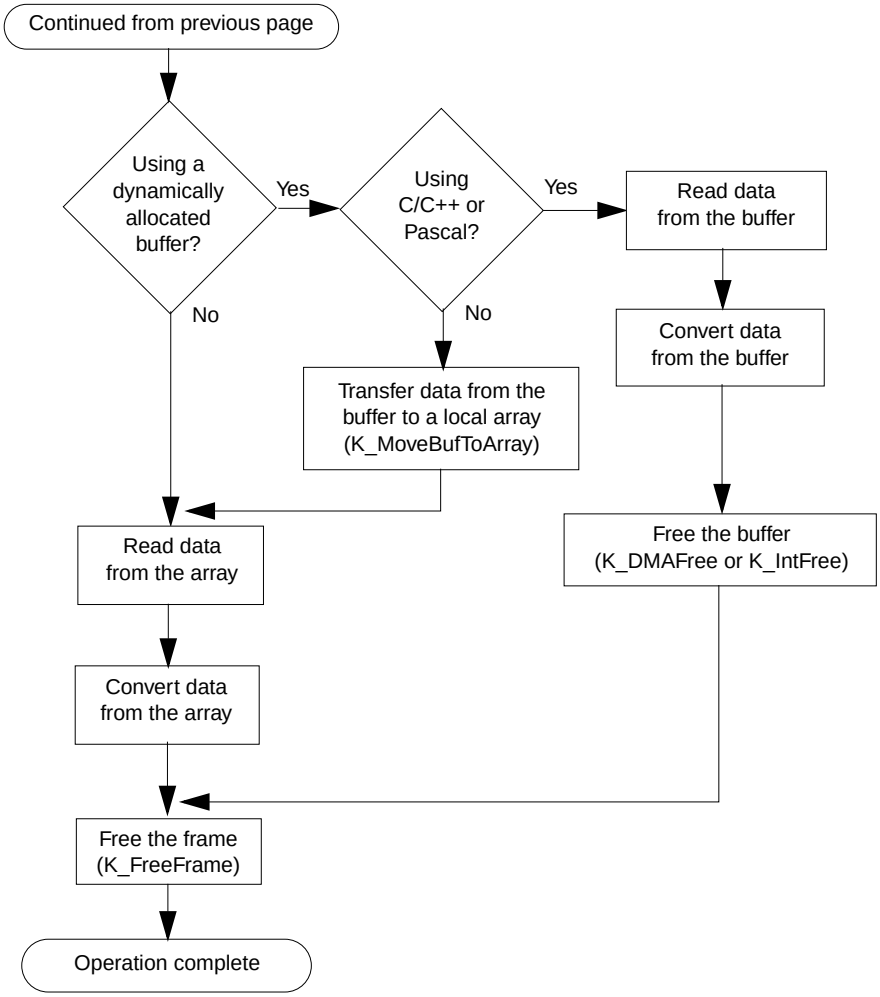
Steps for an Analog Input Operation (cont.)



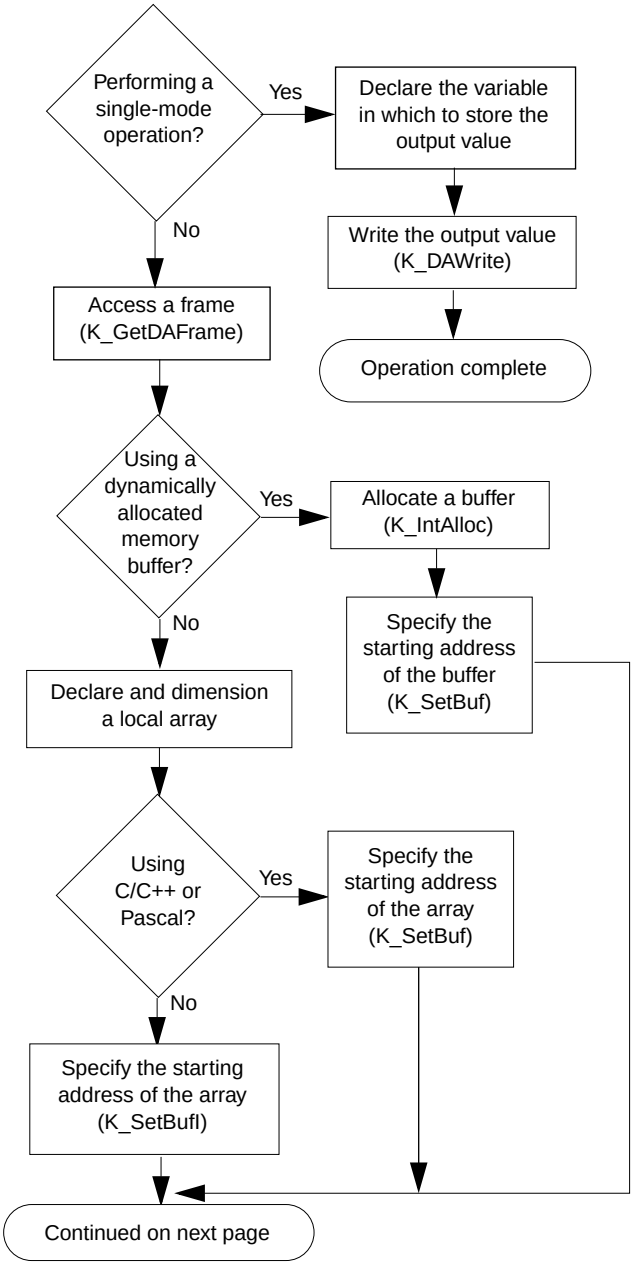
Steps for an Analog Input Operation (cont.)



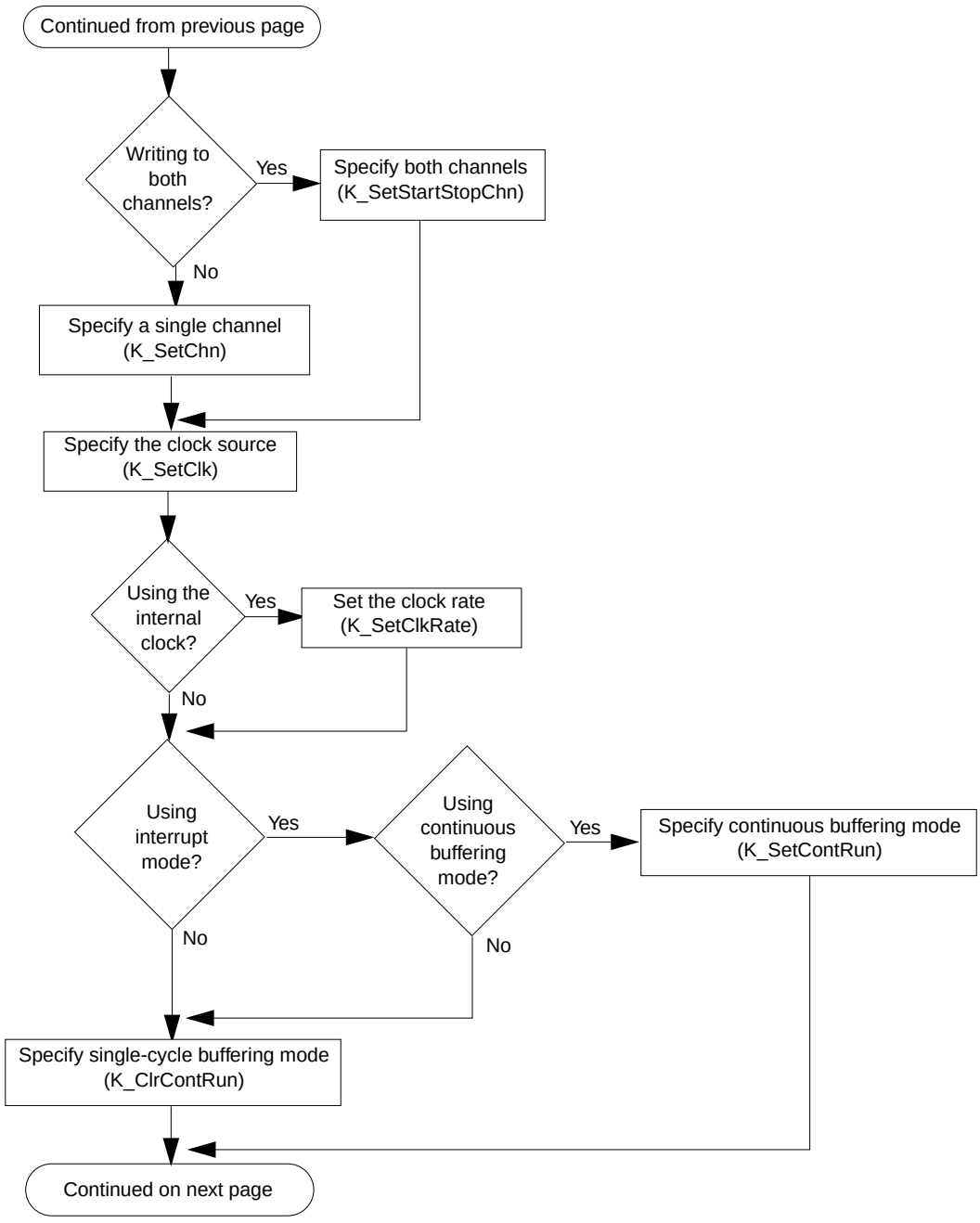
Steps for an Analog Input Operation (cont.)



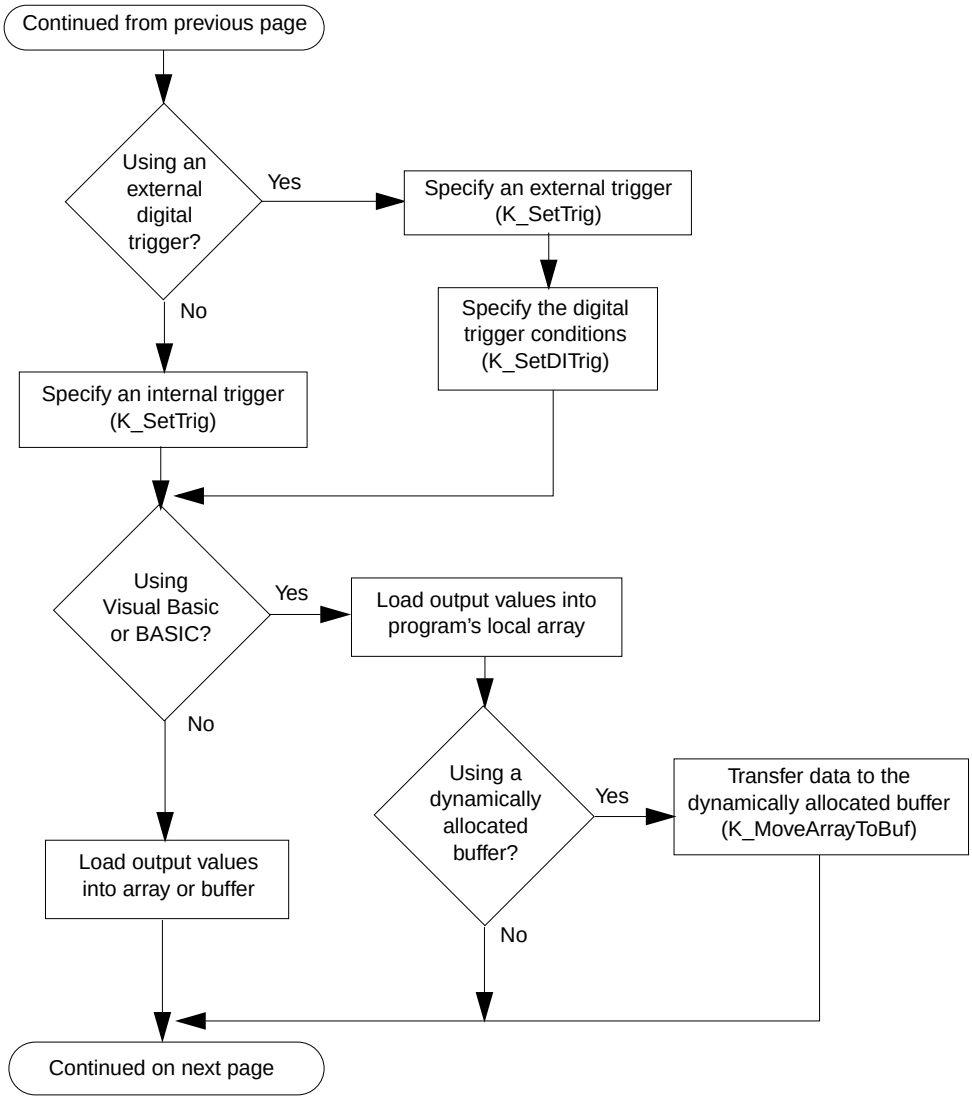
Steps for an Analog Output Operation



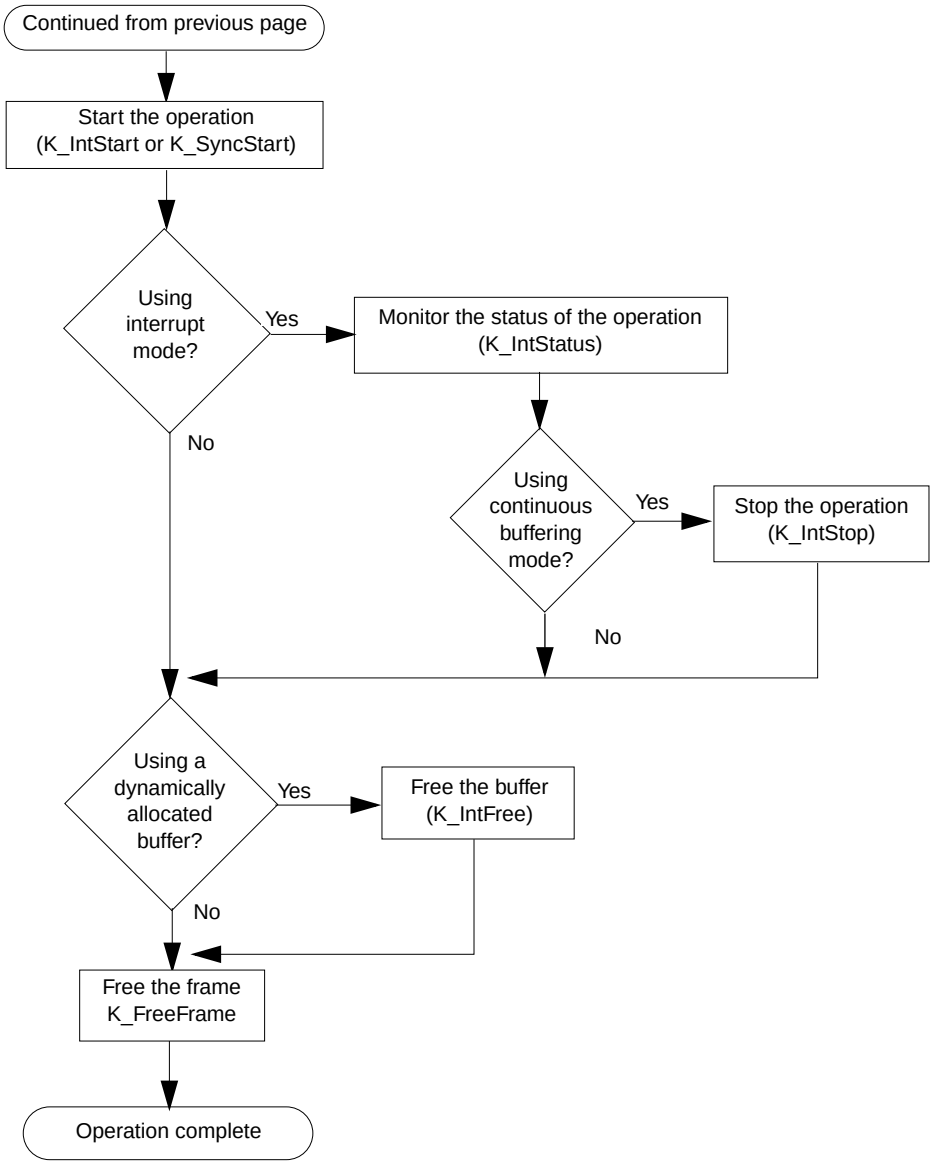
Steps for an Analog Output Operation (cont.)



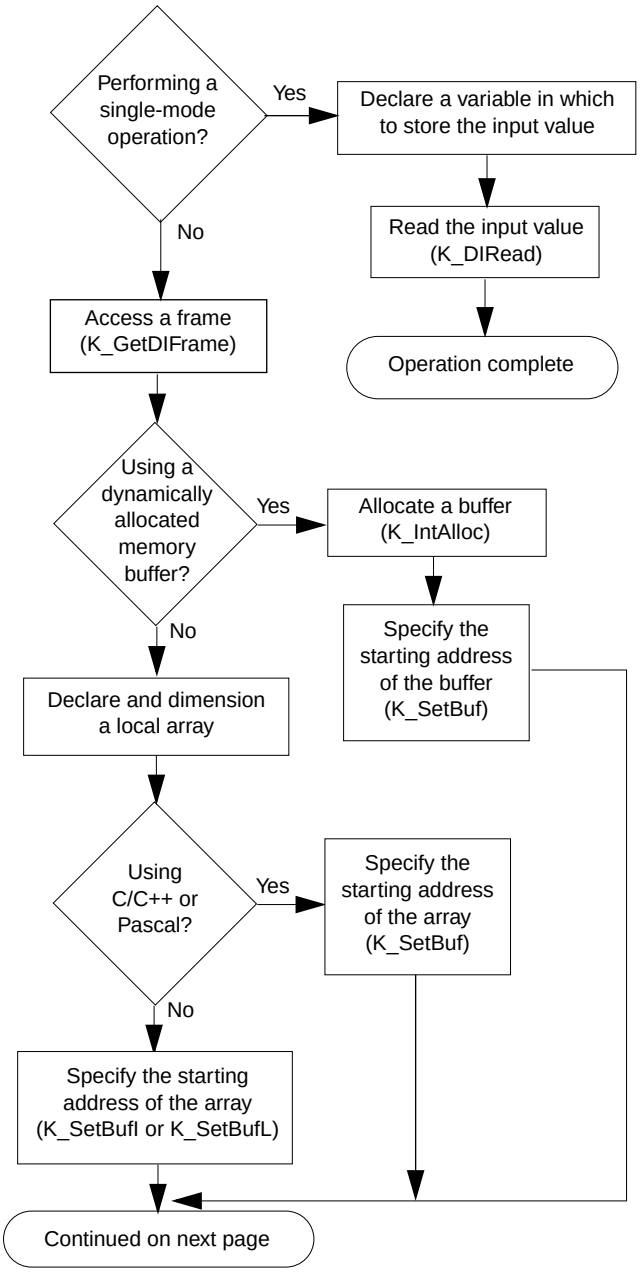
Steps for an Analog Output Operation (cont.)



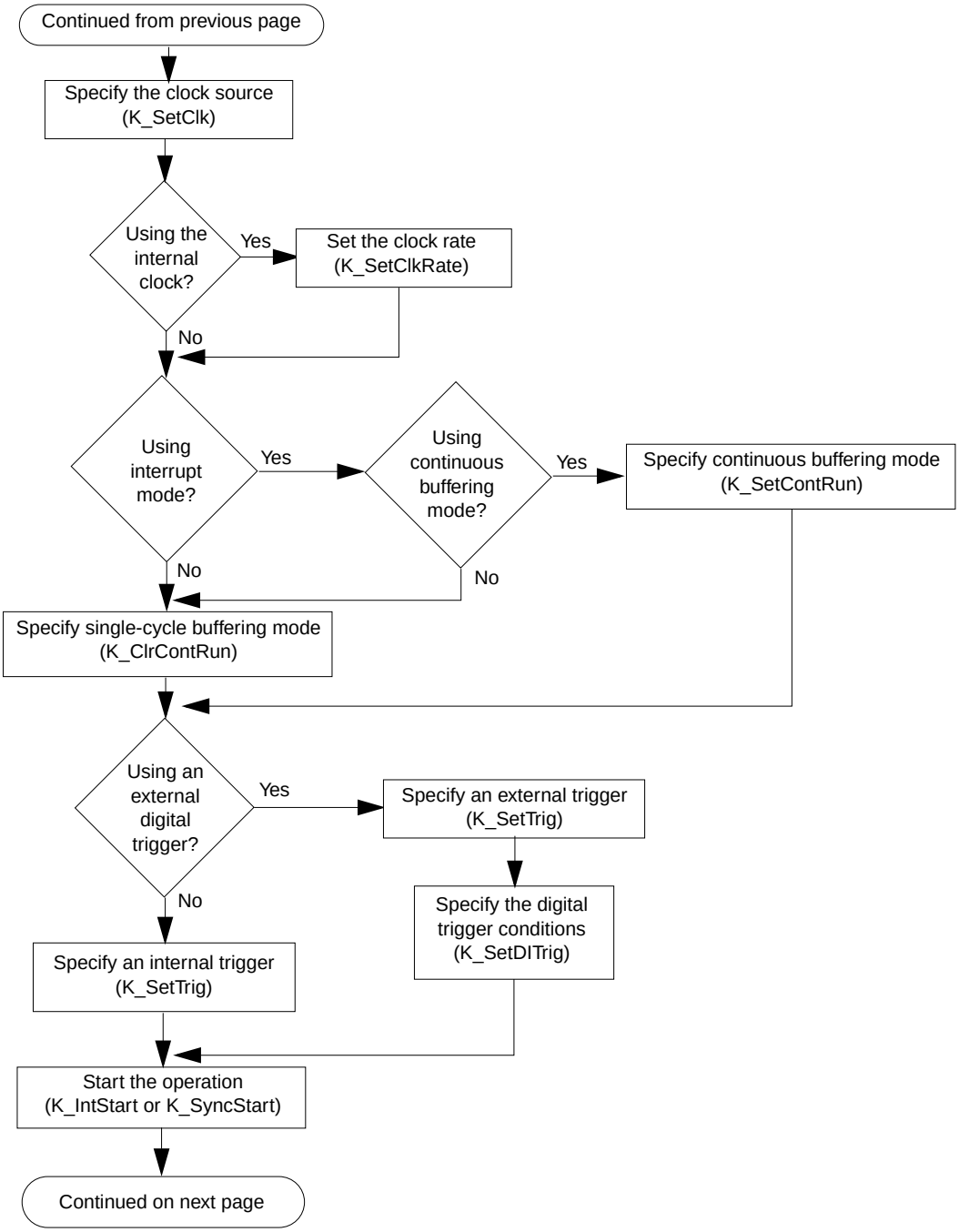
Steps for an Analog Output Operation (cont.)



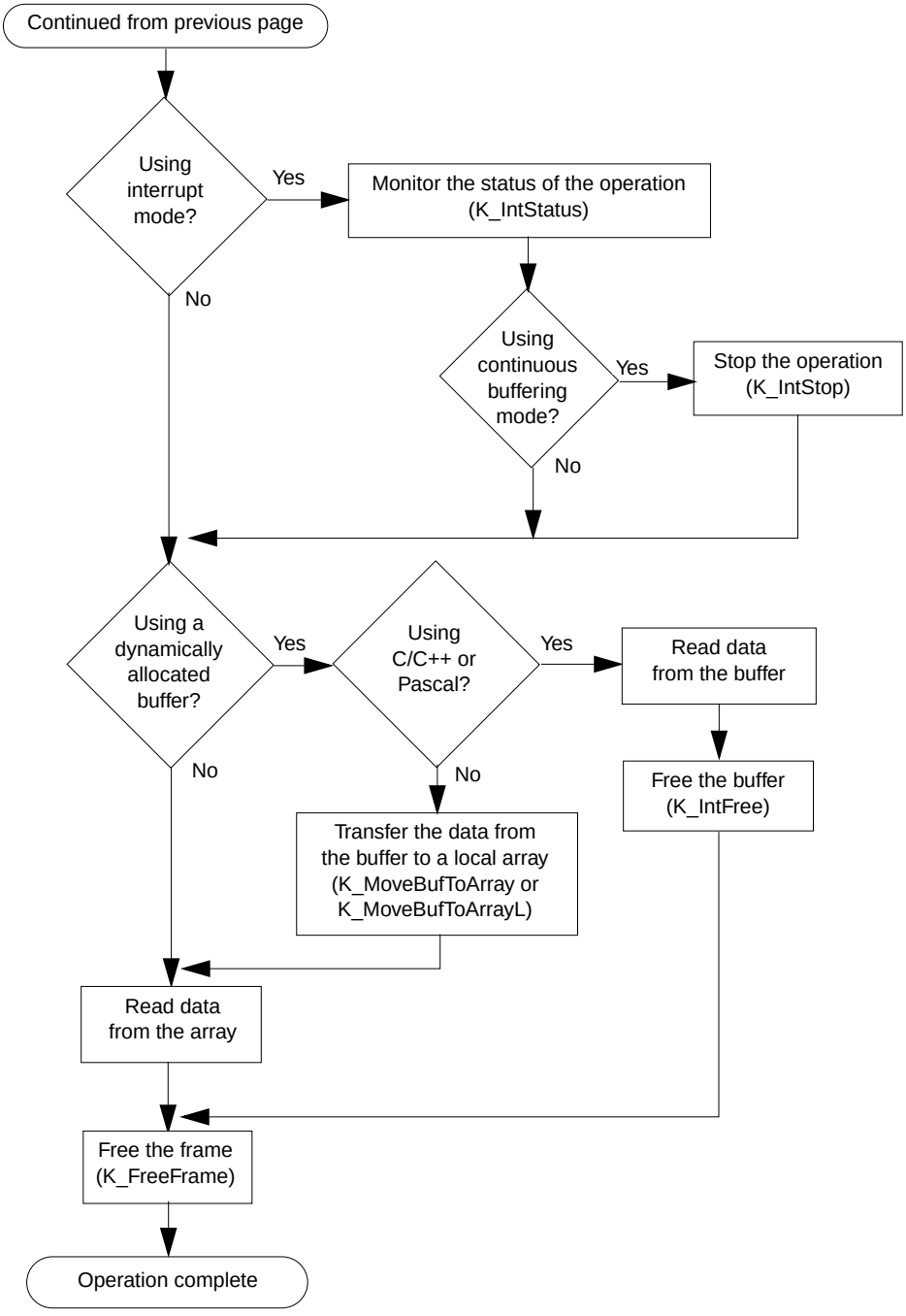
Steps for a Digital Input Operation



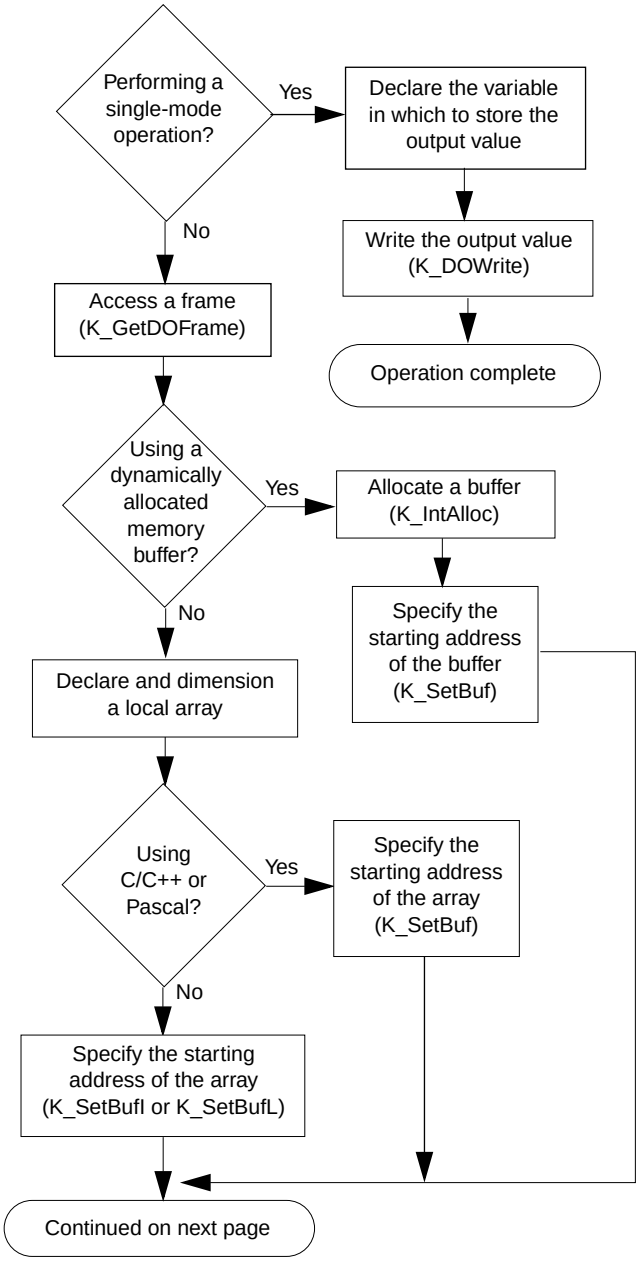
Steps for a Digital Input Operation (cont.)



