DAS-800 Series
Function Call Driver
User's Guide

DAS-800 Series
Function Call Driver
User's Guide

Revision A - December 1993
Part Number: 86770

The information contained in this manual is believed to be accurate and reliable. However, the manufacturer assumes no responsibility for its use; nor for any infringements or 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 the manufacturer.

THE MANUFACTURER 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 THAT IS SUITED FOR USE IN LIFE SUPPORT OR CRITICAL APPLICATIONS.

Table of Contents

Preface

1 Getting Started

Installing the Software 1-2
Installing the DAS-800 Series Standard Software Package . . 1-2
Installing the ASO-800 Software Package 1-3
DOS Installation 1-3
Windows Installation 1-4
Setting Up the Boards 1-5
Getting Help 1-6

2 Available Operations

Analog Input Operations 2-1
Operation Modes 2-2
Memory Allocation and Management 2-3
Input Range Type 2-5
Gains 2-5
Channels 2-6
Single Channel 2-8
Multiple Channels Using a Group of Consecutive Channels 2-9
Multiple Channels Using a Channel-Gain List 2-9
Conversion Clocks 2-13
Buffering Mode 2-16
Triggers 2-16
Analog Triggers 2-17
Digital Triggers 2-20
Hardware Gates 2-22
Digital I/O Operations 2-24
Counter/Timer I/O Operations 2-26
System Operations 2-27
Initializing the Driver 2-28
Initializing a Board 2-29
Retrieving the Revision Level 2-30
Handling Errors 2-30

3 Programming with the Function Call Driver

How the Driver Works 3-1
Programming Overview 3-5
Preliminary Tasks 3-6
Operation-Specific Programming Tasks 3-6
Analog Input Operations 3-6
Single Mode 3-7
Synchronous Mode 3-7
Interrupt Mode 3-9
Digital I/O Operations 3-12
Language-Specific Programming Information 3-12
Microsoft C/C++ 3-13
Borland C/C++ 3-14
Microsoft QuickC for Windows 3-15
Microsoft Visual C++ 3-16
Borland Turbo Pascal 3-16
Borland Turbo Pascal for Windows 3-17
Specifying the Buffer Address (Pascal) 3-18
Specifying the Channel-Gain List Starting Address (Pascal) 3-19
Microsoft QuickBASIC (Version 4.0) 3-20
Microsoft QuickBasic (Version 4.5) 3-21
Microsoft Professional Basic (Version 7.0) 3-22
Microsoft Visual Basic for DOS 3-23
Microsoft Visual Basic for Windows 3-24
Specifying the Buffer Address (All BASIC Languages) . . . 3-25

4 Function Reference

DAS800_DevOpen 4-6
DAS800_GetADGainMode 4-9
DAS800_GetDevHandle 4-11
DAS800_Get8254 4-13
DAS800_SetADGainMode 4-15
DAS800_Set8254 4-17
K_ADRead 4-19
K_BufListAdd 4-22
K_BufListReset 4-24
K_ClearFrame 4-26
K_CloseDriver 4-28
K_ClrContRun 4-30
K_DASDevInit 4-32
K_DIRead 4-33

K_DOWrite 4-35
K_FormatChanGAry 4-37
K_FreeDevHandle 4-38
K_FreeFrame 4-39
K_GetADFrame 4-40
K_GetADTrig 4-42
K_GetBuf 4-44
K_GetChn 4-46
K_GetChnGAry 4-48
K_GetClk 4-50
K_GetClkRate 4-52
K_GetContRun 4-54
K_GetDevHandle 4-56
K_GetDITrig 4-58
K_GetErrMsg 4-60
K_GetG 4-61
K_GetGate 4-63
K_GetStartStopChn 4-65
K_GetStartStopG 4-67
K_GetTrig 4-70
K_GetTrigHyst 4-72
K_GetVer 4-74
K_InitFrame 4-76
K_IntAlloc 4-78
K_IntFree 4-80
K_IntStart 4-81
K_IntStatus 4-83
K_IntStop 4-86
K_MoveBufToArray 4-88
K_OpenDriver 4-89
K_RestoreChanGAry 4-92
K_SetADTrig 4-93
K_SetBuf 4-95
K_SetBufI 4-97
K_SetChn 4-99
K_SetChnGAry 4-101
K_SetClk 4-103
K_SetClkRate 4-105
K_SetContRun 4-107
K_SetDITrig 4-109
K_SetG 4-111
K_SetGate 4-113

K_SetStartStopChn 4-115
K_SetStartStopG 4-117
K_SetTrig 4-120
K_SetTrigHyst 4-122
K_SyncStart 4-124

A Error/Status Codes

B Data Formats

Converting Raw Counts to Voltage B-2
Converting Voltage to Raw Counts B-3
Specifying an Analog Trigger Level B-3
Specifying a Hysteresis Value B-5

Index

List of Figures

Figure 2-1. Analog Input Channels 2-8
Figure 2-2. Channel-Gain List (C or Pascal) 2-10
Figure 2-3. Sample Channel-Gain List (C or Pascal) 2-11
Figure 2-4. Channel-Gain List (BASIC) 2-12
Figure 2-5. Sample Channel-Gain List (BASIC) 2-12
Figure 2-6. Initiating Conversions 2-15
Figure 2-7. Analog Trigger Conditions 2-17
Figure 2-8. Using a Hysteresis Value 2-19
Figure 2-9. Initiating Conversions with an External Analog Trigger 2-20
Figure 2-10. Initiating Conversions with an External Digital Trigger 2-21
Figure 2-11. Hardware Gate 2-23
Figure 2-12. Digital Input Bits 2-24
Figure 2-13. Digital Output Bits 2-25

List of Tables

Table 2-1. Supported Operations 2-1
Table 2-2. Analog Input Ranges 2-6
Table 2-3. Channels in Maximum Configuration 2-7
Table 3-1. A/D Frame Elements 3-3
Table 3-2. Setup Functions for Synchronous-Mode Operations 3-7
Table 3-3. Setup Functions for Interrupt-Mode Operations . 3-10
Table 4-1. FCD Functions. 4-2
Table 4-2. Default Configuration 4-7
Table A-1. Error/Status Codes. A-1

Preface

The *DAS-800 Series Function Call Driver User's Guide* describes how to write application programs for DAS-800 Series boards using the DAS-800 Series Function Call Driver. The DAS-800 Series Function Call Driver supports the following DOS-based languages:

- Microsoft® QuickBASIC (Version 4.0)
- Microsoft QuickBasic™ (Version 4.5 and higher)
- 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 6.0 and higher)

The DAS-800 Series Function Call Driver also supports the following Windows™-based languages:

- Microsoft Visual Basic for Windows (Version 2.0 and higher)
- Microsoft QuickC® for Windows (Version 1.0)
- Microsoft Visual C++™ (Version 1.0)
- Borland Turbo Pascal for Windows (Version 1.0 and higher)

The manual is intended for application programmers using a DAS-800, DAS-801, or DAS-802 board in an IBM® PC/XT™, AT® or compatible computer. It is assumed that users have read the *DAS-800 Series User's Guide* to familiarize themselves with the boards' functions, and that they have completed the appropriate hardware installation and configuration. It is also assumed that users are experienced in programming in their selected language and that they are familiar with data acquisition principles.

The *DAS-800 Series Function Call Driver User's Guide* is organized as follows:

- Chapter 1 contains the information needed to install the DAS-800 Series Function Call Driver and to set up DAS-800 Series boards.
- Chapter 2 contains the background information needed to use the functions included in the DAS-800 Series Function Call Driver.
- Chapter 3 contains programming guidelines and language-specific information related to using the DAS-800 Series Function Call Driver.
- Chapter 4 contains detailed descriptions of the DAS-800 Series Function Call Driver functions, arranged in alphabetical order.
- Appendix A contains a list of the error codes returned by DAS-800 Series Function Call Driver functions.
- Appendix B contains instructions for converting raw counts to voltage and for converting voltage to raw counts.

An index completes this manual.

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

- References to DAS-800 Series boards apply to the DAS-800, DAS-801, and DAS-802 boards. 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 (Version 4.0), Microsoft QuickBasic (Version 4.5), 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 enclosed in square brackets ([ ]).

1

Getting Started

The DAS-800 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-800 Series standard software package** - This is the software package that is shipped with DAS-800 Series boards; it includes the following:
  - Libraries of FCD functions for Microsoft QuickBASIC (Version 4.0), Microsoft QuickBasic (Version 4.5), Microsoft Professional Basic, and Microsoft Visual Basic for DOS.
  - Support files, containing such program elements as function prototypes and definitions of variable types, which are required by the FCD functions.
  - Utility programs, running under DOS, that allow you to configure, calibrate, and test the functions of DAS-800 Series boards.
  - Language-specific example programs.
- **ASO-800 software package** - This is the optional Advanced Software Option for DAS-800 Series boards. You purchase the ASO-800 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 Visual Basic for Windows, Microsoft QuickC for Windows, Microsoft Visual C++, 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-800 Series boards.
- Language-specific example programs.

This chapter contains the information needed to install the DAS-800 Series Function Call Driver in your computer and set up your DAS-800 Series boards. It also contains information on where to get help if you have problems installing or using the Function Call Driver.

Installing the Software

Before you can use the Function Call Driver, you must install the appropriate software package, either the DAS-800 Series standard software package or the ASO-800 software package.

The following sections describe how to install the DAS-800 Series standard software package and how to install the ASO-800 software package from both DOS and Windows.

Installing the DAS-800 Series Standard Software Package

To install the DAS-800 Series standard software package, perform the following steps:

1. Make a back-up copy of the supplied disks.
2. Insert disk #1 into the disk drive.

3. Assuming that you are using disk drive A, enter the following at the DOS prompt:

```
A:install
```

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

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

   The installation program expands any files that are stored in a compressed format and copies them into the directory you specified (DAS800 directory on hard disk C if you do not specify otherwise).

5. Review the following files:

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

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

Installing the ASO-800 Software Package

This section describes how to install the ASO-800 software package from both DOS and Windows.

DOS Installation

To install the ASO-800 software package from DOS, perform the following steps:

1. Make a back-up copy of the supplied disks.

2. Insert disk #1 into the disk drive.

3. Assuming that you are using disk drive A, enter the following at the DOS prompt:

```
A:install
```

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

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

   The installation program expands any files that are stored in a compressed format and copies them into the directory you specified (ASO800 directory on hard drive C if you do not specify otherwise).

5. Review the following files:

   - FILES.TXT lists and describes all the files copied to the hard disk by the installation program.
   - README.TXT contains information that was not available when this manual was printed.

Windows Installation

To install the ASO-800 software package from Windows, perform the following steps:

1. Make a back-up copy of the ASO-Windows disk.
2. Insert the ASO-Windows disk into the disk drive.
3. Start Windows.
4. From the Program Manager menu, choose File and then choose Run.
5. Assuming that you are using disk drive A, type the following at the command line in the Run dialog box, and then select OK:

   ```
   A:SETUP
   ```

   The installation program prompts you for your installation preferences, including the name of the directory you want to copy the software to.

6. Type the path name and select Continue.

The installation program expands any files that are stored in a compressed format and copies them into the directory you specified (ASO800\WINDOWS directory on hard drive C if you do not specify otherwise).

The installation program also creates a DAS-800 family group; this group includes example Windows programs and help files.

7. Review the following files:

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

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

Setting Up the Boards

Before you use the Function Call Driver, make sure that you have performed the following steps:

1. Installed the software.

   If not, install the appropriate software package (either the DAS-800 Series standard software package or the ASO-800 software package) on your IBM PC/XT, AT or compatible computer. Refer to page 1-2 for information on installing the DAS-800 Series standard software package; refer to page 1-3 for information on installing the ASO-800 software package.

2. Created a configuration file.

   If not, use the D800CFG.EXE utility to create a configuration file for the DAS-800 Series boards you are using. For each board, make sure that you specify the board model, the base address, the use of counter/timer 2 (C/T2) on the 8254 counter/timer circuitry, the input range type (unipolar or bipolar), the input configuration (single-ended or differential) for each channel on each DAS-801 and DAS-802 board, the interrupt level, and the expansion boards used. Refer to the *DAS-800 Series User's Guide* for more information.

3. Configured the hardware.

   If not, use switches on the boards to set the base address of each DAS-800 Series board and the input configuration (single-ended or differential) for each channel on each DAS-801 and DAS-802 board. Use the jumper on the boards to set the interrupt level of each DAS-800 Series board. Refer to the instructions in the D800CFG.EXE utility and the *DAS-800 Series User's Guide* for more information.