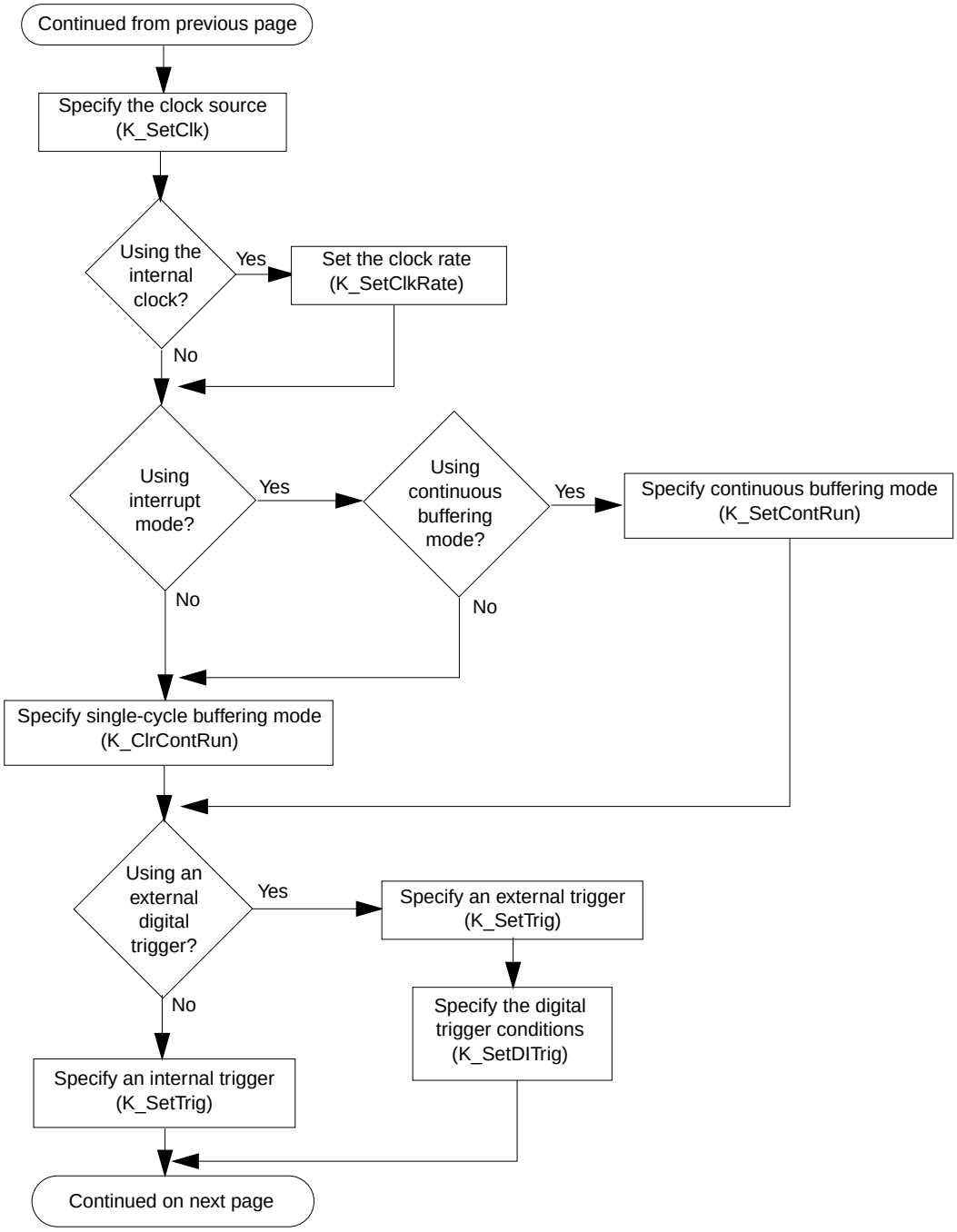
Steps for a Digital Input Operation (cont.)



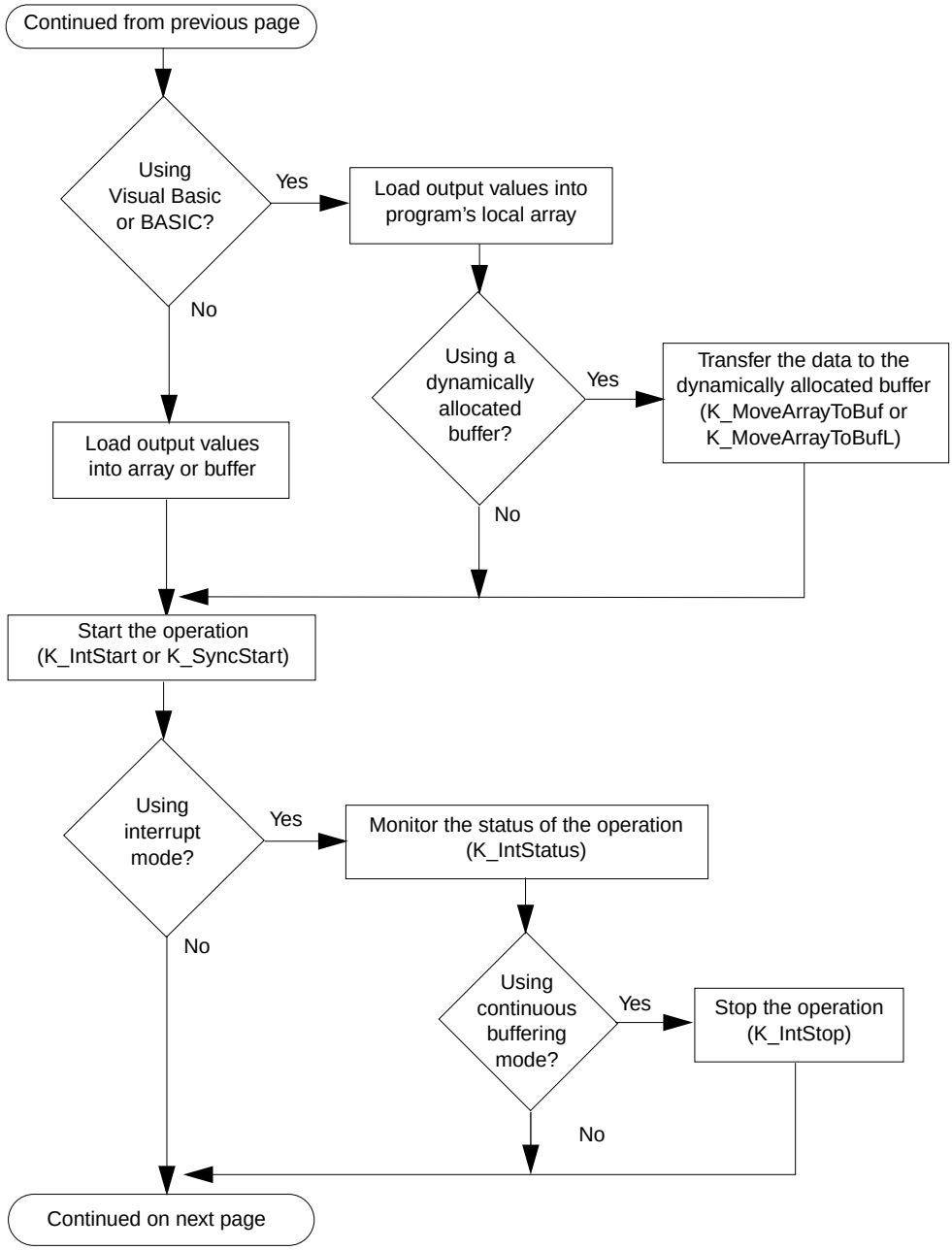
Steps for a Digital Output Operation



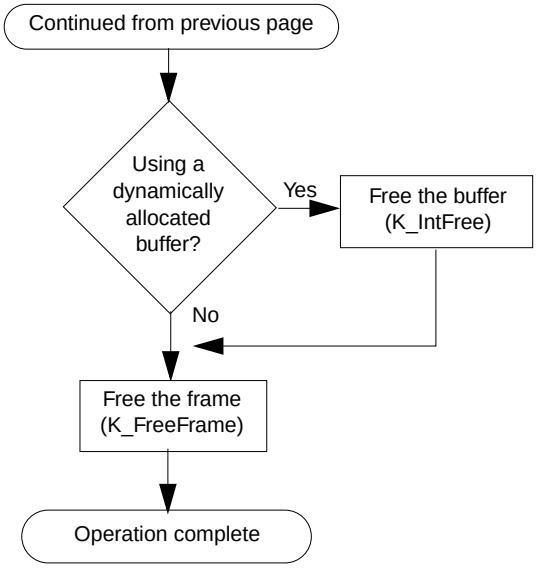
Steps for a Digital Output Operation (cont.)



Steps for a Digital Output Operation (cont.)



Steps for a Digital Output Operation (cont.)



Getting Help

If you need help installing or using the DAS-1600/1400/1200 Series Function Call Driver, call your local sales office or call Keithley MetraByte at the following number for technical support:

(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:

DAS-1600/1400/1200 Series board configuration	Model	_____
	Serial #	_____
	Revision code	_____
	Base address setting	_____
	Interrupt level setting	2, 3, 4, 5, 6, 7, None
	Input configuration	single-ended, differential
	Input range type	unipolar, bipolar
	DMA channel	1, 3
Computer	Manufacturer	_____
	CPU type	_____
	Clock speed (MHz)	_____
	Amount of RAM	_____
	Video system	_____
	BIOS type	_____
Operating system	DOS version	_____
	Windows version	_____
Software package	Name	_____
	Serial #	_____
	Version	_____
	Invoice/Order #	_____
Compiler (if applicable)	Language	_____
	Manufacturer	_____
	Version	_____
Accessories	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____

2

Available Operations

This chapter contains conceptual information about the DAS-1600/1400/1200 Function Call Driver functions. It includes the following sections:

- **System Operations** - information on initializing the Function Call Driver, initializing a board, retrieving revision levels, and handling errors.
- **Analog Input Operations** - information on operation modes, frames, memory allocation and management, gains and ranges, channels, conversion modes, pacer clocks, buffering modes, and triggers.
- **Analog Output Operations (DAS-1600 Series Only)** - information on operation modes, frames, memory allocation and management, channels, pacer clocks, buffering modes, and triggers.
- **Digital I/O Operations** - information on operation modes, frames, memory allocation and management, the digital I/O channel, pacer clocks, buffering modes, and triggers.
- **Counter/Timer I/O Operations** - information on using the 82C54 counter/timer circuitry of DAS-1600/1400/1200 Series boards.

System Operations

This section describes the miscellaneous and general maintenance operations that apply to DAS-1600/1400/1200 Series boards and to the DAS-1600/1400/1200 Series Function Call Driver. It includes information on initializing the driver, initializing a board, retrieving revision levels, and handling errors.

Initializing the Driver

You must initialize the DAS-1600/1400/1200 Series Function Call Driver and any other Keithley DAS Function Call Drivers you are using in your program. To initialize the drivers, use the **K_OpenDriver** function. Specify the driver you are using and the configuration file that defines the use of the driver. The driver returns a unique identifier for that use of the driver; this identifier is called the driver handle.

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

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

Note: If you are programming in Turbo Pascal (for DOS) or BASIC, **K_OpenDriver** and **K_CloseDriver** are not available. You must use the board-specific **DAS1600_DevOpen** function instead. **DAS1600_DevOpen** initializes the DAS-1600/1400/1200 Series Function Call Driver according to the configuration file you specify. Refer to page 4-27 for more information. In Turbo Pascal (for DOS) and BASIC, closing the DAS-1600/1400/1200 Series Function Call Driver is not required.

Initializing a Board

The DAS-1600/1400/1200 Series Function Call Driver supports up to two boards. You must use the **K_GetDevHandle** function to initialize each board you want to use. Specify the driver handle and the board number (0 or 1). **K_GetDevHandle** verifies that the board is present and sets the board to its power-up state. (Note that **K_GetDevHandle** does not set the analog output and digital output channels to a known state.)

K_GetDevHandle returns a unique identifier for each board; this identifier is called the device handle. Device handles allow you to communicate with more than one board. Use the device handle returned by **K_GetDevHandle** in subsequent function calls related to the board.

You can specify a maximum of 30 device handles for all the Keithley DAS products accessed from all your programs. If you are no longer using a Keithley DAS product and you want to free some memory or if you have used all 30 device handles, use the **K_FreeDevHandle** function to free a device handle.

Note: If you are programming in Turbo Pascal (for DOS) or BASIC, **K_GetDevHandle** and **K_FreeDevHandle** are not available. You must use the board-specific **DAS1600_GetDevHandle** function instead. Refer to page 4-30 for more information. In Turbo Pascal (for DOS) and BASIC, freeing a device handle is not required.

Use **K_GetDevHandle** the first time you initialize a board only. To reinitialize a board, use the **K_DASDevInit** function, specifying the device handle returned by **K_GetDevHandle**. **K_DASDevInit** stops all operations currently in progress and sets the board back to its power-up state. (Note that **K_DASDevInit** does not reset the current analog output and digital output values.)

Retrieving Revision Levels

If you are using functions from different Keithley DAS Function Call Drivers in the same program or if you are having problems with your 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 are shared by all DAS boards).

Handling Errors

Each FCD function returns a code indicating the status of the function. To ensure that your 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 program takes the appropriate action. Refer to Appendix A for a complete list of error codes.

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

C/C++	page 3-6
Pascal	page 3-14
Visual Basic for Windows	page 3-22
BASIC	page 3-30

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

Analog Input Operations

This section describes analog input operations. It includes information on the operation modes available, how to access a frame, how to allocate and manage memory, and how to specify channels and gains, the conversion mode, the pacer clock source, the buffering mode, and the trigger source for an analog input operation.

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 an analog input operation in single mode, synchronous mode, interrupt mode, or DMA mode, as described in the following sections.

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 perform 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.

The data in the variable is stored as a count value. Refer to Appendix B for information on converting the count value to voltage.

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. The hardware transfers the data from the board to a user-defined buffer in computer memory. After the driver transfers the specified number of samples, the driver returns control to the program. You cannot perform any other operation until the synchronous-mode operation is complete.

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

The data in the user-defined buffer is stored as count values. Refer to Appendix B for information on converting the count values to voltage.

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 program. The hardware transfers the data from the board to a user-defined buffer in computer memory using an interrupt service routine.

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

You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-24 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.

The data in the user-defined buffer is stored as count values. Refer to Appendix B for information on converting the count values to voltage.

DMA Mode

DMA mode provides the fastest data transfer rates. 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 program. The hardware transfers the data from the board to a user-defined DMA buffer in computer memory.

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 2-24 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 data in the user-defined buffer is stored as count values. Refer to Appendix B for information on converting the count values to voltage.

Frames

Synchronous-mode, interrupt-mode, and DMA-mode analog input operations require frames. A frame is a data structure whose elements define the attributes of the operation. Use the **K_GetADFrame** function to access an analog input frame, called an A/D (analog-to-digital) frame. The driver returns a unique identifier for the frame; this identifier is called the frame handle.

Specify the attributes of the operation by using a separate setup function to define each element of the A/D frame. Use the frame handle returned by the driver in each setup function to ensure that you always define the same operation. For example, assume that you access an A/D frame with the frame handle ADFrame. To specify the channel on which to perform the operation, use the **K_SetChn** setup function, referencing the frame handle ADFrame; to specify the gain at which to read the channel, use the **K_SetG** setup function, also referencing the frame handle ADFrame.

When you are ready to perform the operation you set up, use the **K_SyncStart**, **K_IntStart**, or **K_DMAStart** function to start the operation, again referencing the appropriate frame handle. Figure 2-1 illustrates the use of an A/D frame for a DMA-mode operation where the frame handle is ADFrame.

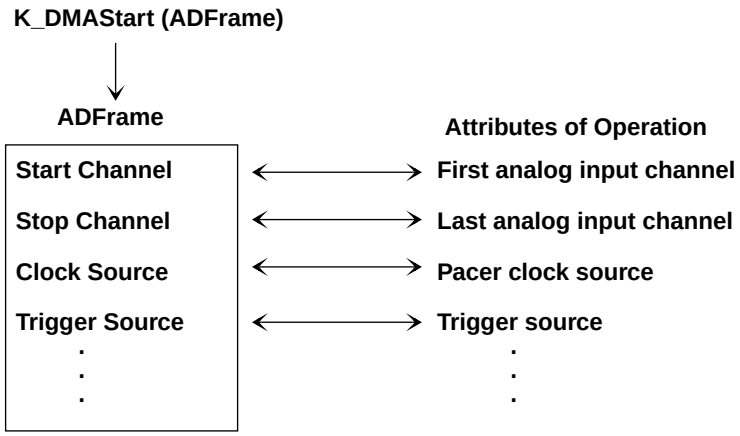


Figure 2-1. Frame-Based Operation

Frames help you create structured programs. They 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 operations that use frames.

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

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

Table 2-1 lists the elements of an A/D frame, the default value of each element, the setup functions used to define each element, and the page(s) in this manual on which to find additional information.

Table 2-1. A/D Frame Elements

Element	Default Value	Setup Function	Page Number
Buffer[1]	0 (NULL)	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetDMABuf	page 4-156
Number of Samples	0	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetDMABuf	page 4-156
Buffering Mode	Single-cycle	K_SetContRun	page 4-151
		K_ClrContRun[2]	page 4-41
Start Channel	0	K_SetChn	page 4-141
		K_SetStartStopChn	page 4-164
		K_SetStartStopG[3]	page 4-166
Stop Channel	0	K_SetStartStopChn	page 4-164
		K_SetStartStopG[3]	page 4-166
Gain	0 (gain of 1)	K_SetG[3]	page 4-159
		K_SetStartStopG[3]	page 4-166
Channel-Gain Queue	0 (NULL)	K_SetChnGAry[3]	page 4-143

Table 2-1. A/D Frame Elements (cont.)

Element	Default Value	Setup Function	Page Number
Conversion Mode	Paced	K_SetADFreeRun	page 4-127
		K_ClrADFreeRun[2]	page 4-39
SSH Mode	Disabled	K_SetSSH	page 4-162
Clock Source	Internal	K_SetClk	page 4-146
Pacer Clock Rate[1]	0	K_SetClkRate	page 4-148
Burst Clock Rate	0	K_SetBurstTicks	page 4-139
Trigger Source	Internal	K_SetTrig	page 4-169
Trigger Type	Digital	K_SetADTrig	page 4-129
		K_SetDITrig	page 4-153
Trigger Channel	0	K_SetADTrig	page 4-129
Trigger Polarity	Positive edge	K_SetADTrig	page 4-129
		K_SetDITrig	page 4-153
Trigger Level	0	K_SetADTrig	page 4-129
Trigger Hysteresis	0	K_SetTrigHyst	page 4-171

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.

[3] Not applicable to DAS-1200 Series boards.

Memory Allocation and Management

Synchronous-mode, interrupt-mode, and DMA-mode analog input operations require memory in which to store acquired data. The ways you can allocate and manage memory are described in the following sections.

Dimensioning a Local Array

The simplest way to reserve memory is to dimension an array within your program's memory area. The advantage of this method is that the array is directly accessible to your program. The limitations of this method are as follows:

- Certain programming languages limit the size of local arrays.
- Local arrays are not recommended for DMA-mode operations.
- Local arrays occupy permanent memory areas; these memory areas cannot be freed to make them available to other programs or processes.
- You cannot use local arrays with Windows 95, 32-bit programs.

Since the DAS-1600/1400/1200 Series Function Call Driver stores data in 16-bit integers, you must dimension all local arrays as integers.

Dynamically Allocating a Memory Buffer

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

- The size of the buffer is limited only by the amount of free physical memory available in your computer at run-time.
- You can free a dynamically allocated memory buffer to make it available to other programs or processes.

The limitation of this method is that, for BASIC and Visual Basic for Windows, data in a dynamically allocated memory buffer is not directly accessible to your program. You must use the **K_MoveBufToArray** function to move the data from the dynamically allocated buffer to a local array within your program; refer to page 4-116 for more information.

Use the **K_IntAlloc** function to dynamically allocate a memory buffer for synchronous-mode and interrupt-mode operations; use the **K_DMAAlloc** function to dynamically allocate a memory buffer for DMA-mode operations. Specify the number of samples to store in the buffer (up to 5,000,000 for **K_IntAlloc** and up to 32,767 for **K_DMAAlloc**). The driver returns the starting address of the buffer and a unique identifier for the buffer (this identifier is called the memory handle).

If you no longer require the buffer, free the buffer for another use by specifying the memory handle in the **K_IntFree** function (for synchronous-mode and interrupt-mode operations) or the **K_DMAFree** function (for DMA-mode operations).

Notes: If you are writing Windows 95, 32-bit programs, you must install the Keithley Memory Manager. Refer to your board user's guide for information.

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

For DOS-based languages, the **K_IntAlloc** and **K_DMAAlloc** functions use the DOS Int 21H function 48H to dynamically allocate far heap memory. For Windows-based languages, the **K_IntAlloc** and **K_DMAAlloc** functions call the **GlobalAlloc** API function to allocate the desired buffer size from the global heap.

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

To eliminate page wrap conditions and to guarantee that dynamically allocated memory is suitable for use by the computer's 8237 DMA controller, **K_DMAAlloc** may allocate an area twice as large as actually needed. Once the data in this buffer is processed and/or saved elsewhere, use **K_DMAFree** to free the memory for other uses.

Assigning a Starting Address

After you dimension your array or allocate your buffer, you must assign the starting address of the array or buffer and the number of samples to store in the array or buffer.

Each supported programming language requires a particular procedure for dimensioning local arrays, allocating a memory buffer, and assigning the starting address; refer to the following pages for more information:

C/C++	page 3-3
Pascal	page 3-11
Visual Basic for Windows	page 3-16
BASIC	page 3-24

Gains and Ranges

Each channel on a DAS-1600/1400 Series board can measure analog input signals in one of four, software-selectable unipolar or bipolar analog input ranges; the input range type (unipolar or bipolar) is switch-selectable. Each channel on a DAS-1200 Series board can measure analog input signals in one of four, switch-selectable bipolar analog input ranges.

Table 2-2 lists the analog input ranges supported by DAS-1600/1400/1200 Series boards and the gain and gain code associated with each range. Gain codes are used by the Function Call Driver to represent the gain.

Table 2-2. Analog Input Ranges

Boards	Analog Input Range		Gain	Gain Code
	Bipolar	Unipolar		
DAS-1601 DAS-1401	±10.0 V	0.0 to +10.0 V	1	0
	±1.0 V	0.0 to +1.0 V	10	1
	±100 mV	0 to +100 mV	100	2
	±20 mV	0 to +20 mV	500	3
DAS-1602 DAS-1402	±10.0 V	0.0 to +10.0 V	1	0
	±5.0 V	0.0 to +5.0 V	2	1
	±2.5 V	0.0 to +2.5 V	4	2
	±1.25 V	0.0 to +1.25 V	8	3
DAS-1201[1]	±5.0 V	Not available	1	Not applicable
	±0.5 V	Not available	10	Not applicable
	±0.05 V	Not available	100	Not applicable
	±0.01 V	Not available	500	Not applicable
DAS-1202[1]	±5.0 V	Not available	1	Not applicable
	±2.5 V	Not available	2	Not applicable
	±1.25 V	Not available	4	Not applicable
	±0.625 V	Not available	8	Not applicable

Notes
[1] The gains of DAS-1200 Series boards are switch-selectable. You do not specify the gain through software.

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

For a synchronous-mode, interrupt-mode, or DMA-mode operation, you specify the gain code in the **K_SetG** or **K_SetStartStopG** function; the function you use depends on how you specify the logical channels, as described in the following section.

Channels

DAS-1600/1400/1200 Series boards are switch-configurable for either 16 single-ended analog input channels (numbered 0 through 15) or eight differential analog input channels (numbered 0 through 7).

The driver determines the channel configuration (single-ended or differential) by reading the configuration file. If desired, you can use the **K_GetADConfig** function to get the channel configuration by reading the switches on the board.

If you require more than the 16 single-ended or eight differential onboard channels, you can use any combination of up to eight 16-channel EXP-16 or EXP-16/A expansion accessories, and/or 8-channel EXP-GP expansion accessories to increase the number of available channels to 128, or you can use up to 16 16-channel EXP-1600 expansion accessories to increase the number of available channels to 256. You can also use up to four MB02 backplanes to increase the number of available channels to 76.

Note: You cannot perform DMA-mode operations on channels on EXP-16, EXP-16/A, EXP-GP, or EXP-1600 expansion accessories.

You assign expansion accessories to consecutive onboard analog input channels, beginning with onboard channel 0. To ensure that the DAS-1600/1400/1200 Series Function Call Driver reads the channel numbers correctly, you must attach all EXP-16s and EXP-16/As first, followed by all EXP-GPs, then all EXP-1600s. You can also use the remaining onboard channels. Refer to your board user's guide and to the documentation provided with your expansion accessories for more information.

The maximum supported configuration is eight EXP-16s or EXP-16/As, eight EXP-GPs, 16 EXP-1600s, or four MB02 backplanes. Table 2-3 lists the software (or logical) channels associated with each expansion accessory.

Table 2-3. Channels in Maximum Configuration

Onboard Channel	Software (Logical) Channels			
	EXP-16 / EXP-16/A	EXP-GP	EXP-1600	MB02
0	0 to 15	0 to 7	0 to 15	0 to 15
1	16 to 31	8 to 15	16 to 31	16 to 31
2	32 to 47	16 to 23	32 to 47	32 to 47
3	48 to 63	24 to 31	48 to 63	48 to 63
4	64 to 79	32 to 39	64 to 79	64
5	80 to 95	40 to 47	80 to 95	65
6	96 to 111	48 to 55	96 to 111	66
7	112 to 127	56 to 63	112 to 127	67
8	Not available	64	128 to 143	68
9	Not available	65	144 to 159	69
10	Not available	66	160 to 175	70
11	Not available	67	176 to 191	71
12	Not available	68	192 to 207	72
13	Not available	69	208 to 223	73
14	Not available	70	224 to 239	74
15	Not available	71	240 to 255	75

Figure 2-2 illustrates the use of one EXP-16, two EXP-GPs, one EXP-1600, and the 12 remaining onboard channels on a DAS-1600/1400/1200 Series board configured for single-ended mode. The physical channels on the EXP-16 attached to analog input channel 0 are referred to in software as logical channels 0 to 15; the physical channels on the EXP-GP attached to analog input channel 1 are referred to in software as logical channels 16 to 23; the physical channels on the EXP-GP attached to analog input channel 2 are referred to in software as logical channels 24 to 31; the physical channels on the EXP-1600 attached to analog input channel 3 are referred to in software as logical

channels 32 to 47; the remaining 12 onboard analog input channels (4 through 15) are referred to in software as logical channels 48 through 59.

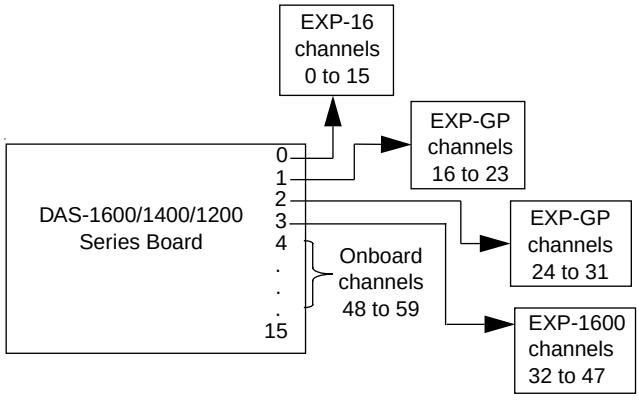


Figure 2-2. Analog Input Channels

Note: The configuration utility CFG1600.EXE is useful in determining logical channel assignments for a given expansion accessory arrangement.

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

Specifying a Single Channel

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.

Refer to Table 2-2 on page 2-13 for a list of the analog input ranges supported by DAS-1600/1400 Series boards and the gain code associated with each range. Note that the gain code is not applicable to DAS-1200 Series boards.

Specifying 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.

For example, assume that you have an EXP-16/A attached to onboard channel 0 on a DAS-1600/1400/1200 Series board configured for single-ended mode. You specify the start channel as 14, the stop channel as 17, and you want to acquire five samples. Your program reads data first from channels 14 and 15 (on the EXP-16/A), then from channels 16 and 17 (onboard channels 1 and 2), and finally from channel 14 again.

You can specify a start channel that is higher than the stop channel in the following cases:

- You are not using any expansion accessories.
- The analog input channels are configured as single-ended.

For example, assume that the start channel is 15, the stop channel is 2, and you want to acquire five samples. Your program reads data first from channel 15, then from channels 0, 1, and 2, and finally from channel 15 again.

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.

Refer to Table 2-2 on page 2-13 for a list of the analog input ranges supported by DAS-1600/1400 Series boards and the gain code associated

with each range. Note that the gain code is not applicable to DAS-1200 Series boards.