4. Installed the board(s).

   If not, with the computer powered down, install the DAS-800 Series boards in your computer. The DAS-800 requires a single, short slot; the DAS-801 and DAS-802 require a single, 1/2-slot. Refer to the documentation provided with your computer for more information on installing boards.

   Note: The DAS-800 Series Function Call Driver supports a maximum of four DAS-800 Series boards.

5. Tested the board(s), if desired.

   If you want to test the functions of the boards before writing your application program, use the CTL800.EXE utility (for DOS) or the CTL800W.EXE utility (for Windows). Refer to the *DAS-800 Series User's Guide* for more information.

Getting Help

If you need help installing or using the DAS-800 Series Function Call Driver, contact the factory.

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:

Software package	Version	_____
	Invoice/order #	_____
Compiler	Language	_____
	Manufacturer	_____
	Version	_____
Operating system	DOS version	_____
	Windows version	3.0 3.1 _____
	mode	Standard Enhanced
Computer	Manufacturer	_____
	CPU type	8088 286 386 486 ____
	Clock speed (MHz)	8 12 20 25 33 ____
	Math coprocessor	Yes No
	Amount of RAM	_____
	Video system	CGA Hercules EGA VGA
	BIOS type	_____
DAS-800 board	Model	800 801 802
	Serial #	_____
	Base address setting	_____
	Interrupt level setting	2 3 4 5 6 7 None
	Input configuration	Single-ended Differential
	Input range type	Unipolar Bipolar
	8254 C/T2 usage	Cascaded Normal
Expansion boards	Type	_____
	Type	_____
	Type	_____
	Type	_____
	Type	_____
	Type	_____
	Type	_____
	Type	_____

2

Available Operations

This chapter contains the background information you need to use the FCD functions to perform operations on DAS-800 Series boards. The supported operations are listed in Table 2-1.

Table 2-1. Supported Operations

Operation	Page Reference
Analog input	page 2-1
Digital input and output (I/O)	page 2-24
Counter/timer I/O	page 2-26
System	page 2-27

Analog Input Operations

This section describes the following:

- Analog input operation modes available.
- How to allocate and manage memory.
- How to modify the input range type.
- How to specify channels and gains, a conversion clock source, a buffering mode, and a trigger source for an analog input operation.

Operation Modes

The operation mode determines which attributes you can specify for an analog input operation and whether the operation is performed in the foreground or in the background. You can perform analog input operations in one of the following modes:

- **Single mode** - In single mode, the board acquires a single sample from an analog input channel. The driver initiates the conversion and the board acquires the data in the foreground; you cannot perform any other operation until the single-mode operation is complete.

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

- **Synchronous mode** - In synchronous mode, the board acquires a single sample or multiple samples from one or more analog input channels. A hardware conversion clock initiates conversions while the board acquires data in the foreground; you cannot perform any other operation until the synchronous-mode operation is complete. After the driver transfers the specified number of samples to the host, it returns control to the application program, which reads the data. Synchronous mode provides the fastest acquisition of multiple samples.

  You use the **K_SyncStart** function to start an analog input operation in synchronous mode. You specify the channel(s), gain(s), conversion clock source, buffer address, and trigger source.

- **Interrupt mode** - In interrupt mode, the board acquires a single sample or multiple samples from one or more analog input channels. A hardware conversion clock initiates conversions while the board acquires data in the background; system resources can be used by other programs. The driver transfers data to the host in the background using an interrupt service routine.

  You use the **K_IntStart** function to start an analog input operation in interrupt mode. You specify the channel(s), gain(s), conversion clock source, buffering mode, buffer address, and trigger source.

You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-16 for more information on buffering modes. You can use the **K_IntStop** function to stop a continuous-mode interrupt operation.

You can use the **K_IntStatus** function to determine the current status of an interrupt operation. In addition, you can use the **K_InitFrame** function to determine the status of all interrupt operations on a particular board.

For single mode, synchronous mode, and interrupt mode, the converted data is stored as raw counts. For information on converting raw counts to voltage, refer to Appendix B.

Note: In applications where you must accurately control the sampling rate, it is recommended that you perform the analog input operation in either synchronous mode or interrupt mode so that you can specify a conversion clock source.

Memory Allocation and Management

Synchronous-mode and interrupt-mode analog input operations require a memory buffer in which to store the acquired data. You can provide the required memory buffer in one of the following ways:

- **Within your application program's memory area** - The local memory buffer is always available to your program; however, your application program may require a large amount of memory. You can dimension a local memory buffer for any supported language. Since the DAS-800 Series Function Call Driver stores data in 16-bit integers, you must dimension all local memory buffers as integers.

- **Outside of your application program's memory area** - You allocate memory as needed. For all C languages, all Pascal languages, and Visual Basic for Windows, you can use the **K_IntAlloc** function to allocate memory dynamically, outside of your program's memory area. You specify the operation requiring the buffer, the number of samples to store in the buffer, the starting address of the buffer, and the name you want to use to identify the buffer (this name is called the memory handle). When the buffer is no longer required, you can free

the buffer for another use by specifying this memory handle in the **K_IntFree** function.

Note: You cannot allocate memory dynamically in BASIC; in BASIC, you must dimension the memory buffer locally.

You can use multiple buffers to increase the number of samples you can acquire. Each synchronous-mode or interrupt-mode analog input operation has a buffer list associated with it. You can use the **K_BufListAdd** function to add a buffer to the list of multiple buffers. You can use the **K_BufListReset** function to clear the list of multiple buffers.

Note: If you are using a Windows-based language in Enhanced mode, you may be limited in the amount of memory you can allocate. If you are allocating memory dynamically or if you are using multiple buffers, it is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate a large enough buffer or buffers. Refer to the *DAS-800 Series User's Guide* for more information about the Keithley Memory Manager.

After you allocate or dimension your buffer(s), you must specify the starting address of the buffer(s) and the number of samples to store in the buffer(s), as follows:

- **For BASIC** - You use the **K_SetBufI** function to specify the starting address of a single, locally dimensioned memory buffer. When using multiple buffers, you use the **K_BufListAdd** function both to add buffers to the multiple-buffer list and to specify the starting address of each buffer.

- **For Visual Basic for Windows** - You use the **K_SetBufI** function to specify the starting address of a single, locally dimensioned integer memory buffer; you use the **K_SetBuf** function to specify the starting address of a single buffer allocated dynamically using **K_IntAlloc**. When using multiple buffers, you use the **K_BufListAdd** function both to add buffers to the multiple-buffer list and to specify the starting address of each buffer.

Note: If you allocated your buffer dynamically using **K_IntAlloc**, you must use the **K_MoveBufToArray** function to transfer the acquired data from the dynamically allocated buffer to a local buffer that your Visual Basic for Windows program can use. Refer to page 3-25 for more information.

- **For C and Pascal** - You use the **K_SetBuf** function to specify the starting address of a single buffer, whether the buffer was dimensioned locally or allocated dynamically using **K_IntAlloc**. When using multiple buffers, you use the **K_BufListAdd** function both to add buffers to the multiple-buffer list and to specify the starting address of each buffer.

Input Range Type

Normally, the driver determines the input range type for a DAS-801 or DAS-802 board (bipolar or unipolar) by reading the configuration file. You can change the input range type without modifying the configuration file by using the **DAS800_SetADGainMode** function.

Note: The input range type of the DAS-800 board is always bipolar.

Use the **DAS800_GetADGainMode** function to get the current input range type. If you never used **DAS800_SetADGainMode**, **DAS800_GetADGainMode** reads the input range type from the configuration file; if you have used **DAS800_SetADGainMode**, **DAS800_GetADGainMode** reads the last input range type you programmed through software.

Gains

DAS-800 boards measure analog input signals in the range of ±5 V. DAS-801 and DAS-802 boards measure analog input signals in one of several software-selectable unipolar and bipolar ranges. For each channel on a DAS-801 or DAS-802 board, you can select one of five bipolar and four unipolar analog input ranges.

Table 2-2 lists the analog input ranges supported by DAS-800 Series boards and the gain and gain code associated with each range. (The gain code is used by the FCD functions to represent the gain.)

Table 2-2. Analog Input Ranges

Board	Analog Input Range		Gain	Gain Code
	Bipolar	Unipolar		
DAS-800	±5 V	Not available	1	0
DAS-801	±5 V	0 - 10 V	1	0
	±10 V	Not available	0.5	1
	±500 mV	0 - 1 V	10	2
	±50 mV	0 - 100 mV	100	3
	±10 mV	0 - 20 mV	500	4
DAS-802	±5 V	0 - 10 V	1	0
	±10 V	Not available	0.5	1
	±2.5 V	0 - 5 V	2	2
	±1.25 V	0 - 2.5 V	4	3
	±625 mV	0 - 1.25 V	8	4

Channels

The analog input channels are the analog input connections from which you acquire data. DAS-800 Series boards contain eight on-board analog input channels, numbered 0 through 7. If you require additional channels, you can use any combination of up to eight 16-channel EXP-16 or EXP-16/A expansion boards and/or 8-channel EXP-GP expansion boards to increase the number of available channels to 128. You can also use up to four MB-02 backplanes to increase the number of available channels to 68.

Expansion boards are assigned to consecutive on-board analog input channels, beginning with on-board channel 0. To ensure that the DAS-800 Series Function Call Driver reads the channel numbers correctly, you must attach all EXP-16 and EXP-16/A expansion boards first, followed by all EXP-GP expansion boards. You can also use the remaining on-board channels. Refer to the *DAS-800 Series User's Guide* or the appropriate expansion board documentation for more information.

The maximum supported configuration is eight EXP-16 or EXP-16/A expansion boards, eight EXP-GP expansion boards, or four MB-02 backplanes. Table 2-3 lists the software channels associated with each expansion board.

Table 2-3. Channels in Maximum Configuration

On-Board Channel	Software Channels		
	EXP-16 / EXP-16/A	EXP-GP	MB-02
0	0 to 15	0 to 7	0 to 15
1	16 to 31	8 to 15	16 to 31
2	32 to 47	16 to 23	32 to 47
3	48 to 63	24 to 31	48 to 63
4	64 to 79	32 to 39	64
5	80 to 95	40 to 47	65
6	96 to 111	48 to 55	66
7	112 to 127	56 to 63	67

Figure 2-1 illustrates the use of one EXP-16 expansion board, two EXP-GP expansion boards, and the five remaining on-board channels. The channels on the EXP-16 attached to analog input channel 0 are referred to in software as channels 0 to 15; the channels on the EXP-GP attached to analog input channel 1 are referred to in software as channels 16 to 23; the channels on the EXP-GP attached to analog input channel 2 are referred to in software as channels 24 to 31; the remaining five

on-board analog input channels (3, 4, 5, 6, and 7) are referred to in software as channels 32, 33, 34, 35, and 36.

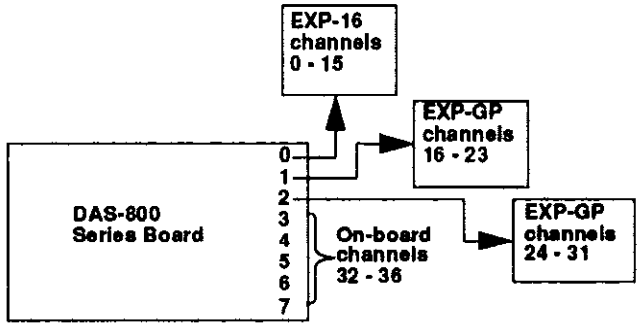


Figure 2-1. Analog Input Channels

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

Single Channel

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

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

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

Multiple Channels Using a Group of Consecutive Channels

For synchronous-mode and interrupt-mode analog input operations, you can acquire samples from a group of consecutive channels. You 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 are read.

For example, assume that you have an EXP-16/A expansion board attached to on-board channel 0. 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 (on-board channels 1 and 2), and finally from channel 14 again.

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

You can use the **K_SetG** function to specify the gain code for all channels in the group. (All channels in a group of consecutive channels must use the same gain code.) You can also 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 2-6 for a list of the analog input ranges supported by DAS-800 Series boards and the gain code associated with each range.

Multiple Channels Using a Channel-Gain List

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

The channels in a channel-gain list are not necessarily in consecutive order, and you can specify the same channel more than once (up to a total of 256 channels in the list). For the DAS-801 and DAS-802 boards, you can use a different gain code for each channel in a channel-gain list; for the DAS-800 board, every channel must use a gain code of 0 (gain of 1).

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

Refer to Table 2-2 on page 2-6 for a list of the analog input ranges supported by DAS-800 Series boards and the gain code associated with each range.

Note: The maximum attainable conversion frequency when using a channel-gain list is less than the maximum attainable conversion frequency when using a group of consecutive channels.