Specifying Channels in a Channel-Gain Queue

For synchronous-mode and interrupt-mode analog input operations on DAS-1600/1400 Series boards, you can acquire samples from channels in a software channel-gain queue. In the channel-gain queue, you specify the channels you want to sample, the order in which you want to sample them, and 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 specified number of samples is read.

Refer to Table 2-2 on page 2-13 for a list of the analog input ranges supported by DAS-1600/1400 Series boards and the gain code associated with each range.

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

C/C++	page 3-5
Pascal	page 3-14
Visual Basic for Windows	page 3-19
BASIC	page 3-28

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 cannot use a channel-gain queue with DMA-mode operations or with DAS-1200 Series boards.

Conversion Modes

The conversion mode determines how the board regulates the timing of conversions when you are acquiring multiple samples from a single channel or from a group of multiple channels (known as a scan). You can specify paced mode, burst mode, or burst mode with SSH, as described in the following sections. Refer to your board user's guide for more information about conversion modes.

Paced Mode

You can specify paced mode for a synchronous-mode, interrupt-mode, or DMA-mode analog input operation. Use paced mode if you want to accurately control the period between conversions of individual channels in a scan. Paced mode is the default conversion mode. To reset the conversion mode to paced mode, use the **K_ClrADFreeRun** function.

Burst Mode

You can specify burst mode for a DMA-mode analog input operation only. Use burst mode if you want to accurately control the period between conversions of the entire scan. Use the **K_SetADFreeRun** function to specify burst mode.

By default, conversions of individual channels in a scan are performed as quickly as possible for the specified gain. Table 2-4 lists the default settling times and burst mode conversion rates for each DAS-1600/1400/1200 Series board gain.

Table 2-4. Default Settling Times

Board	Gain	Settling Time	Burst Mode Conversion Rate
DAS-1601 DAS-1401	1	10 µs	100 kHz
	10	10 µs	100 kHz
	100	14 µs	71.42 kHz
	500	34 µs	29.4 kHz

Table 2-4. Default Settling Times (cont.)

Board	Gain	Settling Time	Burst Mode Conversion Rate
DAS-1602 DAS-1402 DAS-1202	1	10 µs	100 kHz
	2	10 µs	100 kHz
	4	10 µs	100 kHz
	8	10 µs	100 kHz
DAS-1201	1	22 µs	45.45 kHz
	10	22 µs	45.45 kHz
	100	22 µs	45.45 kHz
	500	102 µs	9.8 kHz

In some cases, you may want to adjust the rate of conversions of individual channels in a scan (called the burst mode conversion rate or settling time) to slow the data acquisition rate. For example, in computers with a built-in memory cache, caching takes precedence over DMA operations and data can be lost if you try to acquire data too quickly. You can adjust the settling time by specifying a count value using the **K_SetBurstTicks** function; you can use any count value between 2 and 255.

Use the following formula to determine the appropriate count value:

$$\text{Count} = \frac{\text{Settling Time (in microseconds)} - 2}{4}$$

For example, if you want a settling time of 30 µs, specify a count of 7 as shown in the following equation:

$$\frac{(30 - 2)}{4} = \frac{28}{4} = 7$$

Note: You cannot specify burst mode for a synchronous-mode or interrupt-mode operation.

Burst Mode with SSH

You can specify burst mode with SSH for a DMA-mode analog input operation only. Use burst mode with SSH if you are using an SSH-4/A or SSH-8 accessory to simultaneously sample all channels in a scan and you want to accurately control the period between conversions of the entire scan. Use the **K_SetSSH** function to specify burst mode with SSH.

By default, conversions of individual channels in a scan are performed as quickly as possible for the specified gain. Refer to Table 2-4 on page 2-20 for a list of the default settling times.

Notes: You cannot specify burst mode with SSH for a synchronous-mode or interrupt-mode operation.

If you use an SSH-8 accessory, one extra count is required to allow the SSH-8 to sample and hold the values. Refer to the SSH-8 accessory documentation for more information.

Pacer Clocks

You can specify a pacer clock for a synchronous-mode, interrupt-mode, or DMA-mode operation. In paced mode, the pacer clock determines the period between the conversion of one channel and the conversion of the next channel. In burst mode or burst mode with SSH, the pacer clock determines the period between the conversions of one scan and the conversions of the next scan.

You can specify the internal pacer clock or an external pacer clock, as described in the following sections; refer to your board user's guide for more information.

Note: 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.

Internal Pacer Clock

The internal pacer clock uses two cascaded counters of the onboard 82C54 counter/timer. The counters are normally in an idle state. When you start the analog input operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), a conversion is initiated. Note that a slight delay occurs between when you start the operation and when the conversion is initiated.

After the first conversion is initiated, the counters are loaded with a count value and begin counting down. When the counters count down to 0, another conversion is initiated and the process repeats.

If the 10 MHz time base is specified in the configuration file, each count represents 0.1 μs; if the 1 MHz time base is specified in the configuration file, each count represents 1.0 μs. Use the **K_SetClkRate** function to specify the number of counts (clock ticks) between conversions. For example, if you specify a count of 100 with a 10 MHz time base, the period between conversions is 10 μs (100 ksamples/s); if you specify a count of 87654, the period between conversions is 8.8 ms (114.1 samples/s).

You can specify a count between 100 and 4,294,967,295 for the 10 MHz time base and between 10 and 4,294,967,295 for the 1 MHz time base. The period between conversions ranges from 10 μs to 7.16 minutes (for the 10 MHz time base) and from 10 μs to 71.6 minutes (for the 1 MHz time base).

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 are using the 10 MHz time base and want a conversion rate of 10 ksamples/s, specify a count of 1000, as shown in the following equation:

$$\frac{10,000,000}{10,000} = 1000$$

The internal pacer clock is the default pacer clock. To reset the pacer clock source to an internal pacer clock, use the **K_SetClk** function.

Note: To avoid overrun errors when using the internal pacer clock with a DAS-1201 Series board, specify a count value of at least 200 for the 10 MHz time base and at least 20 for the 1 MHz time base.

External Pacer Clock

You connect an external pacer clock to the IP0/TRIG0/XPCLK pin (25) on the main I/O connector (J1).

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

Use the **K_SetClk** function to specify an external pacer clock.

Note: The analog-to-digital converter (ADC) can acquire samples at a maximum of 100 ksamples/s (one sample every 10 µs) for the DAS-1601, DAS-1602, DAS-1401, DAS-1402, and DAS-1202 boards or 50 ksamples/s (one sample every 20 µs) for the DAS-1201 board. If you are using an external pacer clock, make sure that the clock initiates conversions at a rate that the ADC can handle.

Buffering Modes

The buffering mode determines how the driver stores the converted data in the buffer. For an interrupt-mode or DMA-mode analog input operation, you can specify single-cycle or continuous buffering mode, as described in the following sections.

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

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. To reset the buffering mode to single-cycle mode, use the **K_ClrContRun** function.

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.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For a synchronous-mode, interrupt-mode, or DMA-mode analog input operation, you can specify an internal trigger, an external analog trigger, or an external digital trigger, as described in the following sections.

The trigger event is not significant until the operation has been started (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**). The point at which conversions begin relative to the trigger event depends on the pacer clock; refer to page 2-22 for more information.

Internal Trigger

An internal trigger is a software trigger. The trigger event occurs when you start the operation. Note that a slight delay occurs between the time you start the operation and the time the trigger event occurs.

The internal trigger is the default trigger source. To reset the trigger source to an internal trigger, use the **K_SetTrig** function.

External Analog Trigger

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

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

Figure 2-3 illustrates these analog trigger conditions, where the specified voltage level is +5 V. Note that a slight delay occurs between the time the trigger condition is met and the time the driver realizes the trigger condition is met and begins conversions.

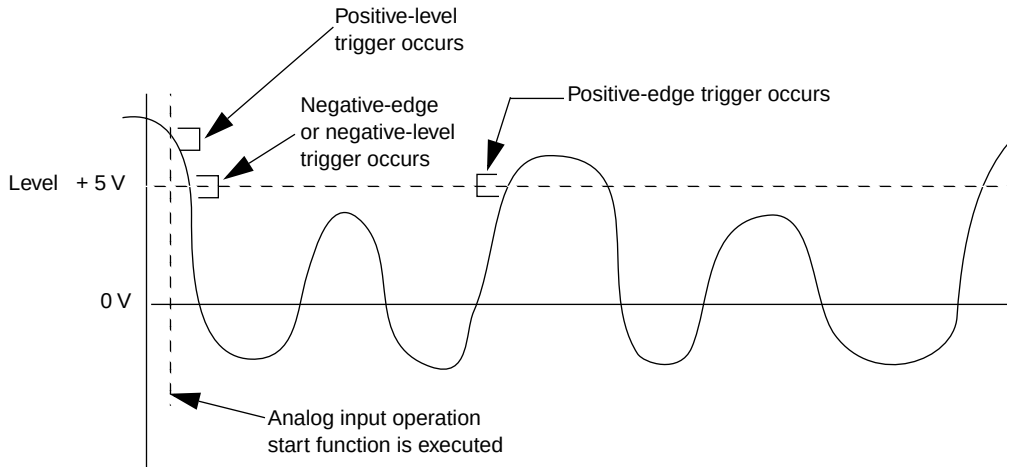


Figure 2-3. Analog Trigger Conditions

Use the **K_SetTrig** function to specify an external trigger. Then, use the **K_SetADTrig** function to specify the following:

- **Analog input channel to use as the trigger channel** - The trigger channel always measures signals at a gain of 1.
- **Voltage level** - You specify the voltage level as a count value between 0 and 4095. Refer to Appendix B for information on how to convert a voltage value to a count value.
- **Trigger polarity and sensitivity** - The trigger can be a positive-edge, negative-edge, positive-level, or negative-level trigger.

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

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

The hysteresis value is an absolute number, which you specify as a count value between 0 and 4095. When you add the hysteresis value to the voltage level (for a negative-edge trigger) or subtract the hysteresis value from the voltage level (for a positive-edge trigger), the resulting value must also be between 0 and 4095.

For example, assume that you are using a negative-edge trigger on a channel on a DAS-12020 Series board configured for an analog input range of ±5 V. If the voltage level is +4.8 V (4014 counts), you can specify a hysteresis value of 0.1 V (41 counts) because 4014 + 41 is less than 4095, but you cannot specify a hysteresis value of 0.3 V (123 counts) because 4014 + 123 is greater than 4095. Refer to Appendix B for information on how to convert a voltage value to a count value.

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

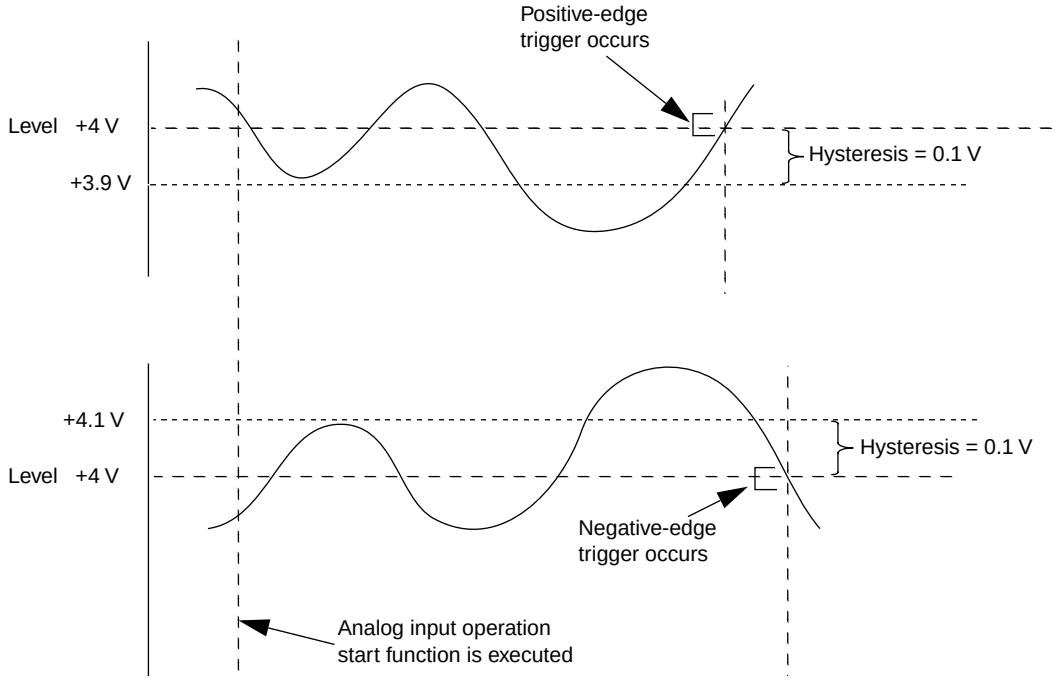


Figure 2-4. Using a Hysteresis Value

Note: The analog trigger is a software-based trigger. When you start the analog input operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), the driver samples the specified trigger channel until the trigger condition is met. Control does not return to your program until the trigger condition is met. (To terminate the operation if a trigger event does not occur, press **Ctrl+Break**.)

External Digital Trigger

An external digital trigger occurs when one of the following occurs on the digital trigger signal connected to the IP1/XTRIG pin (6) on the main I/O connector:

- A rising edge on the IP1/XTRIG pin (positive-edge trigger).
- A falling edge on the IP1/XTRIG pin (negative-edge trigger).
- The signal is high on the IP1/XTRIG pin (positive-level trigger).
- The signal is low on the IP1/XTRIG pin (negative-level trigger).

Use the **K_SetTrig** function to specify an external trigger. Then, use the **K_SetDITrig** function to specify the trigger conditions. The trigger conditions are illustrated in Figure 2-5. Note that a slight delay occurs between the time the trigger condition is met and the time the driver realizes the trigger condition is met and begins conversions.

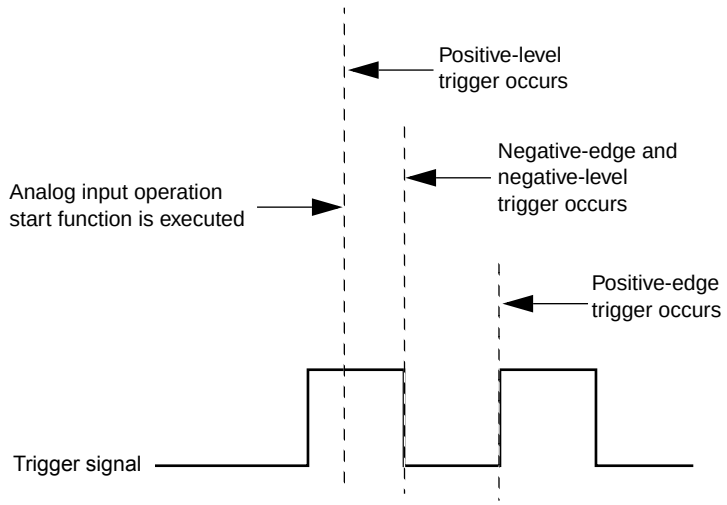


Figure 2-5. Digital Trigger Conditions

Note: The external digital trigger is a software-based trigger. When you start the analog input operation (using **K_SyncStart**, **K_IntStart**, or **K_DMAStart**), the driver reads the signal connected to the IP1/XTRIG pin until the trigger condition is met. Control does not return to your program until the trigger condition is met. (To terminate the operation if a trigger event does not occur, press **Ctrl+Break**.)

Analog Output Operations (DAS-1600 Series Only)

This section describes analog output operations. It includes information on the operation modes available, how to access a frame, how to allocate and manage memory, and how to specify channels, the pacer clock source, the buffering mode, and the digital trigger conditions for an analog output operation.

Operation Modes

The operation mode determines which attributes you can specify for an analog output operation and how values are written from computer memory to the board. You can perform an analog output operation in single mode, synchronous mode, or interrupt mode, as described in the following sections.

Single Mode

In single mode, the driver writes a single value to one or both analog output channels; you cannot perform any other operation until the single-mode operation is complete.

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

You specify the analog output value as a count value. Refer to Appendix B for information on converting a voltage value to a count value.

Note: The hardware does not support simultaneous updating of the DACs. However, if you specify both analog output channels, the channels are updated as close to simultaneously as possible. When you call **K_DAWrite**, channel 0 is updated; channel 1 is updated several microseconds later.

Synchronous Mode

Synchronous mode provides the fastest means of updating the analog output channels. In synchronous mode, the driver writes a single value or multiple values from a user-defined buffer in computer memory to one or both analog output channels. A hardware pacer clock paces the updates of the channels. After the driver writes the specified number of values, the driver returns control to the program. You cannot perform any other operation until the synchronous-mode operation is complete.

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

You specify the analog output values as count values. Refer to Appendix B for information on converting voltage values to count values.

Interrupt Mode

In interrupt mode, the driver writes a single value or multiple values from a user-defined buffer in computer memory to one or both analog output channels. A hardware clock paces the updating of the analog output channels. Once the analog output operation begins, control returns to your program. The driver continues to write values to the analog output channels using an interrupt service routine.

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 2-38 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.

You specify the analog output values as count values. Refer to Appendix B for information on converting voltage values to count values.

Frames

Synchronous-mode and interrupt-mode analog output operations require frames. Use the **K_GetDAFrame** function to access an analog output frame, called a D/A (digital-to-analog) frame. The driver returns the frame handle for the frame. Refer to page 2-6 for more information about frames.

Table 2-5 lists the elements of a D/A frame, the default value of each element, the setup functions used to define each element, and the page(s) in this manual on which to find additional information.

Table 2-5. D/A Frame Elements

Element	Default Value	Setup Function	Page Number
Buffer[1]	0 (NULL)	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
Number of Samples	0	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
Buffering Mode	Single-cycle	K_SetContRun	page 4-151
		K_ClrContRun[2]	page 4-41
Start Channel	0	K_SetChn	page 4-141
		K_SetStartStopChn	page 4-164
Stop Channel	0	K_SetStartStopChn	page 4-164
Clock Source	Internal	K_SetClk	page 4-146

Table 2-5. D/A Frame Elements (cont.)

Element	Default Value	Setup Function	Page Number
Pacer Clock Rate[1]	0	K_SetClkRate	page 4-148
Trigger Source	Internal	K_SetTrig	page 4-169
Trigger Type	Digital	K_SetDITrig	page 4-153

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.

Memory Allocation and Management

Synchronous-mode and interrupt-mode analog output operations require memory in which to store the data that is written to the analog output channels.

Since analog output operations typically require small arrays of data, you can reserve memory by dimensioning a local array within your program's memory area. Since the Function Call Driver writes data as 16-bit integers, you must dimension all local arrays as integers.

If you are using both analog output channels, when you start the analog output operation (using **K_SyncStart** or **K_IntStart**), the driver writes the first value in the array to the first channel and the second value in the array to the second channel. To ensure predictable results, make sure that the number of values stored in the array is a multiple of 2. For example, if you are using both analog output channels, you can dimension an array of 100 values, but you should not dimension an array of 75 values.

After you dimension your array, you must assign the starting address of the array and the number of samples stored in the array. Each supported programming language requires a particular procedure for dimensioning an array and assigning the starting address; refer to the following pages for more information:

C/C++	page 3-4
Pascal	page 3-13
Visual Basic for Windows	page 3-19
BASIC	page 3-28

You can also use the **K_IntAlloc** function to dynamically allocate a memory buffer, if desired. Specify the number of values you want to store in the buffer (up to a maximum of 5,000,000). The driver returns the starting address of the buffer and the memory handle for the buffer. If you no longer require the buffer, free the buffer for another use by specifying the memory handle in the **K_IntFree** function.

For BASIC and Visual Basic for Windows, data in a dynamically allocated memory buffer is not directly accessible to your program. You must use the **K_MoveArrayToBuf** function to move this data from a local array within your program to the dynamically allocated buffer; refer to page 4-112 for more information.

Note: You cannot use a local array with Windows 95, 32-bit programs; you must use **K_IntAlloc** to dynamically allocate a memory buffer. You must also install the Keithley Memory Manager; refer to your board user's guide for information.

Channels

DAS-1600 Series boards contain two digital-to-analog converters (DACs), each of which is associated with an analog output channel. You can perform an analog output operation on a single channel or on both channels.

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

For synchronous-mode and interrupt-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.

For synchronous mode and interrupt mode, you can also write a single value or multiple values to both analog output channels. Use the **K_SetStartStopChn** function to specify channel 0 as the start channel and channel 1 as the stop channel. At each pulse of the pacer clock, the driver writes a new value to both channels.

For example, assume that your array contains two waveforms (0, 4095, 1, 4094, 2, 4093 . . . 0, 4095). At the first pulse of the pacer clock, the driver writes 0 to channel 0 and 4095 to channel 1, at the next pulse of the pacer clock, the driver writes 1 to channel 0 and 4094 to channel 1, and so on.

Pacer Clocks

When performing a synchronous-mode or interrupt-mode analog output operation, you can use a pacer clock to determine the period between updates of the analog output channels. You can specify the internal pacer clock or an external pacer clock, as described in the following sections.

Note: The actual rate at which the analog output channels are updated also depends on other factors, including your computer, the operating system/environment, and other software issues.

Internal Pacer Clock

The internal pacer clock uses two cascaded counters of the onboard 82C54 counter/timer. The counters are normally in an idle state. When you start the analog output operation (using **K_SyncStart** or **K_IntStart**), the specified analog output channels are updated. Note that a slight delay occurs between when you start the operation when the channels are updated.

The counters are loaded with a count value and begin counting down. When the counters count down to 0, the analog output channels are updated again and the process repeats.

If the 10 MHz time base is specified in the configuration file, each count represents 0.1 µs; if the 1 MHz time base is specified in the configuration file, each count represents 1.0 µs. Use the **K_SetClkRate** function to specify the number of counts (clock ticks) between updates. For example, if you specify a count of 2000 with a 10 MHz time base, the period between updates is 200 µs (5 ksamples/s); if you specify a count of 87654, the period between updates is 8.8 ms (114.1 samples/s).

You can specify a count between 100 and 4,294,967,295 for the 10 MHz time base and between 10 and 4,294,967,295 for the 1 MHz time base. The period between updates ranges from 10 µs to 7.16 minutes (for the 10 MHz time base) and from 10 µs to 71.6 minutes (for the 1 MHz time base).

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 are using the 10 MHz time base and want an update rate of 1 ksample/s, specify a count of 10,000, as shown in the following equation:

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

The internal pacer clock is the default pacer clock. To reset the pacer clock source to an internal pacer clock, use the **K_SetClk** function.

Notes: The hardware does not support simultaneous updating of the DACs. However, if you specify both analog output channels (using **K_SetStartStopChn**), the channels are updated as close to simultaneously as possible. Each time the counters of the internal pacer clock count down to 0, channel 0 is updated; channel 1 is updated several microseconds later.

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

The driver accepts a count value as low as 10 for the 1 MHz time base and as low as 100 for the 10 MHz time base. However, a low count value may cause an overrun error. The maximum observed update rates for the internal pacer clock are 1 ksamples/s when running under Windows and 5 ksamples/s when running under DOS.

External Pacer Clock

You connect an external pacer clock to the IP0/TRIG0/XPCLK pin (25) on the main I/O connector (J1).

At the next rising edge of the external pacer clock after you start an analog output operation (using **K_SyncStart** or **K_IntStart**) and at every subsequent rising edge of the external pacer clock, the specified analog output channels are updated. Note that a slight delay may occur between the rising edge of the external pacer clock and the update of the channels.

Use the **K_SetClk** function to specify an external pacer clock.

Note: The hardware does not support simultaneous updating of the DACs. However, if you specify both analog output channels (using **K_SetStartStopChn**), the channels are updated as close to simultaneously as possible. At each rising edge of the external pacer clock, channel 0 is updated; channel 1 is updated several microseconds later.

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

Buffering Modes

The buffering mode determines how the driver writes the values in the buffer to the analog output channels. For interrupt-mode analog output operations, you can specify single-cycle or continuous buffering mode, as described in the following sections.

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

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. To reset the buffering mode to single-cycle mode, use the **K_ClrContRun** function.

Continuous Mode

In continuous mode, the driver continuously writes values from the buffer until the 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.

Triggers

A trigger is an event that occurs based on a specified set of conditions. For synchronous-mode and interrupt-mode analog output operations, you can specify an internal trigger or an external digital trigger, as described in the following sections.

The trigger event is not significant until the operation has been started (using **K_SyncStart** or **K_IntStart**). The point at which an analog output channel is updated depends on the pacer clock; refer to page 2-35 for more information.

Internal Trigger

An internal trigger is a software trigger. The trigger event occurs when you start the analog output operation. Note that a slight delay occurs between the time you start the operation and the time the trigger event occurs.

The internal trigger is the default trigger source. To reset the trigger source to an internal trigger, use the **K_SetTrig** function.

External Digital Trigger

An external digital trigger occurs when one of the following occurs on the digital trigger signal connected to the IP1/XTRIG pin (6) on the main I/O connector:

- A rising edge on the IP1/XTRIG pin (positive-edge trigger).
- A falling edge on the IP1/XTRIG pin (negative-edge trigger).
- The signal is high on the IP1/XTRIG pin (positive-level trigger).
- The signal is low on the IP1/XTRIG pin (negative-level trigger).

Use the **K_SetTrig** function to specify an external trigger. Then, use the **K_SetDITrig** function to specify the digital trigger conditions. The trigger conditions are illustrated in Figure 2-5 on page 2-29.

Note: The external digital trigger is a software-based trigger. When you start the analog output operation (using **K_SyncStart** or **K_IntStart**), the driver reads the signal connected to the IP1/XTRIG pin until the trigger condition is met. Control does not return to your program until the trigger condition is met. (To terminate the operation if a trigger event does not occur, press **Ctrl+Break**.) In addition, a slight delay occurs between the time the trigger condition is met and the time the driver realizes the trigger condition is met and begins updating the analog output channel.

Digital I/O Operations

This section describes digital I/O operations. It includes information on the operation modes available, how to access a frame, how to allocate and manage memory, how to use the digital I/O channel, and how to specify the pacer clock source, the buffering mode, and the digital trigger conditions for a digital I/O operation.

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 single mode, synchronous mode, or interrupt mode, as described in the following sections.

Single Mode

In a single-mode digital input operation, the driver reads the value of digital input channel 0 once; in a single-mode digital output operation, the driver writes a value to digital output channel 0 once. You cannot perform any other operation until the single-mode operation is complete.

Use the **K_DIRead** function to perform 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 perform 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

Synchronous mode provides the fastest means of performing a digital I/O operation. In a synchronous mode digital input operation, the driver reads the value of digital input channel 0 multiple times; in a synchronous mode digital output operation, the driver writes a single value or multiple values to digital output channel 0 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 digital input channel 0 multiple times; in an interrupt-mode digital output operation, the driver writes a single value or multiple values to digital output channel 0 multiple times.

A hardware clock paces the digital I/O operation. Once the digital I/O operation begins, control returns to your program. The driver continues to read values from or write values to the digital I/O channel using an interrupt service routine.

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 2-51 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.

Frames

Synchronous-mode and interrupt-mode digital I/O operations require frames. Use the **K_GetDIFrame** function to access a digital input frame, called a DI frame; use the **K_GetDOFrame** function to access a digital output frame, called a DO frame. The driver returns the frame handle for the frame. Refer to page 2-6 for more information about frames.

Table 2-6 lists the elements of a DI frame; Table 2-7 lists the elements of a DO frame. The tables also list the default value of each element, the setup functions used to define each element, and the page(s) in this manual on which to find additional information.

Table 2-6. DI Frame Elements

Element	Default Value	Setup Function	Page Number
Buffer[1]	0 (NULL)	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetBufL	page 4-137
Number of Samples	0	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetBufL	page 4-137
Buffering Mode	Single-cycle	K_SetContRun	page 4-151
		K_ClrContRun[2]	page 4-41
Clock Source	Internal	K_SetClk	page 4-146
Pacer Clock Rate[1]	0	K_SetClkRate	page 4-148
Trigger Source	Internal	K_SetTrig	page 4-169
Trigger Type	Digital	K_SetDITrig	page 4-153

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 2-7. DO Frame Elements

Element	Default Value	Setup Function	Page Number
Buffer[1]	0 (NULL)	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetBufL	page 4-137
Number of Samples	0	K_SetBuf	page 4-132
		K_SetBufI	page 4-135
		K_SetBufL	page 4-137
Buffering Mode	Single-cycle	K_SetContRun	page 4-151
		K_ClrContRun[2]	page 4-41
Clock Source	Internal	K_SetClk	page 4-146
Pacer Clock Rate[1]	0	K_SetClkRate	page 4-148
Trigger Source	Internal	K_SetTrig	page 4-169
Trigger Type	Digital	K_SetDITrig	page 4-153

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.

Memory Allocation and Management

Synchronous-mode and interrupt-mode digital I/O operations require memory in which to store the data that is read or written.