You specify the channels and gains in one of the following ways:

- **For C and Pascal** - You use two adjacent 8-bit bytes to specify a channel and its gain code (the channel number is specified in the first byte; the gain code is specified in the second byte). The first two bytes in the channel-gain list specify the number of channels (subsequent pairs of bytes) in the list. Figure 2-2 illustrates the format of a channel-gain list for C or Pascal, where *n* is the number of channels (pairs) in the list.

Byte	0	1	2	3	4	5	.....	2n	2n + 1
Value	n		chan	gain	chan	gain	.....	chan	gain
	# of pairs		pair 1		pair 2		.....	pair n	

Figure 2-2. Channel-Gain List (C or Pascal)

Figure 2-3 illustrates a channel-gain list of four channels on a DAS-801 board: channel 5 is sampled at a gain of 0.5 (gain code = 1), channel 2 is sampled at a gain of 10 (gain code = 2), channel 4 is sampled at a gain of 100 (gain code = 3), and channel 2 is sampled at a gain of 500 (gain code = 4).

Byte	0	1	2	3	4	5	6	7	8	9
Value	0	4	5	1	2	2	4	3	2	4
	4 pairs		pair 1		pair 2		pair 3		pair 4	

Figure 2-3. Sample Channel-Gain List (C or Pascal)

After you create the channel-gain list in C or Pascal, use the **K_SetChnGAry** function to specify the starting address of the list.

For Pascal only, you must define a record type for the channel-gain list before you specify the starting address. Refer to page 3-19 for more information.

- **For BASIC and Visual Basic for Windows** - You use two adjacent 16-bit words to specify a channel and its gain code (the channel number is specified in the first word; the gain code is specified in the second word). The first word in the channel-gain list specifies the number of channels (subsequent pairs of words) in the list. Figure 2-4 illustrates the format of a channel-gain list for BASIC and Visual Basic for Windows, where *n* is the number of channels (pairs) in the list.

Word	0	1	2	.....	2n - 1	2n
Value	n	chan	gain	.....	chan	gain
	# of pairs	pair 1		.....	pair n	

Figure 2-4. Channel-Gain List (BASIC)

Figure 2-5 illustrates a channel-gain list of three channels on a DAS-801 board: channel 5 is sampled at a gain of 0.5 (gain code = 1), channel 2 is sampled at a gain of 10 (gain code = 2), and channel 4 is sampled at a gain of 100 (gain code = 3).

Word	0	1	2	3	4	5	6
Value	3	5	1	2	2	4	3
	3 pairs	pair 1		pair 2		pair 3	

Figure 2-5. Sample Channel-Gain List (BASIC)

After you create your channel-gain list in BASIC or Visual Basic for Windows, you must use the **K_FormatChanGAry** function to convert the 16-bit values to 8-bit values that the DAS-800 Series Function Call Driver can use. After you use **K_FormatChanGAry** to convert your list, use the **K_SetChnGAry** function to specify the starting address of the list.

Your program cannot read the channel-gain list converted by the **K_FormatChanGAry** function; you must use the **K_RestoreChanGAry** function to restore the converted list to its original format.

Conversion Clocks

The conversion clock determines the time interval between conversions. For synchronous-mode and interrupt-mode analog input operations, you can use the **K_SetClk** function to specify an internal or an external conversion clock source. These conversion clock sources are described as follows:

- **Internal clock source** - The internal clock source is the on-board 8254 counter/timer circuitry. The 8254 counter/timer circuitry is normally in an idle state. When you start the analog input operation (using **K_IntStart** or **K_SyncStart**), a conversion is initiated immediately. The 8254 is loaded with a count value and begins counting down. When the 8254 counts down to 0, another conversion is initiated and the process repeats.

  Because the 8254 counter/timer uses a 1 MHz time base, each count represents 1 µs. Use the **K_SetClkRate** to specify the number of counts (clock ticks) between conversions. For example, if you specify a count of 25, the time interval between conversions is 25 µs; if you specify a count of 65535, the time interval between conversions is 65.535 ms.

  The 8254 contains three counter/timers: C/T0, C/T1, and C/T2. If you are using an internal clock source, the 8254 uses both C/T2 and C/T1. The driver uses C/T2 and C/T1 in either normal or cascaded mode, as follows:

  - *Normal mode* - The driver loads the count you specify into C/T2 of the 8254 counter/timer circuitry. Each time C/T2 reaches terminal count, a conversion is initiated. The time interval between conversions ranges from 25 µs to 65.535 ms.

  - *Cascaded mode* - The driver divides the count you specify between C/T2 and C/T1 of the 8254 counter/timer circuitry. When C/T2 counts down to 0, C/T1 decrements by 1. C/T2 is reloaded with its count value and begins counting down again. Each time C/T2 counts down to 0, C/T1 decrements by 1. Each time both C/T2 and C/T1 reach terminal count, a conversion is initiated. The time interval between conversions ranges from 25 µs to 1.2 hours.

Note: You configure the 8254 counter/timer circuitry for normal mode or cascaded mode using the D800CFG.EXE configuration utility. Refer to the *DAS-800 Series User's Guide* for more information.

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

$$\text{counts} = \frac{1\ \text{MHz}}{\text{conversion frequency}}$$

For example, if you want a conversion frequency of 10 kHz, specify a count of 100.

- **External clock source** - Use an external clock source if you want to sample at rates not available with the 8254 counter/timer circuitry, if you want to sample at uneven intervals, or if you want to sample on the basis of an external event.

  You attach an external clock source to the INT_IN / XCLK pin (pin 24). When you start the analog input operation (using **K_IntStart** or **K_SyncStart**), conversions are armed. At the next falling edge of the external clock source (and at every subsequent falling edge of the external clock source), a conversion is initiated.

Figure 2-6 illustrates the initiation of conversions when using an internal and an external clock source. (Note that Figure 2-6 assumes that you are not using an external trigger; refer to Figure 2-10 on page 2-21 for an illustration of conversions when using an external trigger.)