Since digital I/O operations typically require small arrays of data, you can reserve memory by dimensioning a single local array within your program's memory area. Your array must be able to accommodate the digital I/O lines you are using. Table 2-8 lists the types of arrays you can dimension. The configuration of the digital I/O lines is discussed in the next section.

Table 2-8. Dimensioning Arrays for Digital I/O Operations

Number of Digital I/O Lines	Type of Array
4	Byte (8 bits)
8	
12	Integer (16 bits)
16	
20	Long (32 bits)
24	
28	

For example, if you are using the 24 bidirectional bits for digital input and you want to read the bits five times, dimension an array of five long-type variables. If you are using 12 bits for digital output and you want to write to the ports 10 times, dimension an array of 10 integer-type variables.

Note: You cannot dimension a byte-type array in BASIC or Visual Basic for Windows. If you are using four or eight digital I/O lines, dimension an integer-type array instead. Refer to page 3-21 (Visual Basic for Windows) or page 3-29 (BASIC) for information on converting the integer data to data you can use.

After you dimension your array, you must assign the starting address of the array and the number of samples to store in the array. Each supported programming language requires a particular procedure for dimensioning an array and assigning the starting address; refer to the following pages for more information:

C/C++	page 3-4
Pascal	page 3-13
Visual Basic for Windows	page 3-19
BASIC	page 3-28

You can also use the **K_IntAlloc** function to dynamically allocate a memory buffer, if desired. Specify the number of values to store in the buffer (up to a maximum of 32,767). The driver returns the starting address of the buffer and a unique identifier for the buffer (this identifier is called the memory handle). If you no longer require the buffer, free the buffer for another use by specifying the memory handle in the **K_IntFree** function.

For BASIC and Visual Basic for Windows, data in a dynamically allocated memory buffer is not directly accessible to your program. The number of digital I/O lines configured for the digital I/O channel determines the function you should use to move the data to or from the dynamically allocated buffer, as shown in the following table:

Digital I/O Lines	Function	Description	Page
Less than 16 digital input lines	K_MoveBufToArray	Moves digital input data from the buffer to a local integer array in your program.	page 4-116
More than 16 digital input lines	K_MoveBufToArrayL	Moves digital input data from the buffer to a local long array in your program.	page 4-118
Less than 16 digital output lines	K_MoveArrayToBuf	Moves digital output data from a local integer array in your program to the buffer.	page 4-112
More than 16 digital output lines	K_MoveArrayToBufL	Moves digital output data from a local long array in your program to the buffer.	page 4-114

Note: You cannot use a local array with Windows 95, 32-bit programs; you must use **K_IntAlloc** to dynamically allocate a memory buffer. You must also install the Keithley Memory Manager; refer to your board user's guide for information.

Digital Input/Output Channel

DAS-1600/1400/1200 Series boards contain four unidirectional digital input lines and four unidirectional digital output lines that are accessible through the main I/O connector (J1). DAS-1600/1200 Series boards provide an additional 24 bits of bidirectional digital I/O on the PIO cable connector (J2 on the DAS-1600, J4 on the DAS-1200). These 24 bits are configured as follows:

- Port A, 8-bit
- Port B, 8-bit
- Port CH (High), 4-bit
- Port CL (Low), 4-bit

Since each of these four ports is configurable for either input or output, 16 port configurations are available. In any of these configurations, the driver concatenates data from each input port with data from the onboard digital inputs, if they are available, into a composite value on a single digital input channel (channel 0). Similarly, the driver concatenates data from each output port with data from the onboard digital outputs into a composite value on a single digital output channel (channel 0).

Data on digital input channel 0 or digital output channel 0 can be up to 28 bits wide when all ports are configured for one direction (input or output) and the onboard digital lines are available. A value of 1 in the bit position indicates that the input or output is high; a value of 0 in the 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).

Note the following limitations when using the digital I/O lines:

- If you are using an external pacer clock, you cannot use the IP0/TRIG0/XPCLK line for general-purpose digital input operations.
- If you are using an external digital trigger, you cannot use the IP1/XTRIG line for general-purpose digital input operations.
- If you are using an expansion accessory, you cannot use any of the unidirectional digital output lines for general-purpose digital output operations.
- If you are using counter 0 as an external gate, you cannot use IP2/CTR0 GATE for general-purpose digital input operations.

Starting from the least significant bit of the digital I/O channel, Port A uses the first eight bits available, Port B uses the next eight bits available, Port CL uses the next four bits available, Port CH uses the next four bits available, and the unidirectional bits use the next four bits available.

If a particular port is configured for input, none of the bits in the output channel is used; if a particular port is configured for output, none of the bits in the input channel is used.

For example, a DAS-1600/1200 Series board is configured with no EXPs and with Port A, Port B, Port CL, and Port CH all configured for output. Table 2-9 illustrates how the bits in the digital I/O channels are used.

Table 2-9. Digital I/O Channel Usage; No EXPs, All Ports Output

Bits	Output Channel Use	Input Channel Use
0 to 3	Port A	4 unidirectional input bits
4 to 7		
8 to 11	Port B	
12 to 15		
16 to 19	Port CL	
20 to 23	Port CH	
24 to 27	4 unidirectional output bits	

As another example, a DAS-1600/1200 Series board is configured with one or more EXPs and with Port A, Port B, Port CL, and Port CH configured for output. Table 2-10 illustrates how the bits in the digital I/O channels are used. Note that the four unidirectional output bits are dedicated to EXP board control and are not available.

Table 2-10. Digital I/O Channel Usage; EXPs Used, All Ports Output

Bits	Output Channel Use	Input Channel Use
0 to 3	Port A	4 unidirectional input bits
4 to 7		
8 to 11	Port B	
12 to 15		
16 to 19	Port CL	
20 to 23	Port CH	

As another example, a DAS-1600/1200 Series board is configured with no EXPs, with Port A and Port B configured for output, and with Port CL and Port CH configured for input. Table 2-11 illustrates how the bits in the digital I/O channels are used.

Table 2-11. Digital I/O Channel Usage; No EXPs, A and B Output, CL and CH Input

Bits	Output Channel Use	Input Channel Use
0 to 3	Port A	Port CL
4 to 7		Port CH
8 to 11	Port B	4 unidirectional input bits
12 to 15		
16 to 19	4 unidirectional output bits	

As a final example, a DAS-1600/1200 Series board is configured with no EXPs, with Port B and Port CH configured for output, and with Port A and Port CL configured for input. Table 2-12 illustrates how the bits in the digital I/O channels are used.

Table 2-12. Digital I/O Channel Usage; No EXPs, B and CH Output, A and CL Input

Bits	Output Channel Use	Input Channel Use
0 to 3	Port B	Port A
4 to 7		
8 to 11	Port CH	Port CL
12 to 15	4 unidirectional output bits	4 unidirectional input bits

Pacer Clocks

When performing synchronous-mode and interrupt-mode digital I/O operations, you can use a pacer clock to determine the period between reading the digital input channel or writing to the digital output channel.

You can specify the internal pacer clock or an external pacer clock, as described in the following sections.

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

Internal Pacer Clock

The internal pacer clock uses two cascaded counters of the onboard 82C54 counter/timer. The counters are normally in an idle state. When you start the digital I/O operation (using **K_SyncStart** or **K_IntStart**), a value is read or written. Note that a slight delay occurs between when you start the operation and when the value is read or written.

The counters are loaded with a count value and begin counting down. When the counters count down to 0, another value is read or written and the process repeats.

If the 10 MHz time base is specified in the configuration file, each count represents 0.1 µs; if the 1 MHz time base is specified in the configuration file, each count represents 1.0 µs. Use the **K_SetClkRate** function to specify the number of counts (clock ticks) between reads/writes. For example, if you specify a count of 2000 with a 10 MHz time base, the period between reads/writes is 200 µs (5 ksamples/s); if you specify a count of 87654, the period between reads/writes is 8.8 ms (114.1 samples/s).

You can specify a count between 100 and 4,294,967,295 for the 10 MHz time base and between 10 and 4,294,967,295 for the 1 MHz time base. The period between reads/writes ranges from 10 µs to 7.16 minutes (for the 10 MHz time base) and from 10 µs to 71.6 minutes (for the 1 MHz time base).

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 are using the 10 MHz time base and want to write data to digital output channel 0 at a rate of 500 samples/s, specify a count of 20,000, as shown in the following equation:

$$\frac{10,000,000}{500} = 20,000$$

The internal pacer clock is the default pacer clock. To reset the pacer clock source to an internal pacer clock, use the **K_SetClk** function.

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

The driver accepts a count value as low as 10 for the 1 MHz time base and as low as 100 for the 10 MHz time base. However, a low count value may cause an overrun error. The maximum observed read/write rates for the internal pacer clock are 1 ksamples/s when running under Windows and 5 ksamples/s when running under DOS.

External Pacer Clock

You connect an external pacer clock to the IP0/TRIG0/XPCLK pin (25) on the main I/O connector (J1).

At the next rising edge of the external pacer clock after you start a digital I/O operation (using **K_SyncStart** or **K_IntStart**) and at every subsequent rising edge of the external pacer clock, a value is read or written. Note that a slight delay may occur between the rising edge of the external pacer clock and the reading of or writing to the channel.

Use the **K_SetClk** function to specify an external pacer clock.

Note: You cannot use an external pacer clock for a digital I/O operation if the clock is being used by another operation.

Buffering Modes

The buffering mode determines how the driver reads or writes the values in the buffer. For interrupt-mode digital I/O operations, you can specify single-cycle or continuous buffering mode, as described in the following sections.

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

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. To reset the buffering mode to single-cycle mode, use the **K_ClrContRun** function.

Continuous Mode

In a continuous-mode digital input operation, the driver continuously reads digital input channel 0 and stores the values in the buffer until the 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 digital output channel 0 until the 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.

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, you can specify an internal trigger or an external digital trigger, as described in the following sections.

The trigger event is not significant until the operation has been started (using **K_SyncStart** or **K_IntStart**). The point at which a value is read or written depends on the pacer clock; refer to page 2-49 for more information.

Internal Trigger

An internal trigger is a software trigger. The trigger event occurs when you start the digital I/O operation. Note that a slight delay occurs between the time you start the operation and the time the trigger event occurs.

The internal trigger is the default trigger source. To reset the trigger source to an internal trigger, use the **K_SetTrig** function.

External Digital Trigger

An external digital trigger occurs when one of the following occurs on the digital trigger signal connected to the IP1/XTRIG pin (6) on the main I/O connector:

- A rising edge on the IP1/XTRIG pin (positive-edge trigger).
- A falling edge on the IP1/XTRIG pin (negative-edge trigger).
- The signal is high on the IP1/XTRIG pin (positive-level trigger).
- The signal is low on the IP1/XTRIG pin (negative-level trigger).

Use the **K_SetTrig** function to specify an external trigger. Then, use the **K_SetDITrig** function to specify the digital trigger conditions. The trigger conditions are illustrated in Figure 2-5 on page 2-29.

Note: The external digital trigger is a software-based trigger. When you start the digital I/O operation (using **K_SyncStart** or **K_IntStart**), the driver reads the signal connected to the IP1/XTRIG pin until the trigger condition is met. Control does not return to your program until the trigger condition is met. (To terminate the operation if a trigger event does not occur, press **Ctrl+Break**.) In addition, a slight delay occurs between the time the trigger condition is met and the time the driver realizes the trigger condition is met and begins reading or writing a value.

Counter/Timer I/O Operations

DAS-1600/1400/1200 Series boards contain a 82C54 counter/timer; the 82C54 contains three counters: counter 0, counter 1, and counter 2. If these counters are not being used for an internal operation, you can use them for another task, such as frequency measurement.

The DAS-1600/1400/1200 Series Function Call Driver provides the following functions for programming the 82C54 counter/timer:

- **DAS1600_8254Control** - Allows you to write to the 82C54 counter/timer control register.
- **DAS1600_8254SetCounter** - Sets one of the counters on the 82C54 counter/timer.
- **DAS1600_8254SetClk0** - Specifies whether you want counter 0 of the 82C54 counter/timer to use the 100 kHz onboard clock or an external signal connected to the CTR0 CLOCK IN pin (21) of the main I/O connector.
- **DAS1600_8254SetTrig0** - Specifies whether you want the signal at the IP0/TRIG0/XPCLK pin (25) of the main I/O connector to act as a hardware gate for counters 1 and 2.
- **DAS1600_8254GetCounter** - Indicates the current count value of one of the counters on the 82C54 counter/timer.
- **DAS1600_8254GetClk0** - Indicates whether counter 0 of the 82C54 counter/timer is using the 100 kHz onboard clock or an external signal connected to the CTR0 CLOCK IN pin (21) of the main I/O connector.
- **DAS1600_8254GetTrig0** - Indicates whether the signal at the IP0/TRIG0/XPCLK pin (25) of the main I/O connector is acting as a hardware gate for counters 1 and 2.

Refer to Appendix E of your board user's guide for more information on programming the 82C54 counter/timer.

Notes: Counter 0 is always available for general-purpose tasks. If you are using the internal pacer clock, counter 1 and counter 2 are not available for general-purpose tasks. If you are using an external clock source, all three counters are available for general-purpose tasks.

You cannot use the counter/timer functions with Windows 95, 32-bit programs.

3

Programming with the Function Call Driver

This chapter contains a programming overview and language-specific information related to using the Function Call Driver. It includes the following sections:

- **Programming Overview** - an overview of the tasks required to write a program using the DAS-1600/1400/1200 Series Function Call Driver.
- **C/C++ Programming Information** - language-specific information for programming in Microsoft C/C++ (including Visual C++) and Borland C/C++.
- **Pascal Programming Information** - language-specific information for programming in Borland Turbo Pascal (for DOS) and Borland Turbo Pascal for Windows.
- **Visual Basic for Windows Programming Information** - language-specific information for programming in Microsoft Visual Basic for Windows.
- **BASIC Programming Information** - language-specific information for programming in Microsoft QuickBasic, Microsoft Professional Basic, and Microsoft Visual Basic for DOS.

Programming Overview

To write a program using the DAS-1600/1400/1200 Series Function Call Driver, perform the following steps:

1. Define the program'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 program. Refer to the following for additional information:
   - Programming flow diagrams for the preliminary tasks, on page 1-7, which illustrate the programming tasks common to all programs.
   - Programming flow diagrams for an analog input operation, on page 1-8.
   - Programming flow diagrams for an analog output operation, on page 1-14.
   - Programming flow diagrams for a digital input operation, on page 1-18, and for a digital output operation, on page 1-21.
   - Chapter 4, which contains detailed descriptions of the Function Call Driver functions.
   - The example programs in the DAS-1600/1400/1200 Series standard software package and the ASO-1600/1400/1200 software package. The FILES.TXT file in the installation directory lists and describes the example programs.
3. Compile and link the program. Refer to the following for information on compile and link statements and other language-specific considerations:
   - C/C++ Programming Information on page 3-3.
   - Pascal Programming Information on page 3-11.
   - Visual Basic for Windows Programming Information on page 3-16.
   - BASIC Programming Information on page 3-24.
   - The EXAMPLES.TXT file, which provides information on compiling and linking example programs.

C/C++ Programming Information

The following sections contain information you need to reserve memory, to create a channel-gain queue, and to handle errors when programming in C or C++, as well as language-specific information for Microsoft C/C++ (including Visual C++) and Borland C/C++.

Notes: When programming in C/C++, make sure that you use proper typecasting to prevent C/C++ type-mismatch warnings.

When programming in Borland C/C++, make sure that linker options are set so that case-sensitivity is disabled.

Dynamically Allocating a Memory Buffer

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

If you are using a large buffer and programming in a Windows-based language, it is recommended that you install the Keithley Memory Manager before you begin programming. Refer to your board user's guide for more information about the Keithley Memory Manager.

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

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

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

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

Accessing Data from a Dynamically Allocated Memory Buffer

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

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

Note: Declaring pData as a huge pointer allows the program to directly access all data within the memory buffer, regardless of the buffer size. If you declare pData as an integer pointer, after you store 64 KB of data, the data currently in the buffer is overwritten.

Dimensioning a Local Array

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

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

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{
      BYTE Chan;
      char Gain;
   } GainChanAry[256];
} GainChanTable;
```

The following example illustrates how to create a channel-gain queue called MyChanGainQueue for a DAS-1602 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** or **K_IntStart**), channel 0 is sampled at a gain of 1, channel 1 is sampled at a gain of 2, channel 2 is sampled at a gain of 4, and so on.

Handling Errors

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

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

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

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

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

To program in Microsoft C/C++ (for DOS), you need the following files; these files are provided in the ASO-1600/1400/1200 software package.

File	Description
DAS1600.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C
DAS1600.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DAS1600.HPP	Include file when compiling in C++
USE1600.OBJ	Linkable object

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

Type of Compile	Compile and Link Statements
C	CL /c *filename*.c LINK *filename*+use1600.obj,,,das1600+dasrface;
C++	CL /c *filename*.cpp LINK *filename*+use1600.obj,,,das1600+dasrface;

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

The files you need to program in Microsoft C/C++ (for Windows), including Microsoft Visual C++, depend on whether you are writing 16-bit or 32-bit programs. The following files are provided either in the ASO-1600/1400/1200 software package or on the ASO-Win95/32-Bit disk, which is shipped with the ASO-1600/1400/1200 software package.

Program	File	Description
16 bits	DASSHELL.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DAS1600.DLL	Dynamic Link Library of board-specific functions
	DASDECL.H	Include file of Shell function definitions (used when compiling in C or C++)
	DAS1600.H	Include file of board-specific function definitions (used when compiling in C)
	DAS1600.HPP	Include file of board-specific function definitions (used when compiling in C++)
	DASIMP.LIB	Import library of Shell functions
	D1600IMP.LIB	Import library of board-specific functions
32 bits	DASSHL32.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DASSHL16.DLL	Dynamic Link Library of support functions
	DASDECL.H	Include file of Shell function definitions (used when compiling in C or C++)
	DASSHL32.LIB	Import library of Shell functions

To create an executable file in the Microsoft C/C++ (for Windows) environment, perform the following steps. Refer to the documentation supplied with your compiler for complete information.

1. Create a project file.
2. Add all necessary files to the project make file. Make sure that you include *filename*.c (or *filename*.cpp), *filename*.rc, *filename*.def, DASIMP.LIB (or DASSHL32.LIB), and D1600IMP.LIB (16-bit programs only), where *filename* indicates the name of your program.
3. Create a stand-alone executable file (.EXE) that you can execute from within Windows.

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

To program in Borland C/C++ (for DOS), you need the following files; these files are provided in the ASO-1600/1400/1200 software package.

File	Description
DAS1600.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C
DAS1600.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DAS1600.HPP	Include file when compiling in C++
USE1600.OBJ	Linkable object

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

Type of Compile	Compile and Link Statements
C	BCC *filename*.c use1600.obj das1600.lib dasrface.lib
C++	BCC *filename*.cpp use1600.obj das1600.lib dasrface.lib

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

The files you need to program in Borland C/C++ (for Windows) depend on whether you are writing 16-bit or 32-bit programs. The following files are provided either in the ASO-1600/1400/1200 software package or on the ASO-Win95/32-Bit disk, which is shipped with the ASO-1600/1400/1200 software package.

Program	File	Description
16 bits	DASSHELL.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DAS1600.DLL	Dynamic Link Library of board-specific functions
	DASDECL.H	Include file of Shell function definitions (used when compiling in C or C++)
	DAS1600.H	Include file of board-specific function definitions (used when compiling in C)
	DAS1600.HPP	Include file of board-specific function definitions (used when compiling in C++)
	DASIMP.LIB	Import library of Shell functions
	D1600IMP.LIB	Import library of board-specific functions
32 bits	DASSHL32.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DASSHL16.DLL	Dynamic Link Library of support functions
	DASDECL.H	Include file of Shell function definitions (used when compiling in C or C++)
	DASSHL32.LIB	Import library of Shell functions

To create an executable file in the Borland C/C++ environment, perform the following steps. Refer to the documentation supplied with your compiler for complete information.

1. Create a project file.
2. Add all necessary files to the project make file. Make sure that you include *filename*.c (or *filename*.cpp), *filename*.rc, *filename*.def, DASIMP.LIB (or DASSHL32.LIB), and D1600IMP.LIB (16-bit programs only), where *filename* indicates the name of your program.
3. Make sure that you turn OFF both the Case sensitive link and the Case sensitive exports and imports options.
4. Create a stand-alone executable file (.EXE) that you can execute from within Windows.

Pascal Programming Information

The following sections contain information you need to reserve memory, to create a channel-gain queue, and to handle errors when programming in Pascal, as well as language-specific information for Borland Turbo Pascal (for DOS) and Borland Turbo Pascal for Windows.

Reducing the Memory Heap

Note: Reducing the memory heap is recommended for Borland Turbo Pascal (for DOS) only; if you are programming in Borland Turbo Pascal for Windows, reducing the memory heap is not required.

By default, when Borland Turbo Pascal (for DOS) programs begin to run, Pascal reserves all available DOS memory for use by the internal memory manager; this allows you to perform **GetMem** and **FreeMem** operations. Pascal uses the compiler directive $M to distribute the available memory. The default configuration is {$M 16384, 0, 655360}, where 16384 bytes is the stack size, 0 bytes is the minimum heap size, and 655360 is the maximum heap size.

It is recommended that you use the compiler directive $M to reduce the maximum heap reserved by Pascal to 0 bytes by entering the following:

```
{$M (16384, 0, 0)}
```

Reducing the maximum heap size to 0 bytes makes all far heap memory available to DOS (and therefore available to the driver) and allows your program to take maximum advantage of the **K_IntAlloc** and **K_DMAAlloc** functions. You can reserve some space for the internal memory manager or for DOS, if desired. Refer to your Borland Turbo Pascal (for DOS) documentation for more information.

Dynamically Allocating a Memory Buffer

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

If you are using a large memory buffer and programming in Borland Turbo Pascal for Windows, it is recommended that you use the Keithley Memory Manager before you begin programming. Refer to your board user's guide for more information about the Keithley Memory Manager.

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

It is recommended that you declare a dummy type array of ^Integer. The dimension of this array is irrelevant; it is used only to satisfy Pascal's type-checking requirements.

```
{$m (16384, 0, 0)}      { Turbo Pascal for DOS only }
. . .
Type
   IntArray = Array[0..1] of Integer;
. . .
Var
   AcqBuf : ^IntArray;   { Declare buffer of dummy type }
   hMem : Word;   { Declare word for memory handle, hMem }
. . .
wDasErr := K_DMAAlloc (hFrame, Samples, @AcqBuf, hMem);
wDasErr := K_SetDMABuf (hFrame, AcqBuf, Samples);
. . .
```

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

```
. . .
wDasErr := K_DMAFree (hMem);
. . .
```

Accessing Data from a Dynamically Allocated Memory Buffer

You access the data stored in a dynamically allocated buffer through Pascal pointer indirection. For example, assume that you want to display the first 10 samples in the buffer. The following code fragment illustrates how to access and display the data.

```
. . .
for i := 0 to 10 do begin
   writeln (’Sample #’, i,’ =’, AcqBuf^[i]);
End;
. . .
```

Dimensioning a Local Array

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

```
. . .
Data : Array[0..9999] of Integer;
. . .
wDasErr := K_SetBuf (hFrame, Data(0), 10000);
. . .
```

Creating a Channel-Gain Queue

The following example illustrates how to create a channel-gain queue called MyChanGainQueue for a DAS-1602 board by defining a Record as a new type. You must use **K_SetChnGAry** to assign the starting address of MyChanGainQueue to the frame defined by hFrame.

```
Type
   GainChanTable = Record
   num_of_codes : Integer;
   queue : Array[0..15] of Byte;
   end;
. . .
Const
   MyChanGainQueue : GainChanTable =
   num_of_codes : (8);     { Number of entries }
   queue :(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 }
         );
wDasErr := K_SetChnGAry (hFrame, MyChanGainQueue.num_of_codes);
```

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

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 **DAS1600_GetDevHandle** function.

```
wDasErr := DAS1600_GetDevHandle( 0, hDev );
if wDasErr <> 0 then
BEGIN
   FormatStr(HexErr, ' %4x ', wDasErr);
   writeln( 'Error', HexErr,'during DAS1600_GetDevHandle' );
   Halt(1);
END;
```

Programming in Borland Turbo Pascal (for DOS)

To program in Borland Turbo Pascal, you need the D1600TP7.TPU file. D1600TP7.TPU is a Turbo Pascal unit for Version 7.0 and is provided in the ASO-1600/1400/1200 software package.

Note: If you must create a new Turbo Pascal unit when compiling in Borland Turbo Pascal for versions higher than 7.0, refer to FILES.TXT for a list of the files to use.

To create an executable file in Borland Turbo Pascal, use the following compile and link statement:

```
TPC filename.pas
```

where *filename* indicates the name of your program.

Programming in Borland Turbo Pascal for Windows

To program in Borland Turbo Pascal for Windows, you need the following files; these files are provided in the ASO-1600/1400/1200 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DAS1600.DLL	Dynamic Link Library
DASDECL.INC	Include file
DAS1600.INC	Include file

To create an executable file in Borland Turbo Pascal for Windows, perform the following steps:

1. Load *filename*.pas into the Borland Turbo Pascal for Windows environment, where *filename* indicates the name of your program.
2. Create an executable file (.EXE).

Visual Basic for Windows Programming Information

The following sections contain information you need to allocate memory, to create a channel-gain queue, to convert integer data for digital I/O operations, and to handle errors when programming in Microsoft Visual Basic for Windows, as well as language-specific information for Microsoft Visual Basic for Windows.

Dynamically Allocating a Memory Buffer

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

If you are using a large memory buffer, it is recommended that you use the Keithley Memory Manager before you begin programming. Refer to your board user's guide for more information about the Keithley Memory Manager.

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

```
. . .
Global AcqBuf As Long   ' Declare pointer to buffer
Global hMem As Integer  ' Declare integer for memory handle
. . .
wDasErr = K_DMAAlloc (hFrame, Samples, AcqBuf, hMem)
wDasErr = K_SetDMABuf (hFrame, AcqBuf, Samples)
. . .
```

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

```
. . .
wDasErr = K_DMAFree (hMem)
. . .
```

Accessing Data from a Dynamically Allocated Memory Buffer with Fewer than 64 KB of Data

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

```
. . .
Dim Buffer(1000) As Integer   ' Declare local array
. . .
wDasErr = K_MoveBufToArray (Buffer(0), AcqBuf, 100)
. . .
```

Accessing Data from a Dynamically Allocated Memory Buffer with More than 64 KB of Data

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

When a memory buffer of more than 64 KB (32,768 values) is used, multiple selectors are required. Under Windows, **K_IntAlloc** uses a "tiled" method to allocate memory whereby a mathematical relationship does exist among the selectors. Specifically, if you allocate a buffer of more than 64 KB, each selector that is allocated has an arithmetic value that is eight greater than the previous one. The format of the address is a 32-bit value whose high word is the 16-bit selector value and low word is the 16-bit offset value. When the offset reaches 64 KB, the next consecutive memory address location can be accessed by adding eight to the selector and resetting the offset to zero; to do this, add &h80000 to the buffer starting address.

Table 3-1 illustrates the mapping of consecutive memory locations in protected-mode "tiled" memory, where *xxxxxxxx* indicates the address calculated by the CPU memory mapping mechanism.

Table 3-1. Protected-Mode Memory Architecture

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

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

```
Dim AcqBuf As Long
Dim NumSamps As Long

Dim Array(1000) As Integer

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

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

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

Accessing More than 64 KB of Data from a Dynamically Allocated Memory Buffer

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

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

'Same selector, add 32,768 bytes to offset: add &h8000
wDasErr = K_MoveBufToArray (Data(1,0), pBuf + &h8000, 16384)
'Add 8 to selector, offset = 0: add &h80000
wDasErr = K_MoveBufToArray (Data(2,0), pBuf + &h80000, 16384)
'Add 8 to selector, add 32,768 bytes to offset: add &h88000
wDasErr = K_MoveBufToArray (Data(3,0), pBuf + &h88000, 16384)
```

Dimensioning a Local Array

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

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

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 DAS-1600/1400/1200 Series Function Call Driver.

The following code fragment illustrates how to create a four-entry channel-gain queue called MyChanGainQueue for a DAS-1602 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** or **K_IntStart**), channel 0 is sampled at a gain of 1, channel 1 is sampled at a gain of 2, channel 2 is sampled at a gain of 4, and so on.

Converting Integer Data for Digital I/O Operations

You cannot dimension a byte-type array or allocate a byte-type buffer in Visual Basic for Windows. If you specify in your configuration file that you are using four or eight digital I/O lines, you must dimension an integer-type array or allocate an integer-type buffer instead.

For digital input operations, the driver stores two samples in each integer of the array or buffer. To convert the data to a usable format, whether you access the data directly using **K_SetBufI** (if the data is stored in a locally dimensioned array) or indirectly using **K_MoveBufToArray** and **K_SetBuf** (if the data is stored in a dynamically allocated buffer), you must unpack the data.

The following code fragment illustrates how to unpack 1,000 8-bit samples that have been stored in an array of half the size, with each element in the array holding two bytes of data.

```
Dim UnpackedData(1000) As Integer
Dim PackedData(500) As Integer

For n = 0 to 998 step 2
   UnpackedData(n) = PackedData(n/2) AND &HFF
   UnpackedData(n+1) = (PackedData(n/2) / 256) AND &HFF
Next n
```

For digital output operations, you must pack two samples into each integer in the program's local array. This ensures that when the driver accesses the data, either directly using **K_SetBufI** (if the driver is using a locally dimensioned array) or indirectly using **K_MoveArrayToBuf** and **K_SetBuf** (if the driver is using a dynamically allocated buffer), the samples will be in consecutive memory locations as the driver expects.

The following code fragment illustrates how to pack 1,000 8-bit samples into an array of half the size, with each element in the packed array holding two bytes of data.

```
Dim UnpackedData(1000) As Integer
Dim PackedData(500) As Integer

For n = 0 to 499
   PackedData(n) = UnpackedData(n*2)+256*UnpackedData(n*2+1)
Next n
```

Handling Errors

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

```
. . .
wDASErr = K_GetDevHandle (hDrv, BoardNum, hDev)
If (wDASErr <> 0) Then
   MsgBox "K_GetDevHandle Error: " + Hex$ (wDASErr),
      MB_ICONSTOP, "DAS-1600 SERIES ERROR"
   End
End If
. . .
```

Programming in Microsoft Visual Basic for Windows

The files you need to program in Microsoft Visual Basic for Windows depend on whether you are writing a 16-bit or 32-bit program. The following files are provided either in the ASO-1600/1400/1200 software package or on the ASO-Win95/32-Bit disk, which is shipped with the ASO-1600/1400/1200 software package.

Program	File	Description
16 bits	DASSHELL.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DAS1600.DLL	Dynamic Link Library of board-specific functions
	DASDECL.BAS	Include file of Shell function definitions
	DAS1600.BAS	Include file of board-specific function definitions
32 bits	DASSHL32.DLL	Dynamic Link Library of Shell functions
	DASSUPRT.DLL	Dynamic Link Library of support functions
	DASSHL16.DLL	Dynamic Link Library of support functions
	DASDEC32.BAS	Include file of Shell function definitions

To create an executable file in Visual Basic for Windows, perform the following steps. Refer to the documentation supplied with your compiler for complete information.

1. Start Visual Basic for Windows.
2. Add the necessary include files to the project: DASDECL.BAS (or DASDEC32.BAS) and DAS1600.BAS (16-bit programs only).
3. Create an executable file (.EXE).

BASIC Programming Information

The following sections contain information you need to reserve memory, to create a channel-gain queue, to convert integer data for digital I/O operations, and to handle errors when programming in BASIC, as well as language-specific information for Microsoft QuickBasic, Microsoft Professional Basic, and Microsoft Visual Basic for DOS.

Reducing the Memory Heap

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

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

Dynamically Allocating a Memory Buffer

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

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

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

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

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

Accessing Data from a Dynamically Allocated Memory Buffer with Fewer than 64 KB of Data

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

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

Accessing Data from a Dynamically Allocated Memory Buffer with More than 64 KB of Data

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

When a memory buffer of more than 64 KB (32,768 values) is used, multiple segments are required. When an offset reaches 64 KB, the next linear memory address location can be accessed by adding &h1000 to the buffer segment and resetting the offset to zero.

Table 3-2 illustrates the mapping of consecutive memory locations at a segment page boundary.

Table 3-2. Real-Mode Memory Architecture

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

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

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

Dim Array(1000) As Integer

. . .                ’Initialize array with desired values

NumSamps = 50000
wDasErr = KIntAlloc% (hFrame, NumSamps, AcqBuf, hMem)
DesiredSamp = 40000
DesiredByte = DesiredSamp * 2       ’Number of bytes into buffer

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

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

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

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

Accessing More than 64 KB of Data from a Dynamically Allocated Memory Buffer

To move more than 64 KB of data (32,767 values) from the memory buffer to the program's local array, the program must call **KMoveBufToArray** more than once. For example, assume that pBuf is a pointer to a dynamically allocated buffer that contains 65,536 values. The following code fragment illustrates how to move 65,536 values from the memory buffer to a local array (Data) in the program.

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

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

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

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

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

Dimensioning a Local Array

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

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

Creating a Channel-Gain Queue

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

Next, you must fill the array with the channel-gain information. After you create the channel-gain queue, you must use **KFormatChnGAry** to reformat the channel-gain queue so that it can be used by the DAS-1600/1400/1200 Series Function Call Driver.

The following code fragment illustrates how to create a four-entry channel-gain queue called MyChanGainQueue for a DAS-1602 board and how to use **KSetChnGAry** to assign the starting address of MyChanGainQueue to the frame defined by hFrame.

```
. . .
Dim 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 = KFormatChnGAry% (MyChanGainQueue(0))
wDasErr = KSetChnGAry% (hFrame, MyChanGainQueue(0))
. . .
```

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

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

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

Converting Integer Data for Digital I/O Operations

You cannot dimension a byte-type array or allocate a byte-type buffer in BASIC. If you specify in your configuration file that you are using four or eight digital I/O lines, you must dimension an integer-type array or allocate an integer-type buffer instead.

For digital input operations, the driver stores two samples in each integer of the array or buffer. To convert the data to a usable format, whether you access the data directly using **KSetBufI** (if the data is stored in a locally dimensioned array) or indirectly using **KMoveBufToArray** and **KSetBuf** (if the data is stored in a dynamically allocated buffer), you must unpack the data.

The following code fragment illustrates how to unpack 1,000 8-bit samples that have been stored in an array of half the size, with each element in the array holding two bytes of data.

```
Dim UnpackedData(1000) As Integer
Dim PackedData(500) As Integer

For n = 0 to 998 step 2
   UnpackedData(n) = PackedData(n/2) AND &HFF
   UnpackedData(n+1) = (PackedData(n/2) / 256) AND &HFF
Next n
```

For digital output operations, you must pack two samples into each integer in the program's local array. This ensures that when the driver accesses the data, either directly using **KSetBufI** (if the driver is using a locally dimensioned array) or indirectly using **KMoveArrayToBuf** and **KSetBuf** (if the driver is using a dynamically allocated buffer), the samples will be in consecutive memory locations as the driver expects.

The following code fragment illustrates how to pack 1,000 8-bit samples into an array of half the size, with each element in the packed array holding two bytes of data.

```
Dim UnpackedData(1000) As Integer
Dim PackedData(500) As Integer

For n = 0 to 499
   PackedData(n) = UnpackedData(n*2)+256*UnpackedData(n*2+1)
Next n
```

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 **DAS1600GetDevHandle** function.

```
. . .
wDASErr = DAS1600GETDEVHANDLE% (BoardNum, hDev)
IF (wDASErr <> 0) THEN
BEEP
PRINT "Error";HEX$(wDASErr);"occurred during'DAS1600GETDEVHANDLE%'"
 END
END IF
. . .
```

Programming in Microsoft QuickBasic

To program in Microsoft QuickBasic, you need the following files; these files are provided in the DAS-1600/1400/1200 Series standard software package.

File	Description
D1600Q45.LIB	Linkable driver for QuickBasic, Version 4.5, stand-alone, executable (.EXE) programs
D1600Q45.QLB	Command-line loadable driver for the QuickBasic, Version 4.5, integrated environment
QB4DECL.BI	Include file
DASDECL.BI	Include file
DAS1600.BI	Include file

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

1. Enter the following to invoke the environment:

   ```
   QB /L D1600Q45 filename.bas
   ```

   where *filename* indicates the name of your program.

2. Create an executable file (.EXE).

Programming in Microsoft Professional Basic

To program in Microsoft Professional Basic, you need the following files; these files are provided in the DAS-1600/1400/1200 Series standard software package.

File	Description
D1600QBX.LIB	Linkable driver for Professional Basic stand-alone, executable (.EXE) programs
D1600QBX.QLB	Command-line loadable driver for the Professional Basic integrated environment
DASDECL.BI	Include file
DAS1600.BI	Include file

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

1. Enter the following to invoke the environment:

   ```
   QBX /L D1600QBX filename.bas
   ```

   where *filename* indicates the name of your program.

2. Create an executable file (.EXE).

Programming in Microsoft Visual Basic for DOS

To program in Microsoft Visual Basic for DOS, you need the following files; these files are provided in the DAS-1600/1400/1200 Series standard software package.

File	Description
D1600VBD.LIB	Linkable driver for Visual Basic for DOS stand-alone, executable (.EXE) programs
D1600VBD.QLB	Command-line loadable driver for the Visual Basic for DOS integrated environment
DASDECL.BI	Include file
DAS1600.BI	Include file

To create an executable file in Microsoft Visual Basic for DOS, perform the following steps:

1. Invoke the Visual Basic for DOS environment by entering the following:

   `VBDOS /L D1600VBD.QLB` *`filename`*`.BAS`

   where *filename* indicates the name of your program.

2. Create an executable file (.EXE).

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
- Conversion mode functions
- Channel and gain functions
- Clock functions
- Trigger functions
- 82C54 counter/timer 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	DAS1600_DevOpen	page 4-27
	K_OpenDriver	page 4-122
	K_CloseDriver	page 4-37
	DAS1600_GetDevHandle	page 4-30
	K_GetDevHandle	page 4-83
	K_FreeDevHandle	page 4-69
	K_DASDevInit	page 4-43
Operation	K_ADRead	page 4-32
	K_DAWrite	page 4-45
	K_DIRead	page 4-48
	K_DOWrite	page 4-64
	K_DMAStart	page 4-56
	K_DMAStatus	page 4-58
	K_DMAStop	page 4-61
	K_IntStart	page 4-102
	K_IntStatus	page 4-104
	K_IntStop	page 4-107
	K_SyncStart	page 4-173
Frame Management	K_GetADFrame	page 4-75
	K_GetDAFrame	page 4-81
	K_GetDIFrame	page 4-85
	K_GetDOFrame	page 4-87
	K_FreeFrame	page 4-71
	K_ClearFrame	page 4-35

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Memory Management	K_DMAAlloc	page 4-51
	K_DMAFree	page 4-54
	K_IntAlloc	page 4-97
	K_IntFree	page 4-100
	KMakeDMABuf	page 4-110
	K_MoveArrayToBuf	page 4-112
	K_MoveArrayToBufL	page 4-114
	K_MoveBufToArray	page 4-116
	K_MoveBufToArrayL	page 4-118
	K_MoveDataBuf	page 4-120
Buffer Address	K_SetBuf	page 4-132
	K_SetBufI	page 4-135
	K_SetBufL	page 4-137
	K_SetDMABuf	page 4-156
Buffering Mode	K_SetContRun	page 4-151
	K_ClrContRun	page 4-41
Conversion Mode	K_SetADFreeRun	page 4-127
	K_ClrADFreeRun	page 4-39
	K_SetSSH	page 4-162

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Channel and Gain	K_SetChn	page 4-141
	K_SetStartStopChn	page 4-164
	K_SetG	page 4-159
	K_SetStartStopG	page 4-166
	K_SetChnGAry	page 4-143
	K_FormatChnGAry	page 4-67
	K_RestoreChnGAry	page 4-125
	K_GetADConfig	page 4-73
	K_GetADMode	page 4-77
Clock	K_SetClk	page 4-146
	K_SetClkRate	page 4-148
	K_GetClkRate	page 4-79
	K_SetBurstTicks	page 4-139
Trigger	K_SetTrig	page 4-169
	K_SetADTrig	page 4-129
	K_SetTrigHyst	page 4-171
	K_SetDITrig	page 4-153
82C54 Counter/Timer[1]	DAS1600_8254Control	page 4-7
	DAS1600_8254SetCounter	page 4-21
	DAS1600_8254SetClk0	page 4-19
	DAS1600_8254SetTrig0	page 4-24
	DAS1600_8254GetCounter	page 4-13
	DAS1600_8254GetClk0	page 4-10
	DAS1600_8254GetTrig0	page 4-16

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Miscellaneous	K_GetErrMsg	page 4-89
	K_GetVer	page 4-94
	K_GetShellVer	page 4-91

Notes

[1] These functions allow you to program the 82C54 counter/timer on the DAS-1600/1400/1200 Series board. See Appendix E of your user's guide for more information.

Keep the following conventions in mind throughout this chapter:

- Under "Boards Supported," *All* refers to the following boards: DAS-1601, DAS-1602, DAS-1401, DAS-1402, DAS-1201, DAS-1202.
- Although the function names are shown with underscores, do not use the underscores in the BASIC languages.
- The data types DWORD, WORD, and BYTE are defined in the language-specific include files.
- Variable names are shown in italics.
- The return value for all DAS-1600/1400/1200 Series 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 prototype. The name of each variable in both the prototype and usage sections includes a prefix that indicates the associated data type. These prefixes are described in Table 4-2.

Table 4-2. Data Type Prefixes

Prefix	Data Type	Comments
sz	Pointer to string terminated by zero	This data type is typically used for variables that specify the driver's configuration file name.
h	Handle to device, frame, and memory block	This data type is used for handle-type variables. You declare handle-type variables in your program as long or DWORD, depending on the language you are using. The actual variable is passed to the driver by value.
ph	Pointer to a handle-type variable	This data type is used when calling the FCD functions to get a driver handle, a device 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.

Boards Supported

All

Purpose

Writes data to the 82C54 counter/timer control register of the specified board.

Prototype

C/C++

DASErr far pascal DAS1600_8254Control (WORD *nBrdNum*, WORD *nCtrlData*);

Turbo Pascal

Function DAS1600_8254Control (*nBrdNum* : Word; *nCtrlData* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254Control (*nBrdNum* : Word; *nCtrlData* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254Control Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, ByVal *nCtrlData* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254CONTROL% ALIAS "DAS1600_8254Control" (BYVAL *nBrdNum* AS INTEGER, BYVAL n*CtrlData* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
nCtrlData	Data value that is written to the 82C54 control register. Only the low byte is used.

Return Value

Error/status code. Refer to Appendix A.

DAS1600_8254Control (cont.)

Remarks

This function sets the 82C54 counter/timer control register for the board defined by *nBrdNum* to the value of *nCtrlData*. If the counter/timer specified in the control word is currently used by an operation, an error is returned.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254SetCounter, DAS1600_8254GetCounter, DAS1600_8254SetClk0, DAS1600_8254GetClk0, DAS1600_8254SetTrig0, DAS1600_8254GetTrig0

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
WORD nCtrlData;
...
wDasErr = DAS1600_8254Control (0, nCtrlData);
```

Turbo Pascal

```
uses D1600TP7;
...
nCtrlData : Word;
...
wDasErr := DAS1600_8254Control (0, nCtrlData);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
nCtrlData : Word;
...
wDasErr := DAS1600_8254Control (0, nCtrlData);
```

Visual Basic for Windows
(Add DAS1600.BAS to your project)

```
...
Global nCtrlData As Integer
...
wDasErr = DAS1600_8254Control (0, nCtrlData)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM nCtrlData AS INTEGER
...
wDasErr = DAS16008254Control% (0, nCtrlData)
```

DAS1600_8254GetClk0

Boards Supported

All

Purpose

Gets the clock source for counter 0 of the 82C54 counter/timer.

Prototype

C/C++

DASErr far pascal DAS1600_8254GetClk0 (WORD *nBrdNum*, WORD far **pClkSrc*);

Turbo Pascal

Function DAS1600_8254GetClk0 (*nBrdNum* : Word; Var *pClkSrc* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254GetClk0 (*nBrdNum* : Word; Var *pClkSrc* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254GetClk0 Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, *pClkSrc* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254GETCLK0% ALIAS "DAS1600_8254GetClk0" (BYVAL *nBrdNum* AS INTEGER, SEG *pClkSrc* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
pClkSrc	Counter 0 clock source. Value stored: **0** for Internal **1** for External

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board defined by *nBrdNum*, this function stores the counter 0 clock source in *pClkSrc*.

The internal clock source is the onboard clock; an external clock source is an external signal connected to the CTR 0 CLOCK IN pin (21) of the main I/O connector.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254SetClk0

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
WORD nClkSrc;
...
wDasErr = DAS1600_8254GetClk0 (0, &nClkSrc);
```

Turbo Pascal

```
uses D1600TP7;
...
nClkSrc : Word;
...
wDasErr := DAS1600_8254GetClk0 (0, nClkSrc);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
nClkSrc : Word;
...
wDasErr := DAS1600_8254GetClk0 (0, nClkSrc);
```

Visual Basic for Windows

(Add DAS1600.BAS to your project)

```
...
Global nClkSrc As Integer
...
wDasErr = DAS1600_8254GetClk0 (0, nClkSrc)
```

DAS1600_8254GetClk0 (cont.)

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM nClkSrc AS INTEGER
...
wDasErr = DAS16008254GetClk0% (0, nClkSrc)
```

Boards Supported

All

Purpose

Gets the current value of the specified counter and writes it to the location pointed to by *pCntrData*.

Prototype

C/C++

DASErr far pascal DAS1600_8254GetCounter (WORD *nBrdNum*, WORD *nCntr*, word far **pCntrData*);

Turbo Pascal

Function DAS1600_8254GetCounter (*nBrdNum* : Word; *nCntr* : Word; Var *pCntrData* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254GetCounter (*nBrdNum* : Word; *nCntr* : Word; Var *pCntrData* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254GetCounter Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, ByVal *nCntr* As Integer, *pCntrData* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254GETCOUNTER% ALIAS "DAS1600_8254GetCounter" (BYVAL *nBrdNum* AS INTEGER, BYVAL *nCntr* AS INTEGER, SEG *pCntrData* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
nCntr	Counter that you want read. Valid values: **0** to **2**
pCntrData	Location where the counter value is written.

Return Value

Error/status code. Refer to Appendix A.

DAS1600_8254GetCounter (cont.)

Remarks

For the board defined by *nBrdNum*, this function gets the current value of the counter specified by *nCntr* and writes the value to the location specified by *pCntrData*.

You must use the **DAS1600_8254Control** function to set up the 82C54 before you use this function. If the counter/timer is running when you use this function, the count value may not be valid because the counter may be changing the value during the read. Use the 82C54 counter-latch function or the 82C54 readback function to obtain a valid count. The 82C54 returns one byte each time you use this function (in the low byte of *pCntrData*); therefore, you must call this function twice.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254Control, DAS1600_8254SetCounter

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
WORD nCntrData;
...
wDasErr = DAS1600_8254GetCounter (0, 0, &nCntrData);
```

Turbo Pascal

```
uses D1600TP7;
...
nCntrData : Word;
...
wDasErr := DAS1600_8254GetCounter (0, 0, nCntrData);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
nCntrData : Word;
...
wDasErr := DAS1600_8254GetCounter (0, 0, nCntrData);
```

Visual Basic for Windows
(Add DAS1600.BAS to your project)

```
...
Global nCntrData As Integer
...
wDasErr = DAS1600_8254GetCounter (0, 0, nCntrData)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM nCntrData AS INTEGER
...
wDasErr = DAS16008254GetCounter% (0, 0, nCntrData)
```

DAS1600_8254GetTrig0

Boards Supported

All

Purpose

Indicates whether the gate signal is enabled or disabled.

Prototype

C/C++

DASErr far pascal DAS1600_8254GetTrig0 (WORD *nBrdNum*, WORD far **pTrigEnabled*);

Turbo Pascal

Function DAS1600_8254GetTrig0 (*nBrdNum* : Word; Var *pTrigEnabled* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254GetTrig0 (*nBrdNum* : Word; Var *pTrigEnabled* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254GetTrig0 Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, *pTrigEnabled* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254GETTRIG0% ALIAS "DAS1600_8254GetTrig0" (BYVAL *nBrdNum* AS INTEGER, SEG *pTrigEnabled* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
pTrigEnabled	Indicates whether the gate signal is enabled or disabled. Value stored: **0** for Disabled **1** for Enabled

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board defined by *nBrdNum*, this function indicates whether the gate signal is enabled or disabled in *pTrigEnabled*.

The gate signal is the signal at the IP0/TRIG0/XPCLK pin (25) of the main I/O connector. The gate signal determines when counters 1 and 2 of the 82C54 counter/timer are used. If the gate signal is disabled (*pTrigEnabled* = 0), counters 1 and 2 are always used (counters 1 and 2 continually count down). If the gate signal is enabled (*pTrigEnabled* = 1), counters 1 and 2 are used only when the signal at the IP0/TRIG0/XPCLK pin (25) is low; whenever the signal at the IP0/TRIG0/XPCLK pin (25) goes high, counters 1 and 2 stop counting down.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254SetTrig0

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
WORD pTrigEnabled;
...
wDasErr = DAS1600_8254GetTrig0 (0, &pTrigEnabled);
```

Turbo Pascal

```
uses D1600TP7;
...
pTrigEnabled : Word;
...
wDasErr := DAS1600_8254GetTrig0 (0, pTrigEnabled);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
pTrigEnabled : Word;
...
wDasErr := DAS1600_8254GetTrig0 (0, pTrigEnabled);
```

Visual Basic for Windows
(Add DAS1600.BAS to your project)

```
...
Global pTrigEnabled As Integer
...
wDasErr = DAS1600_8254GetClk0 (0, pTrigEnabled)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM pTrigEnabled AS INTEGER
...
wDasErr = DAS16008254GetTrig0% (0, pTrigEnabled)
```

Boards Supported

All

Purpose

Specifies the clock source for counter 0 of the 82C54 counter/timer.

Prototype

C/C++

DASErr far pascal DAS1600_8254SetClk0 (WORD *nBrdNum*, WORD *nClkSrc*);

Turbo Pascal

Function DAS1600_8254SetClk0 (*nBrdNum* : Word; *nClkSrc* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254SetClk0 (*nBrdNum* : Word; *nClkSrc* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254SetClk0 Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, ByVal *nClkSrc* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254SETCLK0% ALIAS "DAS1600_8254SetClk0" (BYVAL *nBrdNum* AS INTEGER, BYVAL *nClkSrc* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
nClkSrc	Counter 0 clock source. Valid values: **0** for Internal **1** for External

Return Value

Error/status code. Refer to Appendix A.

DAS1600_8254SetClk0 (cont.)

Remarks

For the board defined by *nBrdNum*, this function specifies the counter 0 clock source in *nClkSrc*.

The internal clock source is the onboard clock; an external clock source is an external signal connected to the CTR 0 CLOCK IN pin (21) of the main I/O connector.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254GetClk0

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
wDasErr = DAS1600_8254SetClk0 (0, 1);
```

Turbo Pascal

```
uses D1600TP7;   (* Use D1800TP6 for TP ver 6.0 *)
...
wDasErr := DAS1600_8254SetClk0 (0, 1);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
wDasErr := DAS1600_8254SetClk0 (0, 1);
```

Visual Basic for Windows

(Add DAS1600.BAS to your project)

```
...
wDasErr = DAS1600_8254SetClk0 (0, 1)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
wDasErr = DAS16008254SetClk0% (0, 1)
```

Boards Supported

All

Purpose

Sets the specified counter to the value of *nCntrData*.

Prototype

C/C++

DASErr far pascal DAS1600_8254SetCounter (WORD *nBrdNum*, WORD *nCntr*, WORD *nCntrData*);

Turbo Pascal

Function DAS1600_8254SetCounter (*nBrdNum* : Word; *nCntr* : Word; *nCntrData* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254SetCounter (*nBrdNum* : Word; *nCntr* : Word; *nCntrData* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254SetCounter Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, ByVal *nCntr* As Integer, ByVal *nCntrData* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254SETCOUNTER% ALIAS "DAS1600_8254SetCounter" (BYVAL *nBrdNum* AS INTEGER, BYVAL *nCntr* AS INTEGER, BYVAL *nCntrData* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
nCntr	Counter that you want set. Valid values: **0** to **2**
nCntrData	Value that you want to set the counter to.

Return Value

Error/status code. Refer to Appendix A.

DAS1600_8254SetCounter (cont.)

Remarks

For the board defined by *nBrdNum*, this function sets the counter specified by *nCntr* to the value of *nCntrData* (least significant byte).

You must use the **DAS1600_8254Control** function before the **DAS1600_8254SetCounter** function to set up the data transfer, which can be performed as follows:

- Least significant byte (LSB) only
- Most significant byte (MSB) only
- Least significant byte (LSB) followed by most significant byte (MSB)

The 82C54 counter/timer accepts eight bits of data each time you use this function; therefore, you must call the function twice to program the 16 bits of each counter.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254Control, DAS1600_8254GetCounter

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
WORD nCntrData;
...
wDasErr = DAS1600_8254SetCounter (0, 0, nCntrData);
```

Turbo Pascal

```
uses D1600TP7;
...
nCntrData : Word;
...
wDasErr := DAS1600_8254SetCounter (0, 0, nCntrData);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
nCntrData : Word;
...
wDasErr := DAS1600_8254SetCounter (0, 0, nCntrData);
```

Visual Basic for Windows
(Add DAS1600.BAS to your project)

```
...
Global nCntrData As Integer
...
wDasErr = DAS1600_8254SetCounter (0, 0, nCntrData)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM nCntrData AS INTEGER
...
wDasErr = DAS16008254SetCounter% (0, 0, nCntrData)
```

DAS1600_8254SetTrig0

Boards Supported

All

Purpose

Enables and disables the signal at the IPO/TRIG0/XPCLK pin (25) of the main I/O connector to act as a hardware gate signal.

Prototype

C/C++

DASErr far pascal DAS1600_8254SetTrig0 (WORD *nBrdNum*, WORD *nTrigEnable*);

Turbo Pascal

Function DAS1600_8254SetTrig0 (*nBrdNum* : Word; *nTrigEnable* : Word) : Word;

Turbo Pascal for Windows

Function DAS1600_8254SetTrig0 (*nBrdNum* : Word; *nTrigEnable* : Word) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_8254SetTrig0 Lib "DAS1600.DLL" (ByVal *nBrdNum* As Integer, ByVal *nTrigEnable* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS16008254SETTRIG0% ALIAS "DAS1600_8254SetTrig0" (BYVAL *nBrdNum* AS INTEGER, ByVal *nTrigEnable* AS INTEGER)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
nTrigEnable	Specifies whether the gate signal is enabled or disabled. Valid values: **0** for Disabled **1** for Enabled

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board defined by *nBrdNum*, this function specifies whether the gate signal is enabled or disabled in *nTrigEnable*.

The gate signal is the signal at the IP0/TRIG0/XPCLK pin (25) of the main I/O connector. The gate signal determines when counters 1 and 2 of the 82C54 counter/timer are used. If you disable the gate signal (*nTrigEnable* = 0), counters 1 and 2 are always used (counters 1 and 2 continually count down). If you enable the gate signal (*nTrigEnable* = 1), counters 1 and 2 are used only when the signal at the IP0/TRIG0/XPCLK pin (25) is low; whenever the signal at the IP0/TRIG0/XPCLK pin (25) goes high, counters 1 and 2 stop counting down.

Refer to Appendix E of your board user's guide or to the manufacturer's data sheet for information about programming the 82C54 counter/timer.

You cannot use this function with Windows 95, 32-bit programs.

See Also

DAS1600_8254Control, DAS1600_8254SetTrig0

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
wDasErr = DAS1600_8254SetTrig0 (0, 1);
```

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := DAS1600_8254SetTrig0 (0, 1);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
wDasErr := DAS1600_8254SetTrig0 (0, 1);
```

Visual Basic for Windows

(Add DAS1600.BAS to your project)

```
...
wDasErr = DAS1600_8254SetTrig0 (0, 1)
```

DAS1600_8254SetTrig0 (cont.)

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
wDasErr = DAS16008254SetTrig0% (0, 1)
```

Boards Supported

All

Purpose

Initializes the DAS-1600/1400/1200 Series Function Call Driver.

Prototype

C/C++

DASErr far pascal DAS1600_DevOpen (char far **szCfgFile*,
char far **pBoards*);

Turbo Pascal

Function DAS1600_DevOpen (Var *szCfgFile* : char;
Var *pBoards* : Integer) : Word;

Turbo Pascal for Windows

Function DAS1600_DevOpen (Var *szCfgFile* : char;
Var *pBoards* : Integer) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_DevOpen Lib "DAS1600.DLL"
(ByVal *szCfgFile* As String, *pBoards* As Integer) As Integer

BASIC

DECLARE FUNCTION DAS1600DEVOPEN% ALIAS
"DAS1600_DevOpen" (BYVAL *szCfgFile* AS STRING,
SEG *pBoards* AS INTEGER)

Parameters

szCfgFile	Driver configuration file. Valid values: The name of a configuration file.
pBoards	Number of boards defined in *szCfgFile*. Value stored: **1** or **2**

Return Value

Error/status code. Refer to Appendix A.

DAS1600_DevOpen (cont.)

Remarks

This function initializes the driver according to the information in the configuration file specified by *szCfgFile* and stores the number of boards defined in *szCfgFile* in *pBoards*.

You create a configuration file using the CFG1600.EXE utility. Refer to your board user's guide for more information.

You cannot use this function with Windows 95, 32-bit programs.

See Also

K_OpenDriver

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
char nBoards;
...
wDasErr = DAS1600_DevOpen ("DAS1600.CFG", &nBoards);
```

Turbo Pascal

```
uses D1600TP7;
...
szCfgName : String;
nBoards : Integer;
...
szCfgName := 'DAS1600.CFG' + #0;
wDasErr := DAS1600_DevOpen (szCfgName[1], nBoards);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
szCfgName : String;
nBoards : Integer;
...
szCfgName := 'DAS1600.CFG' + #0;
wDasErr := DAS1600_DevOpen (szCfgName[1], nBoards);
```

Visual Basic for Windows
(Add DAS1600.BAS to your project)

```
...
DIM nBoards AS INTEGER
DIM szCfgName AS STRING
...
szCfgName = "DAS1600.CFG" + CHR$(0)
wDasErr = DAS1600_DevOpen(szCfgName, nBoards)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM nBoards AS INTEGER
DIM szCfgName AS STRING
...
szCfgName = "DAS1600.CFG" + CHR$(0)
wDasErr = DAS1600DEVOPEN%(SSEGADD(szCfgName), nBoards)
```

DAS1600_GetDevHandle

Boards Supported

All

Purpose

Initializes a DAS-1600/1400/1200 Series board.