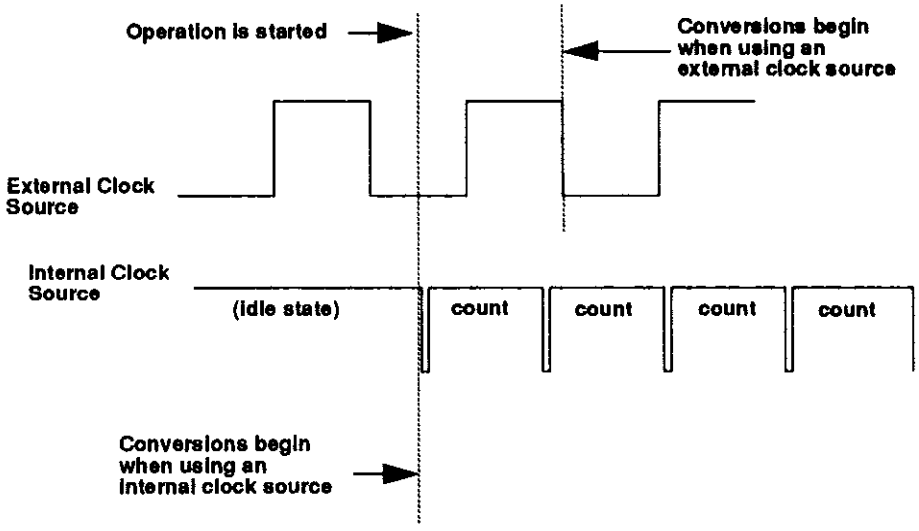


Figure 2-6. Initiating Conversions

Notes: The analog-to-digital converter (ADC) acquires samples at a maximum of 40 kHz (one sample every 25 µs). If you are using an external clock, make sure that the clock does not initiate conversions at a faster rate than the ADC can handle.

To achieve full measurement accuracy when using a gain of 500, you should limit the conversion frequency to a maximum of 25 kHz (one sample every 40 µs).

If you are acquiring samples from multiple channels, the maximum sampling rate for each channel is equal to 40 kHz divided by the number of channels.

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, whether you are using expansion boards, the gains of the channels, and other software issues.

For single-mode analog input operations, the software initiates each conversion with a call to the **K_ADRead** function.

Buffering Mode

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

- **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. You use the **K_SetContRun** function to specify continuous buffering mode.

- **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. You use the **K_ClrContRun** function to specify single-cycle buffering mode. (Note that single-cycle mode is the default buffering mode.)

Triggers

A trigger is a set of conditions that must occur before a DAS-800 Series board starts an analog input operation. For synchronous-mode and interrupt-mode analog input operations, you can use the **K_SetTrig** function to specify one of the following trigger sources:

- **Internal trigger** - An internal trigger is a software trigger; when you start the analog input operation (using **K_IntStart** or **K_SyncStart**), conversions begin immediately.

- **External trigger** - An external trigger is either an analog trigger or a digital trigger; when you start the analog input operation (using **K_IntStart** or **K_SyncStart**), the application program waits until a trigger event occurs and then begins conversions.

Analog and digital triggers are described in the following subsections.

Analog Triggers

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).

Figure 2-7 illustrates these analog trigger conditions, where the specified voltage level is +5 V.

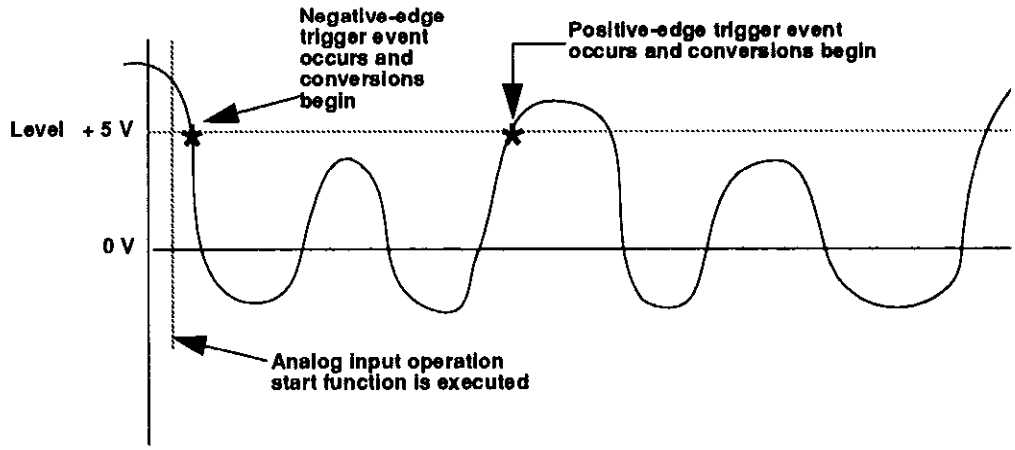


Figure 2-7. Analog Trigger Conditions

You use the **K_SetADTrig** function to specify the analog input channel to use as the trigger channel, the voltage level, the trigger polarity, and the trigger sense.

Note: You specify the voltage level as a raw count value between 0 and 4095. Refer to Appendix B for information on how to convert a voltage value to a raw count value.

You can use the **K_SetTrigHyst** function to specify a hysteresis value to prevent noise from triggering an operation. For a positive-edge trigger, the analog signal must fall below the specified voltage level by at least the amount of the hysteresis value before the trigger event can occur; for a negative-edge trigger, the analog signal must rise above the specified voltage level by at least the amount of the hysteresis value before the trigger event can occur.

The hysteresis value is an absolute number, which you specify as a raw 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 configured for a bipolar input range type. If the voltage level is +4.8 V (4014 counts), you can specify a hysteresis value of 0.1 V (41 counts), but you cannot specify a hysteresis value of 0.3 V (123 counts). Refer to Appendix B for information on how to convert a voltage value to a raw count value.

In Figure 2-8, the specified voltage level is +5 V and the hysteresis value is 0.1 V. The analog signal must fall below +4.9 V and then rise above +5 V before a positive-edge trigger event occurs; the analog signal must rise above +5.1 V and then fall below +5 V before a negative-edge trigger event occurs.

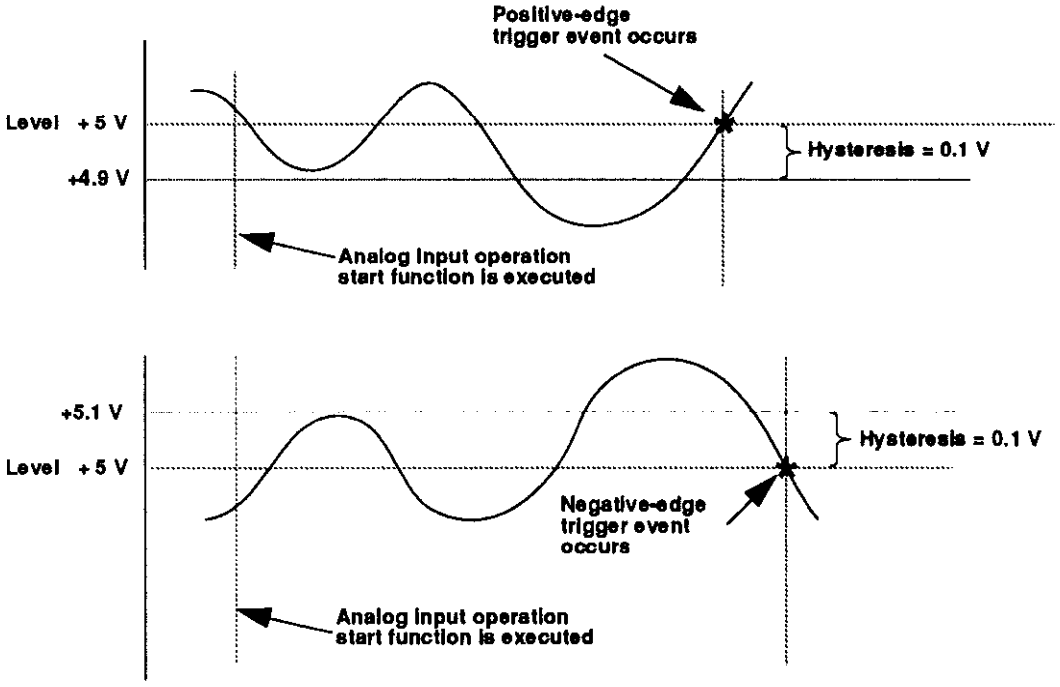


Figure 2-8. Using a Hysteresis Value

When using an analog trigger, the driver samples the specified analog trigger channel to determine whether the trigger condition has been met. Therefore, a slight time delay may occur between the time the trigger condition is actually met and the time the driver realizes that the trigger condition has been met and begins conversions. In addition, the actual point at which conversions begin depends on whether you are using an internal or external clock source. These considerations are described as follows:

- **Internal clock source** - The 8254 counter/timer circuitry remains idle until the driver detects the trigger event. When the driver detects the trigger event, the board begins conversions immediately.
- **External clock source** - Conversions are armed when the driver detects the trigger event. At the next falling edge of the external clock source, the board begins conversions.

Figure 2-9 illustrates how conversions are started when using an external analog trigger.

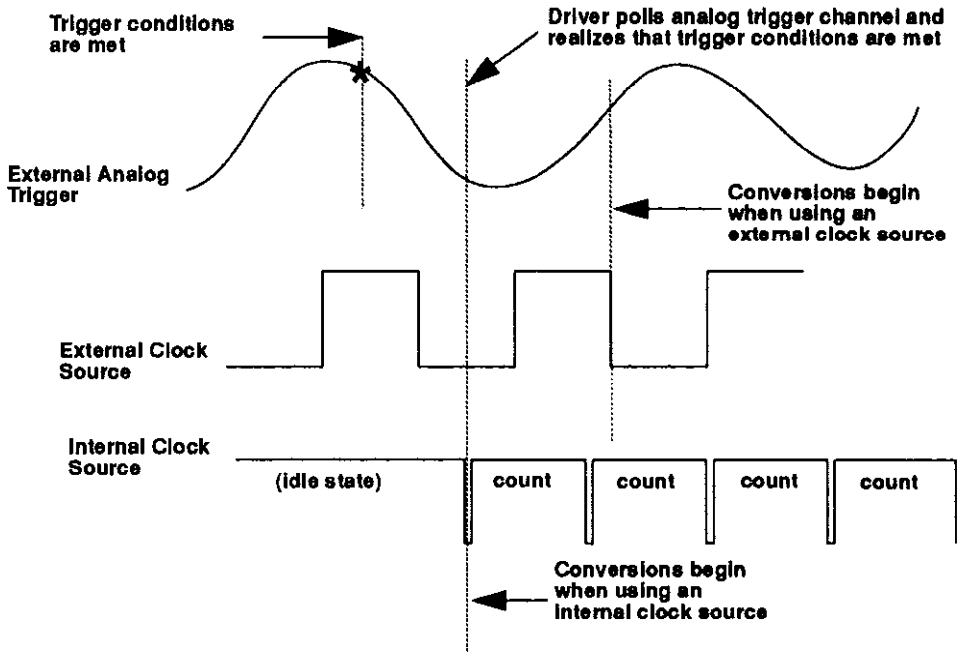


Figure 2-9. Initiating Conversions with an External Analog Trigger

Digital Triggers

A digital trigger event occurs when the board detects a rising edge on the digital trigger signal connected to the IP1 / TRIG pin (pin 25). You use the **K_SetDITrig** function to specify an external digital trigger.

When using a digital trigger, the actual point at which conversions begin depends on whether you are using an internal or external clock source. These considerations are described as follows:

- **Internal clock source** - The 8254 counter/timer circuitry remains idle until the trigger event occurs. When the trigger event occurs, the board begins conversions immediately.
- **External clock source** - Conversions are armed when the trigger event occurs. At the next falling edge of the external clock source, the board begins conversions.

Figure 2-10 illustrates how conversions are started when using an external digital trigger.