Prototype

C/C++

DASErr far pascal DAS1600_GetDevHandle (short *nBrdNum*,
void far *far **phDev*);

Turbo Pascal

Function DAS1600_GetDevHandle (*nBrdNum* : Integer;
Var *phDev* : Longint) : Word;

Turbo Pascal for Windows

Function DAS1600_GetDevHandle (*nBrdNum* : Integer;
Var *phDev* : Longint) : Word; far; external 'DAS1600';

Visual Basic for Windows

Declare Function DAS1600_GetDevHandle Lib "DAS1600.DLL"
(ByVal *nBrdNum* As Integer, *phDev* As Long) As Integer

BASIC

DECLARE FUNCTION DAS1600GETDEVHANDLE% ALIAS
"DAS1600_GetDevHandle" (BYVAL *nBrdNum* AS INTEGER,
SEG *phDev* AS LONG)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
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 device handle of the specified board in *phDev*.

You cannot use this function with Windows 95, 32-bit programs.

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

See Also K_GetDevHandle

Usage

C/C++

```
#include "DAS1600.H"   // Use DAS1600.HPP for C++
...
void far *hDev;
...
wDasErr = DAS1600_GetDevHandle (0, &hDev);
```

Turbo Pascal

```
uses D1600TP7;
...
hDev : Longint;   { Device Handle }
...
wDasErr := DAS1600_GetDevHandle (0, hDev);
```

Turbo Pascal for Windows

```
{$I DAS1600.INC}
...
hDev : Longint;   { Device Handle }
...
wDasErr := DAS1600_GetDevHandle (0, hDev);
```

Visual Basic for Windows

(Add DAS1600.BAS to your project)

```
...
Global hDev As Long   ' Device Handle
...
wDasErr = DAS1600_GetDevHandle (0, hDev)
```

BASIC

```
' $INCLUDE: 'DAS1600.BI'
...
DIM hDev AS LONG   ' Device Handle
wDasErr = DAS1600GetDevHandle%(0, hDev)
```

K_ADRead

Boards Supported

All

Purpose

Reads a single analog input value.

Prototype

C/C++

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

Turbo Pascal

Function K_ADRead (*hDev* : Longint; *nChan* : Byte; *nGain* : Byte; *pData* : Pointer) : Word;

Turbo Pascal for Windows

Function K_ADRead (*hDev* : Longint; *nChan* : Byte; *nGain* : Byte; *pData* : Pointer) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the board.
nChan	Analog input channel. Valid values: **0** to **255**

nGain Gain code.
Valid values:

Board	Gain	Gain Code
DAS-1601 DAS-1401	1	0
	10	1
	100	2
	500	3
DAS-1602 DAS-1402	1	0
	2	1
	4	2
	8	3

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 value in *pData*.

The data stored in pData is a count value. Refer to Appendix B for information on converting the count to voltage.

For DAS-1200 Series boards, the value of *nGain* is ignored.

See Also K_DMAStart, K_IntStart, K_SyncStart

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wADValue : Integer;
...
wDasErr := K_ADRead (hDev, 0, 0, @wADValue);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wADValue : Integer;
...
wDasErr := K_ADRead (hDev, 0, 0, @wADValue);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Sets the elements of a frame to their default values.

Prototype

C/C++

DASErr far pascal K_ClearFrame (DWORD *hFrame*);

Turbo Pascal

Function K_ClearFrame (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_ClearFrame (*hFrame* : Longint) : Word; far;
external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

The following table lists the frame types and where to look for their default values.

A/D	Table 2-1 on page 2-8
D/A	Table 2-5 on page 2-32
DI	Table 2-6 on page 2-42
DO	Table 2-7 on page 2-43

See Also K_GetADFrame, K_GetDAFrame, K_GetDIFrame, K_GetDOFrame

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_ClearFrame (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_ClearFrame (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_ClearFrame (hAD)
```

BASIC

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

Boards Supported

All

Purpose

Closes a previously initialized Keithley DAS Function Call Driver.

Prototype

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

Turbo Pascal
Not supported

Turbo Pascal for Windows
Function K_CloseDriver (*hDrv* : Longint) : Word; far;
external 'DASSHELL';

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

BASIC
Not supported

Parameters

hDrv	Driver handle you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

See Also

K_FreeDevHandle

K_CloseDriver (cont.)

Usage

C/C++

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

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_CloseDriver (hDrv);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

K_ClrADFreeRun

Boards Supported

All

Purpose

Specifies paced conversion mode for an analog input operation.

Prototype

C/C++

DASErr far pascal K_ClrADFreeRun (DWORD *hFrame*);

Turbo Pascal

Function K_ClrADFreeRun (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_ClrADFreeRun (*hFrame* : Longint) : Word; far;
external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KCLRADFREERUN% ALIAS
"K_ClrADFreeRun" (BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function sets the conversion mode for the operation defined by *hFrame* to paced mode and sets the Conversion Mode element in the frame accordingly.

K_GetADFrame and **K_ClearFrame** also enable paced conversion mode.

See Also

K_ClearFrame, K_GetADFrame, K_SetADFreeRun

K_ClrADFreeRun (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_ClrADFreeRun (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_ClrADFreeRun (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_ClrADFreeRun (hAD)
```

BASIC

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

Boards Supported

All

Purpose

Specifies single-cycle buffering mode.

Prototype

C/C++

DASErr far pascal K_ClrContRun (DWORD *hFrame*);

Turbo Pascal

Function K_ClrContRun (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_ClrContRun (*hFrame* : Longint) : Word; far;
external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

K_GetADFrame, **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 the following pages:

Analog input operations	page 2-24
Analog output operations	page 2-38
Digital I/O operations	page 2-51

See Also K_SetContRun

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_ClrContRun (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_ClrContRun (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_ClrContRun (hAD)
```

BASIC

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

Boards Supported

All

Purpose

Reinitializes a board.

Prototype

C/C++

DASErr far pascal K_DASDevInit (DWORD *hDev*);

Turbo Pascal

Function K_DASDevInit (*hDev* : Longint) : Longint;

Turbo Pascal for Windows

Function K_DASDevInit (*hDev* : Longint) : Longint; far;
external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stops all operations currently in progress and sets the board specified by *hDev* back to its power-up state.

K_DASDevInit (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_DASDevInit (hDev);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_DASDevInit (hDev);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

DAS-1601, DAS-1602

Purpose

Writes a single analog output value.

Prototype

C/C++

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

Turbo Pascal

Function K_DAWrite (*hDev* : Longint; *nChan* : Byte; *dwData* : Longint) : Word;

Turbo Pascal for Windows

Function K_DAWrite (*hDev* : Longint; *nChan* : Byte; *dwData* : Longint) : Word; far; external 'DASSHELL';

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

BASIC

DECLARE FUNCTION KDAWRITE% ALIAS "K_DAWrite" (BYVAL *hDev* AS LONG, BYVAL *nChan* AS INTEGER, BYVAL *dwData* AS LONG)

Parameters

hDev	Handle associated with the board.
nChan	Analog output channel. Valid values: **0** = Channel 0 **1** = Channel 1 **2** = Both channels
dwData	Analog output value. Valid values: **0** to **4095**

K_DAWrite (cont.)

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.*

dwData is a 32-bit variable, but the output value must contain only 12 bits.

The data stored in dwData is a count value. For information on converting a voltage value to a count, refer to Appendix B.

Refer to page 2-30 for more information on analog output operations.

See Also

K_IntStart, K_SyncStart

Usage

C/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);
```

Turbo Pascal

```
uses D1600TP7;
...
dwDAValue : Longint;
...
dwDAValue := Round((5.0 * 4096.0 / 20.0) + 2048) shl 4;
wDasErr := K_DAWrite (hDev, 0, dwDAValue);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
dwDAValue : Longint;
...
dwDAValue := Round((5.0 * 4096.0 / 20.0) + 2048) shl 4;
wDasErr := K_DAWrite (hDev, 0, dwDAValue);
```

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwDAValue AS LONG
...
dwDAValue = (INT (5.0 * 4096! / 20!) + 2048) * 16
wDasErr = KDAWrite% (hDev, 0, dwDAValue)
```

K_DIRead

Boards Supported

All

Purpose

Reads a single digital input value.

Prototype

C/C++

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

Turbo Pascal

Function K_DIRead (*hDev* : Longint; *nChan* : Byte; *pData* : Pointer) : Word;

Turbo Pascal for Windows

Function K_DIRead (*hDev* : Longint; *nChan* : Byte; *pData* : Pointer) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function reads the values of all digital input lines on the board specified by *hDev* and stores the value in *pData*.

Make sure that the variable you declare for *pData* is large enough to accommodate the number of digital input lines you are using. Refer to page 2-46 for a description of the digital input lines.

See Also

K_IntStart, K_SyncStart

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDIValue : Word;
...
wDasErr := K_DIRead (hDev, 0, @wDIValue);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDIValue : Word;
...
wDasErr := K_DIRead (hDev, 0, @wDIValue);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Allocates a buffer for a DMA-mode analog input operation.

Prototype

C/C++

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

Turbo Pascal

Function K_DMAAlloc (*hFrame* : Longint; *dwSamples* : Longint;
pBuf : Pointer; Var *phMem* : Word) : Word;

Turbo Pascal for Windows

Function K_DMAAlloc (*hFrame* : Longint; *dwSamples* : Longint;
pBuf : Pointer; Var *phMem* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KDMAALLOC% ALIAS "K_DMAAlloc"
(BYVAL *hFrame* AS LONG, BYVAL *dwSamples* AS LONG,
SEG *pBuf* AS LONG, SEG *phMem* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
dwSamples	Number of samples. Valid values: **1** to **32768**
pBuf	Starting address of the allocated buffer.
phMem	Handle associated with the allocated buffer.

K_DMAAlloc (cont.)

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 far memory address of a buffer that is suitable for a DMA-mode analog input operation and *phMem* contains the handle associated with the allocated buffer.

The data in the allocated buffer is stored as counts. For information on converting the count values to voltages, refer to Appendix B.

Turbo Pascal (for DOS) and BASIC require that you redistribute available memory before you dynamically allocate a buffer. Refer to “Reducing the Memory Heap” on page 3-11 (Turbo Pascal) or page 3-24 (BASIC) for additional information.

See Also

K_DMAFree, K_SetDMABuf

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
pBuf : Pointer;   { DMA buffer pointer }
hMem : Word;   { Handle to DMA buffer }
...
wDasErr := K_DMAAlloc (hAD, dwSamples, @pBuf, hMem);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
pBuf : Pointer;   { DMA buffer pointer }
hMem : Word;   { Handle to DMA buffer }
...
wDasErr := K_DMAAlloc (hAD, dwSamples, @pBuf, hMem);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Global pBuf As Long
Global hMem As Integer
...
wDasErr = K_DMAAlloc (hAD, dwSamples, pBuf, hMem)
```

BASIC

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

K_DMAFree

Boards Supported

All

Purpose

Frees a buffer allocated for a DMA-mode analog input operation.

Prototype

C/C++

DASErr far pascal K_DMAFree (WORD *hMem*);

Turbo Pascal

Function K_DMAFree (*hMem* : Word) : Integer;

Turbo Pascal for Windows

Function K_DMAFree (*hMem* : Word) : Integer;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KDMAFREE% ALIAS "K_DMAFree"
(BYVAL *hMem* 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 dynamically using **K_DMAAlloc**.

See Also

K_DMAlloc, K_SetDMABuf

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_DMAFree (hMem);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_DMAFree (hMem);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_DMAFree (hMem)
```

BASIC

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

K_DMAStart

Boards Supported

All

Purpose

Starts a DMA-mode analog input operation.

Prototype

C/C++

DASErr far pascal K_DMAStart (DWORD *hFrame*);

Turbo Pascal

Function K_DMAStart (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_DMAStart (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KDMASTART% ALIAS "K_DMAStart"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_DMAStatus, K_DMAStop

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_DMAStart (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_DMAStart (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_DMAStart (hAD)
```

BASIC

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

K_DMAStatus

Boards Supported

All

Purpose

Gets status of a DMA-mode analog input operation.

Prototype

C/C++

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

Turbo Pascal

Function K_DMAStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word;

Turbo Pascal for Windows

Function K_DMAStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KDMASTATUS% ALIAS "K_DMAStatus" (BYVAL *hFrame* AS LONG, SEG *pStatus* AS INTEGER, SEG *pCount* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of DMA-mode analog input operation. Value stored: **0** for DMA operation idle **1** for DMA operation active
pCount	Number of samples that were acquired into the current buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

A DMA data overrun occurs if data is lost when the transfer of data between memory and the board is slower than the rate at which the hardware requests the data.

See Also

K_DMAStart, K_DMAStop

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_DMAStatus (hAD, wStatus, dwCount);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_DMAStatus (hAD, wStatus, dwCount);
```

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Stops a DMA-mode analog input operation.

Prototype

C/C++

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

Turbo Pascal

Function K_DMAStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word;

Turbo Pascal for Windows

Function K_DMAStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KDMASTOP% ALIAS "K_DMAStop" (BYVAL *hFrame* AS LONG, SEG *pStatus* AS INTEGER, SEG *pCount* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of DMA-mode analog input operation. Value stored: **0** for DMA operation idle **1** for DMA operation active
pCount	Number of samples that were acquired into the current buffer.

K_DMAStop (cont.)

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

A DMA data overrun occurs if data is lost when the transfer of data between memory and the board is slower than the rate at which the hardware requests the data.

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

See Also

K_DMAStart, K_DMAStatus

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_DMAStop (hAD, wStatus, dwCount);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_DMAStop (hAD, wStatus, dwCount);
```

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_DOWrite

Boards Supported

All

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_DOWrite (*hDev* : Longint; *nChan* : Byte; *dwData* : Longint) : Word;

Turbo Pascal for Windows

Function K_DOWrite (*hDev* : Longint; *nChan* : Byte; *dwData* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the board.
nChan	Digital output channel. Valid value: **0**
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 board specified by *hDev*.

Refer to page 2-46 for a description of the digital I/O lines.

See Also

K_IntStart, K_SyncStart

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
dwDOValue : Longint;
...
dwDOValue := $5;
wDasErr := K_DOWrite (hDev, 0, dwDOValue);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
dwDOValue : Longint;
...
dwDOValue := $5;
wDasErr := K_DOWrite (hDev, 0, dwDOValue);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

DAS-1601, DAS-1602, DAS-1401, DAS-1402

Purpose

Converts the format of a channel-gain queue.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

You cannot use a channel-gain queue with DAS-1200 Series boards.

K_FormatChnGAry (cont.)

See Also K_SetChnGAry, K_RestoreChnGAry

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_FreeDevHandle

Boards Supported

All

Purpose

Frees a previously specified device handle.

Prototype

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

Turbo Pascal
Not supported

Turbo Pascal for Windows
Function K_FreeDevHandle (*hDev* : Longint) : Word;
far; external 'DASSHELL';

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

BASIC
Not supported

Parameters

hDev	Device handle you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_GetDevHandle

K_FreeDevHandle (cont.)

Usage

C/C++

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

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_FreeDevHandle (hDev);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

K_FreeFrame

Boards Supported

All

Purpose

Frees a frame.

Prototype

C/C++

DASErr far pascal K_FreeFrame (DWORD *hFrame*);

Turbo Pascal

Function K_FreeFrame (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_FreeFrame (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to frame you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_GetADFrame, K_GetDAFrame, K_GetDIFrame, K_GetDOFrame

K_FreeFrame (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_FreeFrame (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_FreeFrame (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Get a DAS board's analog input channel configuration.

Prototype

C/C++

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

Turbo Pascal

Function K_GetADConfig (*hDev* : Longint; Var *pMode* : Word) : Word;

Turbo Pascal for Windows

Function K_GetADConfig (*hDev* : Longint; Var *pMode* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board specified by *hDev*, this function stores the code that indicates the analog input channel configuration in *pMode*.

K_GetADConfig (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wADConfig : Word;
...
wDasErr := K_GetADConfig (hDev, wADConfig);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wADConfig : Word;
...
wDasErr := K_GetADConfig (hDev, wADConfig);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_GetADFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetADFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the board.
phFrame	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 *phFrame*.

The frame is initialized to its default settings; refer to Table 2-1 on page 2-8 for a list of the default settings.

See Also

K_ClearFrame, K_FreeFrame

K_GetADFrame (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
hAD : Longint;
...
wDasErr := K_GetADFrame (hDev, hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hAD : Longint;
...
wDasErr := K_GetADFrame (hDev, hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Get a DAS board's analog input range type.

Prototype

C/C++

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

Turbo Pascal

Function K_GetADMode (*hDev* : Longint; Var *pMode* : Word) : Word;

Turbo Pascal for Windows

Function K_GetADMode (*hDev* : Longint; Var *pMode* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the board specified by *hDev*, this function stores the code that indicates the analog input range type in *pMode.*

K_GetADMode (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wADMode : Word;
...
wDasErr := K_GetADMode (hDev, wADMode);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wADMode : Word;
...
wDasErr := K_GetADMode (hDev, wADMode);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_GetClkRate (*hFrame* : Longint; Var *pRate* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetClkRate (*hFrame* : Longint; Var *pRate* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

K_GetClkRate (cont.)

See Also K_SetClkRate

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
dwRate : Longint;
...
wDasErr := K_GetClkRate (hAD, dwRate);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
dwRate : Longint;
...
wDasErr := K_GetClkRate (hAD, dwRate);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

DAS-1601, DAS-1602

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_GetDAFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetDAFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KGETDAFRAME% ALIAS "K_GetDAFrame"
(BYVAL *hDev* AS LONG, SEG *phFrame* AS LONG)

Parameters

hDev	Handle associated with the board.
phFrame	Handle to the frame that defines the analog output operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

The frame is initialized to its default settings; refer to Table 2-5 on page 2-32 for a list of the default settings.

K_GetDAFrame (cont.)

See Also K_FreeFrame, K_ClearFrame

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
hDA : Longint;
...
wDasErr := K_GetDAFrame (hDev, hDA);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hDA : Longint;
...
wDasErr := K_GetDAFrame (hDev, hDA);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Initializes any Keithley DAS board.

Prototype

C/C++

DASErr far pascal K_GetDevHandle (DWORD *hDrv*,
WORD *nBoardNum*, DWORD far * *phDev*);

Turbo Pascal

Not supported

Turbo Pascal for Windows

Function K_GetDevHandle (*hDrv* : Longint; *nBoardNum* : Integer;
Var *phDev* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

Not supported

Parameters

hDrv	Driver handle of the associated Function Call Driver.
nBoardNum	Board number. Valid values: **0** or **1**
phDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

K_GetDevHandle (cont.)

Remarks

This function initializes the board associated with *hDrv* and specified by *nBoardNum*, and stores the device 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 device handle. Your program should not modify the value stored in *phDev*.

See Also

K_FreeDevHandle

Usage

C/C++

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

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hDev : Longint;
...
wDasErr := K_GetDevHandle (hDrv, 0, hDev);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

Boards Supported

All

Purpose

Accesses a DI frame for a digital input operation.

Prototype

C/C++

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

Turbo Pascal

Function K_GetDIFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetDIFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KGETDIFRAME% ALIAS "K_GetDIFrame"
(BYVAL *hDev* AS LONG, SEG *phFrame* AS LONG)

Parameters

hDev	Handle associated with the board.
phFrame	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 or interrupt-mode digital input operation on the board specified by *hDev*, and accesses an available digital input frame with the handle *phFrame*.

The frame is initialized to its default settings; refer to Table 2-6 on page 2-42 for a list of the default settings.

K_GetDIFrame (cont.)

See Also K_FreeFrame, K_ClearFrame

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
hDI : Longint;
...
wDasErr := K_GetDIFrame (hDev, hDI);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hDI : Longint;
...
wDasErr := K_GetDIFrame (hDev, hDI);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Accesses a DO frame for a digital output operation.

Prototype

C/C++

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

Turbo Pascal

Function K_GetDOFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetDOFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KGETDOFRAME% ALIAS "K_GetDOFrame"
(BYVAL *hDev* AS LONG, SEG *phFrame* AS LONG)

Parameters

hDev	Handle associated with the board.
phFrame	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 or interrupt-mode digital output operation on the board specified by *hDev*, and accesses an available digital output frame with the handle *phFrame*.

The frame is initialized to its default settings; refer to Table 2-7 on page 2-43 for a list of the default settings.

K_GetDOFrame (cont.)

See Also K_FreeFrame, K_ClearFrame

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
hDO : Longint;
...
wDasErr := K_GetDOFrame (hDev, hDO);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hDO : Longint;
...
wDasErr := K_GetDOFrame (hDev, hDO);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Gets the address of an error message string.

Prototype

C/C++

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

Turbo Pascal

Not supported

Turbo Pascal for Windows

Not supported

Visual Basic for Windows

Not supported

BASIC

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

K_GetErrMsg (cont.)

Usage

C/C++

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

Boards Supported

All

Purpose

Gets the current DAS shell version.

Prototype

C/C++

DASErr far pascal K_GetShellVer (WORD far **pVersion*);

Turbo Pascal

Function K_GetShellVer (Var *pVersion* : Word) : Word;

Turbo Pascal for Windows

Function K_GetShellVer (Var *pVersion* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_GetShellVer Lib "DASSHELL.DLL"
(*pVersion* As Integer) As Integer

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

K_GetShellVer (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wShellVer : Word;
...
wDasErr := K_GetShellVer (wShellVer);
FormatStr (VerStr, '%4x', wShellVer / 256, '.', wShellVer And $ff);
writeln(' Shell Ver ', VerStr);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wShellVer : Word;
...
wDasErr := K_GetShellVer (wShellVer);
FormatStr (VerStr, '%4x', wShellVer / 256, '.', wShellVer And $ff);
writeln(' Shell Ver ', VerStr);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_GetVer

Boards Supported

All

Purpose

Gets revision numbers.

Prototype

C/C++

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

Turbo Pascal

Function K_GetVer (*hDev* : Longint; Var *pSpecVer* : Word; Var *pDrvVer* : Word) : Word;

Turbo Pascal for Windows

Function K_GetVer (*hDev* : Longint; Var *pSpecVer* : Word; Var *pDrvVer* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the 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 2-byte (16-bit) integers; the high byte of each contains the major revision level and the low byte of each contains the minor revision level. For example, if the driver version number is 2.10, the major revision level is 2 and the minor revision level is 10; therefore, the high byte of *pDrvVer* contains the value of **2** (512) and the low byte of *pDrvVer* contains the value of **10**; the value of both bytes is 522.

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

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

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
nSpecVer : Word;
nDrvVer : Word;
...
wDasErr := K_GetVer (hDev, nSpecVer, nDrvVer);
FormatStr (VerStr, ' %4x ', nDrvVer / 256, '.', nDrvVer And $ff);
writeln(' Driver Ver ', VerStr);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
nSpecVer : Word;
nDrvVer : Word;
...
wDasErr := K_GetVer (hDev, nSpecVer, nDrvVer);
FormatStr (VerStr, ' %4x ', nDrvVer / 256, '.', nDrvVer And $ff);
writeln(' Driver Ver ', VerStr);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_IntAlloc (*hFrame* : Longint; *dwSamples* : Longint; *pBuf* : Pointer; Var *phMem* : Word) : Word;

Turbo Pascal for Windows

Function K_IntAlloc (*hFrame* : Longint; *dwSamples* : Longint; *pBuf* : Pointer; Var *phMem* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame — Handle to the frame that defines the operation.

dwSamples — Number of samples.
Valid values:

Analog I/O operations	**1** to **5000000**
Digital I/O operations	**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 buffer of the size specified by *dwSamples*, and stores the starting address of the buffer in *pBuf* and the handle of the buffer in *phMem*.

For analog input and analog output operations, the data in the allocated buffer is stored as counts. Refer to Appendix B for information on converting a count value to voltage (for analog input operations) or for converting a voltage value to a count (for analog output operations).

Turbo Pascal (for DOS) and BASIC require that you redistribute available memory before you dynamically allocate a buffer. Refer to “Reducing the Memory Heap” on page 3-11 (Turbo Pascal) or page 3-24 (BASIC) for additional information.

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

See Also

K_IntFree, K_SetBuf

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
pBuf : Pointer;   { buffer pointer }
hMem : Word;   { Handle to buffer }
...
wDasErr := K_IntAlloc (hAD, 1000, pBuf, hMem);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
pBuf : Pointer;   { buffer pointer }
hMem : Word;   { Handle to buffer }
...
wDasErr := K_IntAlloc (hAD, 1000, pBuf, hMem);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_IntFree

Boards Supported

All

Purpose

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

Prototype

C/C++

DASErr far pascal K_IntFree (WORD *hMem*);

Turbo Pascal

Function K_IntFree (*hMem* : Word) : Integer;

Turbo Pascal for Windows

Function K_IntFree (*hMem* : Word) : Integer; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hMem	Handle to interrupt buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_IntAlloc

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_IntFree (hMem);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_IntFree (hMem);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_IntStart

Boards Supported

All

Purpose

Starts an interrupt-mode operation.

Prototype

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

Turbo Pascal
Function K_IntStart (*hFrame* : Longint) : Word;

Turbo Pascal for Windows
Function K_IntStart (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

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

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

Parameters

hFrame Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

Refer to the following pages for an illustration of the programming tasks associated with interrupt-mode operations:

Analog input	page 1-8
Analog output	page 1-14
Digital input	page 1-18
Digital output	page 1-21

See Also K_IntStatus, K_IntStop

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_IntStart (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_IntStart (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_IntStart (hAD)
```

BASIC

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

K_IntStatus

Boards Supported

All

Purpose

Gets status of interrupt-mode operation.

Prototype

C/C++

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

Turbo Pascal

Function K_IntStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word;

Turbo Pascal for Windows

Function K_IntStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt operation. See **Remarks** below for value stored
pCount	Current number of samples transferred.

Return Value

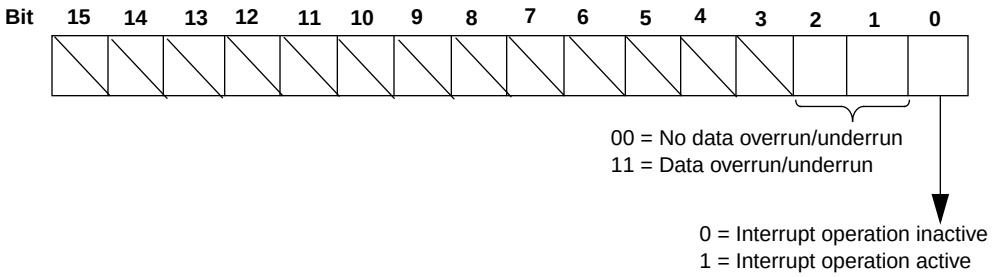
Error/status code. Refer to Appendix A.

Remarks

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

A data overrun/underrun occurs if data is lost when the transfer of data to or from computer memory cannot keep up with the clock rate.

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.
- Bits 1 and 2: For input operations, these bits indicate whether a data overrun occurred. For output operations, these bits indicate whether a data underrun occurred. The overrun or underrun event automatically stops all conversions.

  Both bits 1 and 2 are set when the driver detects an overrun/underrun event. It is recommended that you read bit 2 only; bit 1 is set to provide compatibility with previous revisions of the driver.
- Bits 3 to 15: Unassigned.

See Also

K_IntStart, K_IntStop

K_IntStatus (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_IntStatus (hAD, wStatus, dwCount);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_IntStatus (hAD, wStatus, dwCount);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Stops an interrupt-mode operation.

Prototype

C/C++

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

Turbo Pascal

Function K_IntStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word;

Turbo Pascal for Windows

Function K_IntStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pCount* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt operation.
pCount	Number of samples.

Return Value

Error/status code. Refer to Appendix A.

K_IntStop (cont.)

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*.

Refer to page 4-105 for the meaning of the value stored in *pStatus*.

A data overrun/underrun occurs if data is lost when the transfer of data between computer memory and the board is too slow.

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

See Also

K_IntStart, K_IntStatus

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_IntStop (hAD, wStatus, dwCount);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wStatus : Word;
dwCount : Longint;
...
wDasErr := K_IntStop (hAD, wStatus, dwCount);
```

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

KMakeDMABuf

Boards Supported

All

Purpose

Converts a local array to a buffer suitable for a DMA-mode operation.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

Visual Basic for Windows
Not supported

BASIC
DECLARE FUNCTION KMakeDMABuf% (*dwSamples* AS LONG, *pBuf()* AS INTEGER, *pBufAddr* AS LONG, *pStartIx* AS INTEGER)

Parameters

dwSamples	Number of samples.
pBuf	$DYNAMIC integer array.
pBufAddr	Starting address of the DMA buffer.
pStartIx	Index into *pBuf* that identifies the location in which the first sample is stored.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function ensures that the array address provided to **K_SetDMABuf** is suitable for a DMA-mode analog input operation.

Instead of using this function, it is recommended that you use the BASIC **SetMem** function in conjunction with the **K_DMAAlloc** function to allocate a memory buffer. Refer to page 3-24 for more information.

See Also

K_SetDMABuf

Usage

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
$DYNAMIC DIM ADBuf(10000)As Integer
$STATIC
DIM pDMABuf AS LONG
...
wDasErr = KMakeDMABuf% (dwSamples, ADBuf, pDMABuf, pStartIx)
```

K_MoveArrayToBuf

Boards Supported

All

Purpose

Transfers data from a local array within the program to a buffer allocated through **K_IntAlloc** or **K_DMAAlloc**.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
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

BASIC
DECLARE FUNCTION KMOVEARRAYTOBUF% ALIAS "K_MoveDataBuf" (ByVal *pDest* As Long, SEG *pSource* As Integer, ByVal *nCount* As Integer)

Parameters

pDest	Address of destination buffer.
pSource	Address of source array.
nCount	Number of samples to transfer. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

If the buffer used to store output data for your program was allocated through **K_IntAlloc** or **K_DMAAlloc**, the data in the buffer is not accessible to the program; you must use **K_MoveArrayToBuf** to move the data from a local array within the program to a dynamically allocated buffer. If the array used to store output data for your program was dimensioned locally within the program's memory area, the data in the array is accessible to your program and you do not have to use this function.

See Also

K_DMAAlloc, K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Dim DACArray(2000) As Integer
...
wDasErr = K_IntAlloc (hDA, 1000, pBuf, hMem)
...
wDasErr = K_MoveArrayToBuf (pBuf, DACArray(0), 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM DACArray(2000) AS INTEGER
...
wDasErr = KIntAlloc% (hDA, dwSamples, pBuf, hMem)
...
wDasErr = KMoveArrayToBuf% (pBuf, DACArray(0), 1000)
```

K_MoveArrayToBufL

Boards Supported

All

Purpose

Transfers data from a local long array within the program to a buffer allocated through **K_IntAlloc** or **K_DMAAlloc**.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

BASIC
DECLARE FUNCTION KMOVEARRAYTOBUFL% ALIAS "K_MoveDataBuf" (ByVal *pDest* As Long, SEG *pSource* As Long, ByVal *nCount* As Integer)

Parameters

pDest	Address of destination buffer.
pSource	Address of source array.
nCount	Number of samples to transfer. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

This function is intended for digital output operations when you are writing to more than 16 digital output lines.

If the buffer used to store output data for your program was allocated through **K_IntAlloc** or **K_DMAAlloc**, the data in the buffer is not accessible to the program; you must use **K_MoveArrayToBufL** to move the data from a local array within the program to a dynamically allocated buffer. If the long array used to store output data for your program was dimensioned locally within the program's memory area, the data in the array is accessible to your program and you do not have to use this function.

See Also

K_DMAAlloc, K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Dim DOArray(2000) As Integer
...
wDasErr = K_IntAlloc (hDO, 1000, pBuf, hMem)
...
wDasErr = K_MoveArrayToBufL (pBuf, DOArray(0), 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM DOArray(2000) AS INTEGER
...
wDasErr = KIntAlloc% (hDO, dwSamples, pBuf, hMem)
...
wDasErr = KMoveArrayToBufL% (pBuf, DOArray(0), 1000)
```

K_MoveBufToArray

Boards Supported

All

Purpose

Transfers data from a buffer allocated through **K_IntAlloc** or **K_DMAAlloc** to a local array within your program.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

BASIC
DECLARE FUNCTION KMOVEBUFTOARRAY% ALIAS "K_MoveDataBuf" (SEG *pDest* As Integer, ByVal *pSource* As Long, ByVal *nCount* As Integer)

Parameters

pDest	Address of destination array.
pSource	Address of source buffer.
nCount	Number of samples to transfer. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

See Also

K_DMAAlloc, K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Dim ADArray(2000) As Integer
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
...
wDasErr = K_MoveBufToArray (ADArray(0), pBuf, 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ADArray(2000) AS INTEGER
...
wDasErr = KIntAlloc% (hAD, 1000, pBuf, hMem)
...
wDasErr = KMoveBufToArray% (ADArray(0), pBuf, 1000)
```

K_MoveBufToArrayL

Boards Supported

All

Purpose

Transfers data from a buffer allocated through **K_IntAlloc** or **K_DMAAlloc** to a local long array within your program.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

BASIC
DECLARE FUNCTION KMOVEBUFTOARRAYL% ALIAS "K_MoveDataBuf" (SEG *pDest* As Long, ByVal *pSource* As Long, ByVal *nCount* As Integer)

Parameters

pDest	Address of destination array.
pSource	Address of source buffer.
nCount	Number of samples to transfer. Valid values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

This function is intended for digital input operations when you are reading more than 16 digital input lines.

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

See Also

K_DMAAlloc, K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Dim DIArray(2000) As Integer
...
wDasErr = K_IntAlloc (hDI, 1000, pBuf, hMem)
...
wDasErr = K_MoveBufToArrayL (DIArray(0), pBuf, 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM DIArray(2000) AS INTEGER
...
wDasErr = KIntAlloc% (hDI, 1000, pBuf, hMem)
...
wDasErr = KMoveBufToArrayL% (DIArray(0), pBuf, 1000)
```

K_MoveDataBuf

Boards Supported

All

Purpose

Moves a specified number of samples from one memory area to another.

Prototype

C/C++

DASErr far pascal K_MoveDataBuf (short far **pDest*,
short far **pSource*, WORD *nCount*) ;

Turbo Pascal

Function K_MoveDataBuf (*pDest* : Longint; *pSource* : Longint;
nCount : Word) : Integer;

Turbo Pascal for Windows

Function K_MoveDataBuf (*pDest* : Longint; *pSource* : Longint;
nCount : Word) : Integer; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_MoveDataBuf Lib "DASSHELL.DLL"
(*pDest* As Integer, ByVal *pSource* As Long, ByVal *nCount* As Integer)
As Integer

BASIC

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

Parameters

pDest	Address of destination buffer.
pSource	Address of source buffer.
nCount	Number of samples to transfer. Value values: **1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_DMAAlloc, K_IntAlloc

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
...
wDasErr = K_MoveDataBuf (ADArray[0], pBuf, 1000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KIntAlloc% (hAD, 1000, pBuf, hMem)
...
wDasErr = KMoveDataBuf% (ADArray[0], pBuf, 1000)
```

K_OpenDriver

Boards Supported

All

Purpose

Initializes any Keithley DAS Function Call Driver.

Prototype

C/C++

DASErr far pascal K_OpenDriver (char far * *szDrvName*, char far * *szCfgName*, DWORD far * *phDrv*);

Turbo Pascal

Not supported

Turbo Pascal for Windows

Function K_OpenDriver (Var *szDrvName* : char; Var *szCfgName* : char; Var *phDrv* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_OpenDriver Lib "DASSHELL.DLL" (ByVal *szDrvName* As String, ByVal *szCfgName* As String, *phDrv* As Long) As Integer

BASIC

Not supported

Parameters

szDrvName	Driver name. Valid value: **"DAS1600"** (for DAS-1600/1400/1200 Series boards)
szCfgName	Driver configuration file. Valid value: The name of a configuration file; **0** if the driver has already been opened
phDrv	Handle associated with the driver.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DAS-1600/1400/1200 Series Function Call Driver according to the information in the configuration file specified by *szCfgName*, and stores the driver handle in *phDrv*.

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

For DAS-1600/1400/1200 Series boards, the string stored in *szDrvName* must be DAS1600.

The value stored in *phDrv* 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 *phDrv.*

You create a configuration file using the CFG1600.EXE utility. Refer to your board user's guide for more information. If *szCfgName* = 0, **K_OpenDriver** checks whether the driver has already been opened and linked to a configuration file and if it has, uses the current configuration; this is useful in the Windows environment.

See Also

DAS1600_DevOpen

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
DWORD hDrv;
...
wDasErr = K_OpenDriver ("DAS1600", "DAS1600.CFG", &hDrv);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
szDrvName : String;
szCfgName : String;
hDrv : Longint;
...
szDrvName := 'DAS1600' + #0;
szCfgName := 'DAS1600.CFG' + #0;
wDasErr := K_OpenDriver (szDrvName[1], szCfgName[1], hDrv);
```

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
DIM hDrv As Long
...
wDasErr = K_OpenDriver ("DAS1600", "DAS1600.CFG", hDrv)
```

Boards Supported

DAS-1601, DAS-1602, DAS-1401, DAS-1402

Purpose

Restores a converted channel-gain queue.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

You cannot use a channel-gain queue with DAS-1200 Series boards.

See Also

K_FormatChnGAry, K_SetChnGAry

K_RestoreChnGAry (cont.)

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

Specifies burst conversion mode.

Prototype

C/C++

DASErr far pascal K_SetADFreeRun (DWORD *hFrame*);

Turbo Pascal

Function K_SetADFreeRun (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetADFreeRun (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETADFREERUN% ALIAS
"K_SetADFreeRun" (BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function sets the conversion mode for the operation defined by *hFrame* to burst mode. Refer to page 2-20 for information on conversion modes.

See Also

K_ClrADFreeRun

K_SetADFreeRun (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetADFreeRun (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetADFreeRun (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetADFreeRun (hAD)
```

BASIC

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

Boards Supported

All

Purpose

Sets up an analog trigger.

Prototype

C/C++

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

Turbo Pascal

Function K_SetADTrig (*hFrame* : Longint; *nOpt* : Word; *nChan* : Word; *dwLevel* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetADTrig (*hFrame* : Longint; *nOpt* : Word; *nChan* : Word; *dwLevel* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.

nOpt — Analog trigger polarity and sensitivity.
Valid values:

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

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

dwLevel — Level at which the trigger event occurs.
Valid values: **0** to **4095**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

The *nOpt* variable sets the value of the Trigger Polarity and Trigger Sensitivity elements.

The *nChan* variable sets the value of the Trigger Channel element.

The *dwLevel* variable sets the value of the Trigger Level element.

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

See Also

K_SetTrig

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetADTrig (hAD, 0, 0, 2047);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetADTrig (hAD, 0, 0, 2047);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

K_SetBuf

Boards Supported

All

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_SetBuf (*hFrame* : Longint; *pBuf* : Pointer; *dwSamples* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetBuf (*hFrame* : Longint; *pBuf* : Pointer; *dwSamples* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame — Handle to the frame that defines the operation.

pBuf — Starting address of buffer.

dwSamples — Number of samples.
Valid values:

Analog I/O operations	**1** to **5000000**
Digital I/O operations	**1** to **32767**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

For C and Pascal programs, use this function whether you dimensioned your array locally or allocated your buffer dynamically using **K_IntAlloc**. For a buffer allocated dynamically using **K_DMAAlloc**, use **K_SetDMABuf**.

For C, make sure that you use proper typecasting to prevent C/C++ type-mismatch warnings. For Pascal, a special procedure is needed to satisfy the type-checking requirements; refer to page 3-12 for more information.

For Visual Basic for Windows and BASIC, use this function only for a buffer allocated dynamically using **K_IntAlloc**. For a buffer allocated dynamically using **K_DMAAlloc**, use **K_SetDMABuf**. For a locally dimensioned array, use **K_SetBufI**.

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_DMAAlloc, K_IntAlloc, K_SetBufI, K_SetDMABuf

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
TYPE
BufType = Array [0..1] of Integer;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr (pBuf), hMem);
wDasErr := K_SetBuf (hAD, pBuf, 1000);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
TYPE
BufType = Array [0..1] of Integer;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr (pBuf), hMem);
wDasErr := K_SetBuf (hAD, pBuf, 1000);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

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

Boards Supported

All

Purpose

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

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

K_SetBufI (cont.)

Remarks

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

Do not use this function for C and Pascal; for these languages, use **K_SetBuf**.

For Visual Basic for Windows and BASIC, use this function only for a locally dimensioned array. For a buffer allocated dynamically using **K_IntAlloc**, use **K_SetBuf**. For a buffer allocated dynamically using **K_DMAAlloc**, use **K_SetDMABuf**.

The *pBuf* variable sets the value of the Buffer element.

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

See Also

K_DMAAlloc, K_IntAlloc, K_SetBuf, K_SetDMABuf

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
Dim ADData(2000) AS LONG
...
wDasErr = KSetBufI% (hAD, ADData(0), 2000)
```

Boards Supported

All

Purpose

Specifies the starting address of a locally dimensioned long array and the number of samples in the array.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of the user-dimensioned long array.
dwSize	Number of samples.

Return Value

Error/status code. Refer to Appendix A.

K_SetBufL (cont.)

Remarks

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

This function is useful for digital I/O operations when you are accessing more than 16 of the digital input or digital output lines.

Do not use this function for C and Pascal; for these languages, use **K_SetBuf**.

For Visual Basic for Windows and BASIC, use this function only for a locally dimensioned array. For a buffer allocated dynamically using **K_IntAlloc**, use **K_SetBuf**. For a buffer allocated dynamically using **K_DMAAlloc**, use **K_SetDMABuf**.

The *pBuf* variable sets the value of the Buffer element.

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

See Also

K_DMAAlloc, K_IntAlloc, K_SetBuf, K_SetDMABuf

Usage

Visual Basic for Windows
(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Dim DOData(2000) As Long
...
wDasErr = K_SetBufL (hAD, DOData(0), 2000)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
Dim DOData(2000) AS LONG
...
wDasErr = KSetBufL% (hAD, DOData(0), 2000)
```

K_SetBurstTicks

Boards Supported

All

Purpose

Specifies the count value used to adjust the settling time.

Prototype

C/C++

DASErr far pascal K_SetBurstTicks (DWORD *hFrame*, short *nTicks*);

Turbo Pascal

Function K_SetBurstTicks (*hFrame* : Longint; *nTicks* : Word) : Word;

Turbo Pascal for Windows

Function K_SetBurstTicks (*hFrame* : Longint; *nTicks* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETBURSTTICKS% ALIAS "K_SetBurstTicks" (BYVAL *hFrame* AS LONG, BYVAL *nTicks* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nTicks	Count value used to adjust the settling time. Valid values: **2** to **255**

Return Value

Error/status code. Refer to Appendix A.

K_SetBurstTicks (cont.)

Remarks

For the operation defined by *hFrame*, this function specifies the count value used to adjust the settling time in *nTicks*.

Refer to page 2-20 for more information.

See Also

K_SetADFreeRun

Usage

C/C++

```
#include "DASDECL.H"   // Use "DASDECL.HPP for C++
...
wDasErr = K_SetBurstTicks (hAD, 10);
```

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetBurstTicks (hAD, 10);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetBurstTicks (hAD, 10);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetBurstTicks (hAD, 10)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetBurstTicks% (hAD, 10)
```

Boards Supported

All

Purpose

Specifies a single channel.

Prototype

C/C++

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

Turbo Pascal

Function K_SetChn (*hFrame* : Longint; *nChan* : Word) : Word;

Turbo Pascal for Windows

Function K_SetChn (*hFrame* : Longint; *nChan* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
nChan	Channel on which to perform operation. Valid values: **0** to **255** (analog input) **0** or **1** (analog output)

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*.

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

K_SetChn (cont.)

See Also K_SetStartStopChn, K_SetStartStopG

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetChn (hAD, 2);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetChn (hAD, 2);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetChn (hAD, 2)
```

BASIC

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

Boards Supported

DAS-1601, DAS-1602, DAS-1401, DAS-1402

Purpose

Specifies the starting address of a channel-gain queue.

Prototype

C/C++

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

Turbo Pascal

Function K_SetChnGAry (*hFrame* : Longint;
Var *pArray* : Integer) : Word;

Turbo Pascal for Windows

Function K_SetChnGAry (*hFrame* : Longint;
Var *pArray* : Integer) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

Refer to page 2-19 for information on setting up a channel-gain queue.

K_SetChnGAry (cont.)

If you created your channel-gain queue in BASIC or Visual Basic for Windows, you must use **K_FormatChnGAry** to convert the channel-gain queue before you specify the address with **K_SetChnGAry**.

You cannot use a channel-gain queue with DAS-1200 Series boards.

See Also K_FormatChnGAry, K_RestoreChnGAry

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
// DECLARE AND INITIALIZE CHAN/GAIN PAIRS
// (GainChanTable-TYPE IS DEFINED IN dasdecl.h)
GainChanTable ChanGainArray= {2,   // # of entries
   0, 0,    // chan 0, gain 1
   1, 1};   // chan 1, gain 2
...
wDasErr = K_SetChnGAry (hAD, &ChanGainArray);
```

Turbo Pascal

```
uses D1600TP7;
...
{ Define Gain/Channel array type }
TYPE GainChanTable = Record
       num_of_codes : Integer;
       queue : Array[0..15] of Byte;
       END;
CONST ChanGainArray : GainChanTable = (
       num_of_codes : (8); { # of chan/gain pairs }
       queue : (0,0, 1,1)
);
...
wDasErr := K_SetChnGAry (hAD, ChanGainArray.num_of_codes);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
{ Define Gain/Channel array type }
TYPE GainChanTable = Record
       num_of_codes : Integer;
       queue : Array[0..15] of Byte;
       END;
CONST ChanGainArray : GainChanTable = (
       num_of_codes : (8);   { # of chan/gain pairs }
       queue : (0,0, 1,1)
);
...
wDasErr := K_SetChnGAry (hAD, ChanGainArray.num_of_codes);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
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 (hAD, ChanGainArray(0))
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM 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 = KFormatChnGAry% (ChanGainArray(0))
wDasErr = KSetChnGAry% (hAD, ChanGainArray(0))
```

K_SetClk

Boards Supported

All

Purpose

Specifies the pacer clock source.

Prototype

C/C++
DASErr far pascal K_SetClk (DWORD *hFrame*, short *nMode*);

Turbo Pascal
Function K_SetClk (*hFrame* : Longint; *nMode* : Word) : Word;

Turbo Pascal for Windows
Function K_SetClk (*hFrame* : Longint; *nMode* : Word) : Word;
far; external 'DASSHELL';

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

BASIC
DECLARE FUNCTION KSETCLK% ALIAS "K_SetClk"
(BYVAL *hFrame* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nMode	Pacer clock source. Valid values: **0** for Internal **1** for External

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the pacer clock source in *nMode*.

The *nMode* variable sets the Clock Source element.

The internal clock source is the output of the onboard 82C54 counter/timer; an external clock source is an external signal connected to the IP0/TRIG0/XPCLK pin (25).

For more information on pacer clock sources, refer to the following pages:

Analog input operations	page 2-22
Analog output operations	page 2-35
Digital I/O operations	page 2-49

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

See Also

K_SetClkRate

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetClk (hAD, 1);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetClk (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetClk% (hAD, 1)
```

K_SetClkRate

Boards Supported

All

Purpose

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

Prototype

C/C++

DASErr far pascal K_SetClkRate (DWORD *hFrame*,
DWORD *dwDivisor*);

Turbo Pascal

Function K_SetClkRate (*hFrame* : Longint; *dwDivisor* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetClkRate (*hFrame* : Longint; *dwDivisor* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETCLKRATE% ALIAS "K_SetClkRate"
(BYVAL *hFrame* AS LONG, BYVAL *dwDivisor* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
dwDivisor	Number of clock ticks between conversions. Valid values: **10** to **4294967295** (1 MHz) **100** to **4294967295** (10 MHz)

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 Pacer Clock Rate element.

Refer to page 2-23 for more information about the internal pacer clock.

See Also

K_SetClk, K_GetClkRate

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
dwClkDiv : Longint;
...
dwClkDiv := 1000000 / 10000;
wDasErr := K_SetClkRate (hAD, dwClkDiv);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
dwClkDiv : Longint;
...
dwClkDiv := 1000000 / 10000;
wDasErr := K_SetClkRate (hAD, dwClkDiv);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwClkDiv AS LONG
...
dwClkDiv = 1000000 / 10000
wDasErr = KSetClkRate% (hAD, dwClkDiv)
```

Boards Supported

All

Purpose

Specifies continuous buffering mode.

Prototype

C/C++

DASErr far pascal K_SetContRun (DWORD *hFrame*);

Turbo Pascal

Function K_SetContRun (*hFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetContRun (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETCONTRUN% ALIAS "K_SetContRun"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function sets the buffering mode to continuous mode and sets the Buffering Mode element in the frame accordingly.

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

For more information on buffering modes, refer to the following pages:

Analog input operations	page 2-24
Analog output operations	page 2-38
Digital I/O operations	page 2-51

See Also K_ClrContRun

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetContRun (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetContRun (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetContRun (hAD)
```

BASIC

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

Boards Supported

All

Purpose

Sets up an external digital trigger.

Prototype

C/C++

DASErr far pascal K_SetDITrig (DWORD *hFrame*, short *nOpt*, short *nChan*, DWORD *nPattern*);

Turbo Pascal

Function K_SetDITrig (*hFrame* : Longint; *nOpt* : Word; *nChan* : Word; *nPattern* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetDITrig (*hFrame* : Longint; *nOpt* : Word; *nChan* : Word; *nPattern* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_SetDITrig Lib "DASSHELL.DLL"
(ByVal *hFrame* As Long, ByVal *nOpt* As Integer,
ByVal *nChan* As Integer, ByVal *nPattern* As Long) As Integer

BASIC

DECLARE FUNCTION KSETDITRIG% ALIAS "K_SetDITrig"
(BYVAL *hFrame* AS LONG, BYVAL *nOpt* AS INTEGER,
BYVAL *nChan* AS INTEGER, BYVAL *nPattern* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

nOpt — Trigger polarity and sensitivity.
Valid values:

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

nChan — Digital input channel.
Valid value: **0**

nPattern — Trigger pattern.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the digital trigger polarity and sensitivity in *nOpt*.

Since the DAS-1600/1400/1200 Series Function Call Driver does not currently support digital pattern triggering, the value of *nPattern* is meaningless. Since the external digital trigger must be connected to IP1/XTRIG pin (6), the value of *nChan* is meaningless. The *nPattern* and *nChan* parameters are provided for future compatibility.

The *nOpt* variable sets the value of the Trigger Polarity and Trigger Sensitivity elements.

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

See Also

K_SetTrig

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetDITrig (hAD, 0, 0, 0);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetDITrig (hAD, 0, 0, 0);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetDITrig (hAD, 0, 0, 0)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetDITrig% (hAD, 0, 0, 0)
```

K_SetDMABuf

Boards Supported

All

Purpose

Specifies the starting address of a previously allocated buffer and the number of samples in the buffer.

Prototype

C/C++

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

Turbo Pascal

Function K_SetDMABuf (*hFrame* : Longint; *pBuf* : Pointer; *dwSamples* : Longint) : Word;

Turbo Pascal for Windows

Function K_SetDMABuf (*hFrame* : Longint; *pBuf* : Pointer; *dwSamples* : Longint) : Word; far; external 'DASSHELL';

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

BASIC

DECLARE FUNCTION KSETDMABUF% ALIAS "K_SetDMABuf" (BYVAL *hFrame* AS LONG, BYVAL *pBuf* AS LONG, BYVAL *dwSamples* AS LONG)

Parameters

hFrame	Handle to the frame that defines the DMA-mode analog input operation.
pBuf	Starting address of buffer.
dwSamples	Number of samples. Valid values: **1** to **32768**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

The *pBuf* variable contains the value of the Buffer element.

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

See Also

K_DMAAlloc, KMakeDMABuf

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
TYPE
BufType = Array [0..1] of Integer;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_DMAAlloc (hAD, 1000, Addr (pBuf), hMem);
wDasErr := K_SetDMABuf (hAD, pBuf, 1000);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
TYPE
BufType = Array [0..1] of Integer;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_DMAAlloc (hAD, 1000, Addr (pBuf), hMem);
wDasErr := K_SetDMABuf (hAD, pBuf, 1000);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
Global pBuf As Long
...
wDasErr = K_DMAAlloc (hAD, 1000, pBuf, hMem)
wDasErr = K_SetDMABuf (hAD, pBuf, 1000)
```

BASIC

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

Boards Supported

DAS-1601, DAS-1602, DAS1401, DAS-1402

Purpose

Sets the gain.

Prototype

C/C++

DASErr far pascal K_SetG (DWORD *hFrame*, short *nGain*);

Turbo Pascal

Function K_SetG (*hFrame* : Longint; *nGain* : Word) : Word;

Turbo Pascal for Windows

Function K_SetG (*hFrame* : Longint; *nGain* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETG% ALIAS "K_SetG"
(BYVAL *hFrame* AS LONG, BYVAL *nGain* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.

nGain Gain code.
Valid values:

Board	Gain	Gain Code
DAS-1601 DAS-1401	1	0
	10	1
	100	2
	500	3
DAS-1602 DAS-1402	1	0
	2	1
	4	2
	8	3

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

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

The *nGain* variable sets the Gain element.

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

Gain codes do not apply to DAS-1200 Series boards.

See Also K_SetStartStopG

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetG (hAD, 1);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetG (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetG% (hAD, 1)
```

K_SetSSH

Boards Supported

All

Purpose

Enables and disables SSH mode.

Prototype

C/C++
DASErr far pascal K_SetSSH (DWORD *hFrame*, WORD *nMode*);

Turbo Pascal
Function K_SetSSH (*hFrame* : Longint; *nMode* : Word) : Word;

Turbo Pascal for Windows
Function K_SetSSH (*hFrame* : Longint; *nMode* : Word) : Word;
far; external 'DASSHELL';

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

BASIC
DECLARE FUNCTION KSETSSH% ALIAS "K_SetSSH"
(BYVAL *hFrame* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nMode	Code that indicates the status of SSH mode. Valid values: **0** for Disabled **1** for Enabled

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function stores the code that indicates the status of SSH mode in *nMode*.

K_GetADFrame and **K_ClearFrame** also disable SSH mode.

Refer to page 2-22 for information on SSH mode.

See Also K_SetADFreeRun, K_ClrADFreeRun

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetSSH (hAD, 1);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetSSH (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetSSH% (hAD, 1)
```

K_SetStartStopChn

Boards Supported

All

Purpose

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

Prototype

C/C++

DASErr far pascal K_SetStartStopChn (DWORD *hFrame*, short *nStart*, short *nStop*);

Turbo Pascal

Function K_SetStartStopChn (*hFrame* : Longint; *nStart* : Word; *nStop* : Word) : Word;

Turbo Pascal for Windows

Function K_SetStartStopChn (*hFrame* : Longint; *nStart* : Word; *nStop* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_SetStartStopChn Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *nStart* As Integer, ByVal *nStop* As Integer) As Integer

BASIC

DECLARE FUNCTION KSETSTARTSTOPCHN% ALIAS "K_SetStartStopChn" (BYVAL *hFrame* AS LONG, BYVAL *nStart* AS INTEGER, BYVAL *nStop* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels. Valid values: **0** to **255** (analog input) **0** or **1** (analog output)
nStop	Last channel in a group of consecutive channels. Valid values: **0** to **255** (analog input) **0** or **1** (analog output)

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the first channel in a group of consecutive channels in *nStart* and the last channel in the group of consecutive channels in *nStop*.

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

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

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

See Also

K_SetStartStopG

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetStartStopChn (hAD, 0, 7);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetStartStopChn (hAD, 0, 7);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetStartStopChn% (hAD, 0, 7)
```

K_SetStartStopG

Boards Supported

DAS-1601, DAS-1602, DAS-1401, DAS-1402

Purpose

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

Prototype

C/C++

DASErr far pascal K_SetStartStopG (DWORD *hFrame*, short *nStart*, short *nStop*, short *nGain*);

Turbo Pascal

Function K_SetStartStopG (*hFrame* : Longint; *nStart* : Word; *nStop* : Word; *nGain* : Word) : Word;

Turbo Pascal for Windows

Function K_SetStartStopG (*hFrame* : Longint; *nStart* : Word; *nStop* : Word; *nGain* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_SetStartStopG Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *nStart* As Integer, ByVal *nStop* As Integer, ByVal *nGain* As Integer) As Integer

BASIC

DECLARE FUNCTION KSETSTARTSTOPG% ALIAS "K_SetStartStopG" (BYVAL *hFrame* AS LONG, BYVAL *nStart* AS INTEGER, BYVAL *nStop* AS INTEGER, BYVAL *nGain* AS INTEGER)

Parameters

hFrame — Handle to the frame that defines the operation.

nStart — First channel in a group of consecutive channels.
Valid values: **0** to **255**

nStop — Last channel in a group of consecutive channels.
Valid values: **0** to **255**

nGain — Gain code.
Valid values

Board	Gain	Gain Code
DAS-1601 DAS-1401	1	0
	10	1
	100	2
	500	3
DAS-1602 DAS-1402	1	0
	2	1
	4	2
	8	3

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the first channel in a group of consecutive channels in *nStart*, the last channel in a group of consecutive channels in *nStop*, and the gain code for all channels in the group in *nGain*.

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

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

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

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

Gain codes do not apply to DAS-1200 Series boards.

See Also K_SetChn, K_SetStartStopChn

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetStartStopG (hAD, 0, 7, 0);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetStartStopG (hAD, 0, 7, 0);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetStartStopG% (hAD, 0, 7, 0)
```

Boards Supported

All

Purpose

Specifies the trigger source.

Prototype

C/C++

DASErr far pascal K_SetTrig (DWORD *hFrame*, short *nMode*);

Turbo Pascal

Function K_SetTrig (*hFrame* : Longint; *nMode* : Word) : Word;

Turbo Pascal for Windows

Function K_SetTrig (*hFrame* : Longint; *nMode* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETTRIG% ALIAS "K_SetTrig"
(BYVAL *hFrame* AS LONG, BYVAL *nMode* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nMode	Trigger source. Valid values: **0** for Internal trigger **1** for External trigger

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the trigger source in *nMode*.

An internal trigger is a software trigger; the trigger event occurs when the operation is started. Note that there is a slight delay between when the operation is started and when the trigger event occurs. An external trigger

is either an analog trigger or a digital trigger. Refer to page 2-25 for more information about internal and external trigger sources.

If *nMode* = **1**, an external digital trigger (positive edge on the IP1/XTRIG pin (6)) is assumed. Use **K_SetDITrig** to change the conditions of the digital trigger. Use **K_SetADTrig** to specify the conditions for an external analog trigger.

K_GetADFrame and **K_ClearFrame** set the trigger source to internal.

See Also K_SetADTrig, K_SetDITrig

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetTrig (hAD, 1);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetTrig (hAD, 1);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetTrig% (hAD, 1)
```

Boards Supported

All

Purpose

Specifies the hysteresis value.

Prototype

C/C++

DASErr far pascal K_SetTrigHyst (DWORD *hFrame*, short *nHyst*);

Turbo Pascal

Function K_SetTrigHyst (*hFrame* : Longint; *nHyst* : Word) : Word;

Turbo Pascal for Windows

Function K_SetTrigHyst (*hFrame* : Longint; *nHyst* : Word) : Word;
far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETTRIGHYST% ALIAS "K_SetTrigHyst"
(BYVAL *hFrame* AS LONG, BYVAL *nHyst* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nHyst	Hysteresis value, specified in counts. Valid values: **0** to **4095**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the hysteresis value used for an analog trigger in *nHyst*.

You specify the hysteresis value in counts; refer to Appendix B for information on converting the hysteresis voltage to a count.

The *nHyst* variable sets the Trigger Hysteresis element.

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

Refer to page 2-26 for more information about analog triggers.

See Also K_SetADTrig

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
wDasErr = K_SetTrigHyst (hAD, 50);
```

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SetTrigHyst (hAD, 50);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetTrigHyst (hAD, 50);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SetTrigHyst (hAD, 50)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSetTrigHyst% (hAD, 50)
```

Boards Supported

All

Purpose

Starts a synchronous-mode operation.

Prototype

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

Turbo Pascal
Function K_SyncStart (*hFrame* : Longint) : Word;

Turbo Pascal for Windows
Function K_SyncStart (*hFrame* : Longint) : Word;
far; external 'DASSHELL';

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

BASIC
DECLARE FUNCTION KSYNCSTART% ALIAS "K_SyncStart"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

Refer to the following pages for an illustration of the programming tasks associated with synchronous-mode operations:

Analog input	page 1-8
Analog output	page 1-14
Digital input	page 1-18
Digital output	page 1-21

K_SyncStart (cont.)

See Also K_IntStart, K_DMAStart

Usage

C/C++

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

Turbo Pascal

```
uses D1600TP7;
...
wDasErr := K_SyncStart (hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SyncStart (hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS or DASDEC32.BAS to your project)

```
...
wDasErr = K_SyncStart (hAD)
```

BASIC

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

A

Error/Status Codes

Table A-1 lists the error/status codes that are returned by the DAS-1600/1400/1200 Series Function Call Driver, possible causes for error conditions, and possible solutions for resolving error conditions.

If you cannot resolve an error condition, contact Keithley MetraByte (508-880-3000) for technical support.

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.
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 user's guide for legal interrupt levels.

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

Error Code		Cause	Solution
Hex	Decimal		
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 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 user's guide or to the description of **K_SetStartStopChn** in Chapter 4 for legal channel numbers.
6006	24582	**Illegal gain code:** The specified analog I/O channel gain code is illegal for this board.	Specify a legal gain code. Refer to the user's guide or to the description of **K_SetG** in Chapter 4 for a list of legal gain codes.
6007	24583	**Illegal DMA address:** An FCD function specified a buffer address that is not suitable for a DMA operation for the number of samples required.	Use the **K_DMAAlloc** function to allocate dynamic buffers for DMA operations. In Windows, make sure that the Keithley Memory Manager is installed; refer to Appendix D of the user's guide for information.
6008	24584	**Illegal number in configuration file:** The configuration file contains one or more numeric values that are illegal.	Use the configuration utility to check and then change the configuration file.
600A	24586	**Configuration file not found:** The driver cannot find the configuration file specified as an argument to the driver initialization function.	Check that the file exists at the specified path. Check that the file name is spelled correctly in the driver initialization function parameter list.
600B	24587	**Error returning DMA buffer:** DOS returned an error in INT 21H function 49H during the execution of **K_DMAFree**.	Check that the memory handle passed as an argument to **K_DMAFree** was previously obtained using **K_DMAAlloc**.
600C	24588	**Error returning interrupt buffer:** The memory handle specified in **K_IntFree** is invalid.	Check the memory handle stored by **K_IntAlloc** and make sure that it was not modified.

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

Error Code		Cause	Solution
Hex	Decimal		
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.	The Function Call Driver supports a maximum of 30 frames of all types. Use **K_FreeFrame** to free a frame that the program is no longer using.
600F	24591	**Requested buffer size too large:** The requested buffer cannot be dynamically allocated because of its size.	Specify a smaller buffer size; refer to the description of **K_IntAlloc** in Chapter 4 for the legal range. If the Keithley Memory Manager is installed, use KMMSETUP.EXE to increase the reserved buffer heap size.
6010	24592	**Cannot allocate interrupt buffer:** (Windows-based languages only) **K_IntAlloc** failed because there was not enough available DOS memory.	Remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
6012	24594	**Interrupt buffer deallocation error:** (Windows-based languages only) An error occurred when **K_IntFree** attempted to free a memory handle.	Make sure that the memory handle passed as an argument to **K_IntFree** was previously obtained using **K_IntAlloc**.
6015	24597	**DMA Buffer too large:** The number of samples specified in **K_DMAAlloc** is too large.	Refer to the description of **K_DMAAlloc** in Chapter 4 for the buffer size range.
6016	24598	**VDS - Region not contiguous:** An error occurred while using Windows Virtual DMA Services. You tried to use **K_DMAAlloc** and the Keithley Memory Manager was not installed.	Refer to the user's guide for information on how to install and set up the Keithley Memory Manager.
6017	24599	**VDS - DMA wraparound:** See error 6016.	See error 6016.

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

Error Code		Cause	Solution
Hex	Decimal		
6018	24600	**VDS - Unable to lock region:** See error 6016.	See error 6016.
6019	24601	**VDS - No buffer available:** See error 6016.	See error 6016.
601A	24602	**VDS - Region too large:** See error 6016.	See error 6016.
601B	24603	**VDS - Buffer in use:** See error 6016.	See error 6016.
601C	24604	**VDS - Illegal region:** See error 6016.	See error 6016.
601D	24605	**VDS - Region not locked:** See error 6016.	See error 6016.
601E	24606	**VDS - Illegal page:** See error 6016.	See error 6016.
601F	24607	**VDS - Illegal buffer:** See error 6016.	See error 6016.
6020	24608	**VDS - Copy out of range:** See error 6016.	See error 6016.
6021	24609	**VDS - Illegal DMA channel:** See error 6016.	See error 6016.
6022	24610	**VDS - Count overflow:** See error 6016.	See error 6016.
6023	24611	**VDS - Count underflow:** See error 6016.	See error 6016.
6024	24612	**VDS - Function not supported:** See error 6016.	See error 6016.
6025	24613	**Illegal OBM mode:** The mode number specified in **K_SetOBMMode** is illegal.	Specify a legal mode value.

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

Error Code		Cause	Solution
Hex	Decimal		
6026	24614	**Illegal DMA structure:** An error occurred during the execution of **K_DMAFree**.	Try using **K_DMAFree** again. If the error continues, contact Keithley MetraByte for technical support.
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.EXE 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**. Refer to the description of **K_DMAAlloc** in Chapter 4 for legal values.
602D	24621	**Illegal device handle:** A bad device handle was passed to a function such as **K_GetADFrame**. The handle used was not initialized through a call to **K_GetDevHandle** or **DAS1600_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.

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

Error Code		Cause	Solution
Hex	**Decimal**		
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. Install and configure the Keithley Memory Manager.
6031	24625	**Illegal memory block handle:** A bad memory handle was passed to **K_IntFree** or **K_DMAFree**. The handle used was not initialized through a call to **K_IntAlloc** or **K_DMAAlloc**, or it was corrupted by you program.	Restart your program and monitor the memory handle value(s).
6032	24626	**Out of memory handles:** An attempt to allocate a memory block using **K_IntAlloc** or **K_DMAAlloc** failed because the maximum number of handles has already been assigned.	Use **K_IntFree** or **K_DMAFree** to free previously allocated memory blocks before allocating again.
6034	24628	**Memory corrupted:** Int 21H function 48H, used to allocate a memory block from the DOS far heap, returned the DOS error 7; this means that memory is corrupted. It is likely that you stored data (through a DMA-mode or interrupt-mode operation) into an illegal area of DOS memory.	Recheck the parameters set by **K_DMAAlloc** and **K_SetDMABuf**. If a fatal system error, restart your computer.
6035	24629	**Driver in use:** You attempted to initialize a driver that was already initialized by a call to **K_OpenDriver**. (This can occur since, under Windows, it is possible to open the same driver from multiple programs that are running simultaneously.)	To continue using the driver with the same configuration, pass a null string as the second argument to **K_OpenDriver**. To use the driver with a different configuration, close any programs currently accessing the driver, and then open the driver again (using **K_OpenDriver**).

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

Error Code		Cause	Solution
Hex	Decimal		
6036	24630	**Illegal driver handle:** The specified driver handle is not valid.	Someone may have closed the driver; if so, use **K_OpenDriver** to reopen the driver with the desired driver handle. Try again using another driver handle.
6037	24631	**Driver not found:** The specified driver cannot be found.	Check your link statement to make sure the specified driver is included. Make sure that the device name string is entered correctly in **K_OpenDriver**.
6038	24632	**Invalid source pointer:** (Windows-based languages only) The pointer to the source buffer that you passed as an argument to **K_MoveBufToArray** is invalid for the specified count. (The source pointer, when added to the number of samples, exceeds the programmed addressing range of that pointer.)	Check the pointer to the source buffer and the number of samples to transfer that you specified in **K_MoveBufToArray**.
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.

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

Error Code		Cause	Solution
Hex	Decimal		
603B	24635	**Error freeing buffer selector: K_DMAFree** or **K_IntFree** failed because one or more of the selectors that reference the memory buffer could not be freed.	Check that the memory buffer being freed was previously obtained through **K_DMAAlloc** or **K_IntAlloc.**
603C	24636	**Error allocating buffer selector: K_DMAAlloc** or **K_IntAlloc** failed because a selector could not be allocated from Window's Local Descriptor Table.	Close all programs and restart Windows. If the error continues, contact Keithley MetraByte for technical support.
603D	24637	**Error allocating memory buffer: K_DMAAlloc** or **K_IntAlloc** failed because a necessary internal buffer could not be allocated to complete the operation. You attempted to specify the starting address of a locally dimensioned array in Windows 95.	Close all programs and restart Windows. In Windows 95, make sure that you use **K_IntAlloc** or **K_DMAAlloc** to dynamically allocate a memory buffer and make sure that you use **K_SetBuf** or **K_BufListAdd** to specify the starting address of the dynamically allocated memory buffer. If the error continues, contact Keithley MetraByte for technical support.
7000	28672	**No board name:** The driver initialization function did not find a board name in the specified configuration file.	Specify a legal board name in the configuration file.
7001	28673	**Illegal board name:** The board name in the specified configuration file is illegal.	Specify a legal board name in the configuration file.
7002	28674	**Illegal board number:** The driver initialization function found an illegal board number in the specified configuration file.	Specify a legal board number: 0 or 1.
7003	28675	**Illegal base address:** The driver initialization function found an illegal base address in the specified configuration file.	Specify a base address in the inclusive range &H200 (512) to &H3F0 (1008) in increments of 10h (16). Make sure that &H precedes hexadecimal numbers.

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

Error Code		Cause	Solution
Hex	Decimal		
7004	28676	**Illegal DMA channel:** The driver initialization function found an illegal DMA channel in the specified configuration file.	Specify a legal DMA channel: 1 or 3.
7005	28677	**Illegal interrupt level:** The driver initialization function found an illegal interrupt level in the specified configuration file.	Specify a legal interrupt level: 2 through 7.
7006	28678	**Illegal number of EXPs:** The driver initialization function found an illegal number of expansion accessories in the specified configuration file.	Specify a legal number of expansion accessories: 0 through 8.
7007	28679	**Illegal clock select:** The driver initialization function found an illegal clock specification in the specified configuration file.	Specify a legal clock: 1 MHz or 10 MHz.
7008	28680	**Illegal wait state:** The driver initialization function found an illegal wait state specification in the specified configuration file.	Specify a legal condition for wait state: yes or no.
7009	28681	**Illegal ADC channel mode:** The driver initialization function found an illegal input range type in the specified configuration file.	Specify a legal input range type: bipolar or unipolar.
700A	28682	**Illegal ADC channel configuration:** The driver initialization function found an illegal input configuration in the specified configuration file.	Specify a legal input configuration: single-ended or differential.
700B	28683	**Illegal DAC0 mode:** The driver initialization function found an illegal D/A mode in the specified configuration file.	Specify a legal D/A mode: bipolar or unipolar.

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

Error Code		Cause	Solution
Hex	**Decimal**		
700C	28684	**Illegal DAC1 mode:** The driver initialization function found an illegal D/A mode in the specified configuration file.	Specify a legal D/A mode: bipolar or unipolar.
700D	28685	**Illegal DAC0 ref:** The driver initialization function found an illegal reference voltage in the specified configuration file.	Specify a legal D/A reference voltage: 5 V, 10 V, or user-defined.
700E	28686	**Illegal DAC1 ref:** The driver initialization function found an illegal reference voltage in the specified configuration file.	Specify a legal D/A reference voltage: 5 V, 10 V, or user-defined.
700F	28687	**Illegal port A:** The driver initialization function found an illegal digital I/O configuration in the specified configuration file.	Specify a legal digital I/O configuration: input or output.
7010	28688	**Illegal port B:** The driver initialization function found an illegal digital I/O configuration in the specified configuration file.	Specify a legal digital I/O configuration: input or output.
7011	28689	**Illegal port C low:** The driver initialization function found an illegal digital I/O configuration in the specified configuration file.	Specify a legal digital I/O configuration: input or output.
7012	28690	**Illegal port C high:** The driver initialization function found an illegal digital I/O configuration in the specified configuration file.	Specify a legal digital I/O configuration: input or output.
7013	28691	**Illegal EXP-16 number:** The driver initialization function found an illegal number of EXP-16s in the specified configuration file.	Specify a legal number of EXP-16 accessories: 0 through 8.

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

Error Code		Cause	Solution
Hex	**Decimal**		
7014	28692	**Illegal EXP-16 gain:** The driver initialization function found an illegal gain value in the specified configuration file.	Specify a legal gain value for the EXP-16 accessories.
7015	28693	**Illegal number of EXPGP**: The driver initialization function found an illegal number of EXP-GPs in the specified configuration file.	Specify a legal number of EXP-GP accessories: 0 through 8.
7016	28694	**Illegal EXPGP number:** The driver initialization function found an illegal number assigned to one of the EXP-GPs in the specified configuration file.	Specify a legal EXP-GP number: 0 through 7.
7017	28695	**Illegal EXPGP gain:** The driver initialization function found an illegal EXP-GP gain value in the specified configuration file.	Specify a legal gain value for the EXP-GP accessories: X1 or X2.5.
7018	28696	**Illegal EXPGP Chan:** The driver initialization function found an illegal gain assigned to one of the channels on one of the EXP-GPs in the specified configuration file.	Specify a legal gain for each EXP-GP channel: 1, 10, 100, 1000 (X1) or 2.5, 25, 250, 2500 (X2.5)
7019	28697	**Illegal CJR:** The driver initialization function found an illegal channel assigned to the cold-junction reference (CJR) value in the specified configuration file.	Specify a legal CJR channel: 1 through 7, –1 (unused)
701A	28698	**Illegal rev. number:** The revision of the driver you are using does not match the revision of the Keithley DAS Driver Specification.	Make sure that you are using the appropriate driver.

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

Error Code		Cause	Solution
Hex	Decimal		
701B	28699	**Resource busy:** The program attempted to start an operation while a similar operation was in progress.	An attempt was made to execute interrupt and DMA operations simultaneously. Only one of these operations is allowed at one time. Use **K_IntStop** or **K_DMAStop** to stop the in-progress operation before initiating the second operation.
701C	28700	**Unknown error number:** A request for an undefined message string was made.	Check the error number specified in **K_GetErrMsg** and try again.
701D	28701	**Channel Gain Array Not Supported:** A DMA operation was attempted with a channel-gain queue.	Use interrupt mode or synchronous mode. Use a single channel or a group of consecutive channels.
701E	28702	**DMA not supported on EXP Channels:** A DMA operation was attempted on an EXP board.	Use interrupt mode or synchronous mode.
701F	28703	**Incorrect A/D Uni/Bip switch setting:** The unipolar/bipolar switch on the board does not match the setting in the configuration file.	Make sure that both settings match. Use the CFG1600.EXE utility to modify the configuration file or change the switch setting on the board.
7021	28705	**Incorrect A/D 16/8 channel switch setting:** The single-ended/ differential switch on the board does not match the setting in the configuration file.	Make sure that both settings match. Use the CFG1600.EXE utility to modify the configuration file or change the switch setting on the board.
7022	28706	**Illegal settling time:** An invalid burst mode conversion rate count is set.	Make sure that the count value specified in **K_SetBurstTicks** is in the range of 2 to 255.
7023	28707	**Illegal number of SSH:** The number of SSH accessories in the configuration file is not valid.	Check the number of SSH accessories specified in the configuration file; should be 1 to 4 (for SSH-4/A) or 1 to 2 (for SSH-8).

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

Error Code		Cause	Solution
Hex	**Decimal**		
7024	28708	**Illegal SSH chan:** The number of SSH channels in the configuration file is not valid.	Check the number of channels specified in the configuration file; should be 0 to 3 (for SSH-4/A) or 0 to 7 (for SSH-8).
7025	28709	**Illegal SSH gain:** The SSH-8 or SSH-4/A channel gain in the configuration file is not valid.	Check the gain selection in the configuration file.
7026	28710	**Illegal SSH4 mode:** The SSH-4/A mode in the configuration file is not valid.	Check the mode in the configuration file; should be master or slave.
7027	28711	**Illegal SSH timing:** The SSH mode in the configuration file is not valid.	Check the SSH timing selection in the configuration file; should be internal or external.
7029	28713	**Illegal SSH type:** The SSH type in the configuration file is not valid.	Check the SSH type in the configuration file; must be SSH4A or SSH8.
702A	28714	**Illegal SSH pacer:** The SSH clock selection in the configuration file is not valid (for SSH-8 only).	Check the clock selection in the configuration file; must be 10 MHz.
702B	28715	**Illegal Start/Stop Chan in Diff Mode:** In differential mode, the start channel cannot be greater than the stop channel.	Change the start channel and stop channel numbers.
702C	28716	**Illegal EXP-1600 number:** The driver initialization function found an illegal number of EXP-1600s in the specified configuration file.	Specify a legal number of EXP-1600 accessories: 0 through 16.
702D	28717	**Illegal EXP-1600 gain:** The driver initialization function found an illegal gain value in the specified configuration file.	Specify a legal gain value for the EXP-1600 accessories: 0.5, 1, 5, 10, 50, 100, 250, 500

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

Error Code		Cause	Solution
Hex	Decimal		
702E	28718	**Burst mode is supported for DMA only:** You attempted to use burst or burst with SSH conversion mode for a synchronous-mode or interrupt-mode operation.	Use **K_ClrADFreeRun** to set the conversion mode to paced mode. Use **K_DMAStart** to start your operation in DMA mode.
702F	28719	**Incompatible board detected:** You attempted to use the DAS-1600/1400/1200 Series Function Call Driver or the DAS-1600/1400/1200 Series Control Panel with a board not manufactured by Keithley MetraByte.	The DAS-1600/1400/1200 Series Function Call Driver and the DAS-1600/1400/1200 Series Control Panel are intended for use with Keithley MetraByte boards only. Contact Keithley MetraByte (508-880-3000) or your local sales office for information on supported boards.
8001	32769	**Function not supported:** You have attempted to use a function not supported by the Function Call Driver.	Contact Keithley MetraByte for technical support.
8003	32771	**Illegal board number:** An illegal board number was specified in the board initialization function.	Refer to the description of **K_GetDevHandle** or **DAS1600_GetDevHandle** in Chapter 4 for legal board numbers.
8004	32772	**Illegal error number:** The error message number specified in **K_GetErrMsg** is invalid.	The error number must be one the error numbers listed in this appendix.
8005	32773	**Board not found at configured address:** The 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.

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

Error Code		Cause	Solution
Hex	Decimal		
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 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.
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.

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

Error Code		Cause	Solution
Hex	**Decimal**		
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.
8021	32801	**Illegal clock sync mode:** The two operations you are trying to synchronize cannot be synchronized on your board.	Check the synchronizing clock source that you specified in **K_SetSync**.
FFFF	65535	**User aborted operation:** You pressed **Ctrl+Break** during a synchronous-mode operation or while waiting for an analog trigger event to occur.	Start the operation again, if desired.

B

Data Formats

This appendix contains the following sections:

- **Converting Voltage to Counts** - instructions for converting a voltage value to a count value that the DAS-Scan Function Call Driver can understand.
- **Converting Counts to Voltage** - instructions for converting a count value returned by the DAS-Scan Function Call Driver to a voltage value.

Converting Voltage to Counts

When specifying an analog trigger level (as in **K_SetADTrig**), a hysteresis value (as in **K_SetTrigHyst**), or an analog output value (as in **K_DAWrite**), you must convert the voltage to a count value that the DAS-1600/1400/1200 Series Function Call Driver can understand. The following sections describe how to convert voltage to counts for each of these situations.

Note: The DAS-1600/1400/1200 Series Function Call Driver provides the **K_GetADMode** function, which gets the analog input range type (bipolar or unipolar). You may find this function useful when converting counts to voltage.

Specifying a Trigger Level

To convert a voltage value to a count when specifying an analog trigger level, use the equation that is appropriate for your A/D mode, substituting the desired voltage for V_{trig}.

Bipolar (DAS-1600/1400 Series)

$$\text{Count} = \frac{V_{trig} \times 4096}{20} + 2048$$

Bipolar (DAS-1200 Series)

$$\text{Count} = \frac{V_{trig} \times 4096}{10} + 2048$$

Unipolar (DAS-1600/1400 Series only)

$$\text{Count} = \frac{V_{trig} \times 4096}{10}$$

For example, assume that you want to specify an analog trigger level of 2.5 V for a channel on a DAS-1601 board configured for a bipolar input range. The count is determined as follows:

$$\frac{2.5 \times 4096}{20} + 2048 = 2560$$

Specifying a Hysteresis Value

To convert a voltage value to a count when specifying a hysteresis value, use the equation that is appropriate for your A/D mode, substituting the desired voltage for V_{hyst}.

DAS-1600/1400 Series

$$\text{Count} = \frac{V_{hyst} \times 4096}{20}$$

DAS-1200 Series

$$\text{Count} = \frac{V_{hyst} \times 4096}{10}$$

For example, assume that you want to specify a hysteresis value of 0.05 V for a channel on a DAS-1601 board. The count is determined as follows:

$$\frac{0.05 \times 4096}{10} = 20$$

Specifying an Analog Output Value (DAS-1600 Series Only)

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 your D/A mode, substituting the desired voltage for V_{out}. Refer to Table B-1 for the appropriate *span* value.

   Bipolar

   $$\text{Count} = \frac{V_{out} \times 4096}{\text{span}} + 2048$$

   Unipolar

   $$\text{Count} = \frac{V_{out} \times 4096}{\text{span}}$$

Table B-1. Span Values for Analog Output Equations

Mode	Reference Voltage	Output Range	Span
Bipolar	–5	-5 V to 4.998 V	10
	–10	-10 V to 9.995 V	20
Unipolar	–5	0.0 V to 4.999 V	5
	–10	0.0 V to 9.998 V	10

For example, assume that you want to specify an analog output of 3 V for a DAS-1602 that is set up for a bipolar output with a –5 V reference. The count is determined as follows:

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

2. Next, pack the count into a variable, as follows:

variable data = (left-shift count four bits) bit-wise AND with FFF0

Converting Counts to Voltage

The DAS-Scan Function Call Driver can read count values only. When reading an analog input value (as in **K_ADRead**), you can convert the count value returned by the DAS-Scan Function Call Driver to a voltage value.

FCD functions return counts as left-justified values in the upper 12 bits of variables declared as integers. Perform the following steps to convert a count to an analog input voltage:

1. Use the following equation to unpack a count:

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

2. This method produces a count value that ranges from 0 through 4095. (Note that the lower four bits contain the channel number.)
3. Use the equation that corresponds to your A/D mode, substituting the *count* value arrived at in step 1 and the appropriate *span* value from Table B-2.

   Bipolar

   $$\text{Voltage} = \frac{(\text{count} - 2048) \times \text{span}}{4096}$$

   Unipolar

   $$\text{Voltage} = \frac{\text{count} \times \text{span}}{4096}$$

Table B-2. Span Values for A/D Conversion Equations

Board	A/D Mode	Gain	Input Range	Span
DAS-1601 DAS-1401	Unipolar	1	0.0 V to 10.0V	10 V
		10	0.0 V to 1.0 V	1 V
		100	0.0 V to 100 mV	0.1 V
		500	0.0 V to 20 mV	0.02 V
	Bipolar	1	–10 V to 9.995 V	20 V
		10	–1.0 V to 0.9995 V	2 V
		100	–100 mV to 99.95 mV	0.2 V
		500	–20 mV to 19.99 mV	0.04 V
DAS-1602 DAS-1402	Unipolar	1	0.0 V to 10 V	10.0 V
		2	0.0 V to 5 V	5.0 V
		4	0.0 V to 2.5 V	2.5 V
		8	0.0 V to 1.25 V	1.25 V
	Bipolar	1	–10 V to 9.995 V	20.0 V
		2	–5 V to 4.9976 V	10.0 V
		4	–2.5 V to 2.4988 V	5.0 V
		8	–1.25 V to 1.2494 V	2.5 V
DAS-1201	Bipolar	1	–5.0 V to 4.9976 V	10.0 V
		10	–0.5 V to 0.49976 V	1.0 V
		100	–0.05 V to 0.049976 V	0.1 V
		500	–0.01 V to 0.009995 V	0.02 V
DAS-1202	Bipolar	1	–5.0 V to 4.9976 V	10.0 V
		2	–2.5 V to 2.4988 V	5.0 V
		4	–1.25 V to 1.2494 V	2.5 V
		8	–0.625 V to 0.62469 V	1.250 V

For example, assume that you want to read analog input data from a channel on a DAS-1601 board configured for the unipolar input range and a gain of 1. The count value is 3072. The voltage is determined as follows:

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

As another example, assume that you want to read the analog input data from a channel on a DAS-1402 board configured for a bipolar input range and a gain of 2. The count value is 1024. The voltage is determined as follows:

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

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

V

W