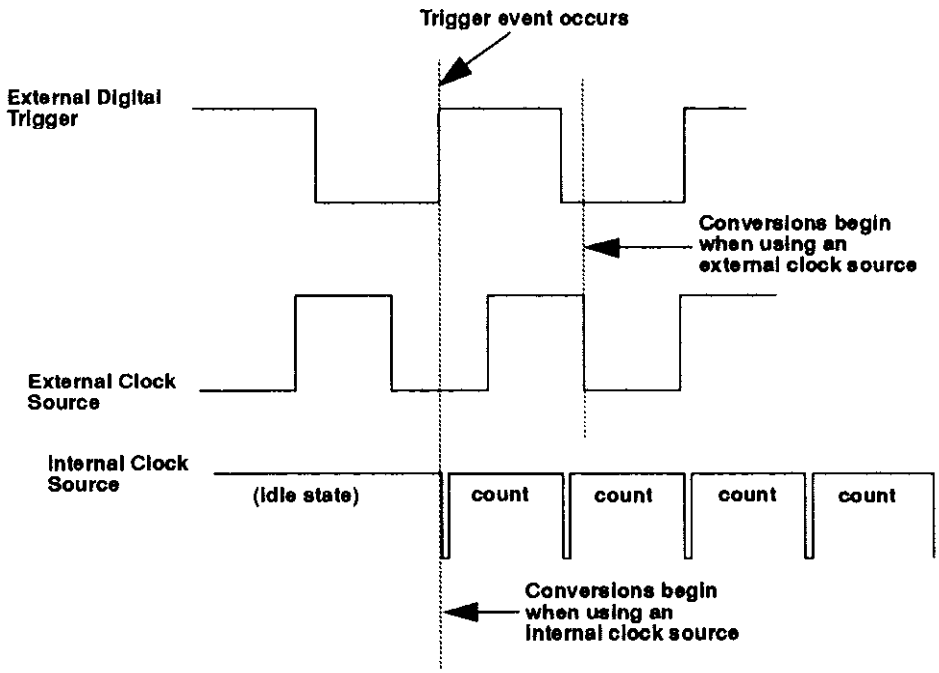


Figure 2-10. Initiating Conversions with an External Digital Trigger

Hardware Gates

A hardware gate is an externally applied digital signal that determines whether conversions occur. You connect the gate signal to the IP1 / TRIG pin (pin 25) on the main I/O connector. If you have started an analog input operation (using **K_IntStart** or **K_SyncStart**) and the hardware gate is enabled, the state of the gate signal determines whether conversions occur.

DAS-800 Series boards support a positive gate only. Therefore, if the signal to IP1 / TRIG is high, conversions occur; if the signal to IP1 / TRIG is low, conversions are inhibited. You use the **K_SetGate** function to enable and disable the hardware gate.

You can use the hardware gate with an external analog trigger. The software waits until the analog trigger event occurs and then checks the state of the gate signal. If the gate signal is high, conversions begin; if the gate signal is low, the software waits until the gate signal goes high before conversions begin.

If you are not using an analog trigger, the gate signal itself can act as a trigger. If the gate signal is low when you start the analog input operation, the software waits until the gate signal goes high before conversions begin.

Note: You cannot use the hardware gate with an external digital trigger. If you use a digital trigger at one point in your application program and later want to use a hardware gate, you must first disable the digital trigger. You disable the digital trigger by specifying an internal trigger in **K_SetTrig** or by setting up an analog trigger (using the **K_SetADTrig** function).

When the hardware gate is enabled, the way conversions are synchronized depends on whether you are using an external or an internal clock source. These considerations are described as follows:

- **Internal clock source** - The 8254 stops counting when the gate signal goes low. When the gate signal goes high again, the 8254 is reloaded with its initial count value and starts counting again; therefore, when using an internal clock, conversions are synchronized to the rising edge of the gate signal.

- **External clock source** - The signal from the external clock continues uninterrupted while the gate signal is low. When the gate signal goes high again, the software waits for the next falling edge of the external clock before initiating another conversion; therefore, when using an external clock, conversions are synchronized to the falling edge of the external clock.

Figure 2-11 illustrates the use of the hardware gate with both an external clock and an internal clock.

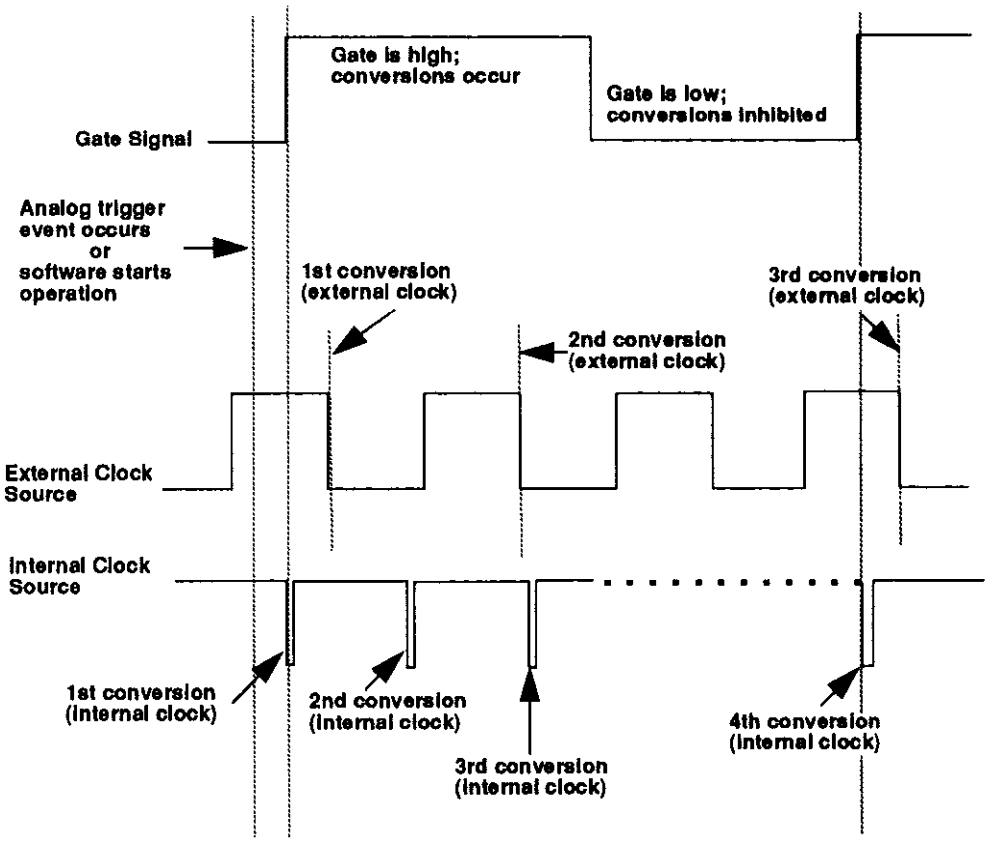


Figure 2-11. Hardware Gate

Digital I/O Operations

DAS-800 Series boards contain three digital input lines and four digital output lines. The digital input lines are associated with the IP1 / TRIG, IP2, and IP3 pins on the main I/O connector; the digital output lines are associated with the OP1, OP2, OP3, and OP4 pins on the main I/O connector. If the digital I/O lines are not used for an internal operation, you can use them for general-purpose digital I/O, as follows:

- **Digital input** - The DAS-800 Series Function Call Driver provides the **K_DIRead** function to read the value of digital input channel 0, a 32-bit channel that contains all the digital input lines. The **K_DIRead** function stores the value of digital input channel 0 in a 32-bit variable, where only bits 0, 1, and 2 are meaningful. As shown in Figure 2-12, bit 0 contains the value of digital input line 1 (IP1 / TRIG); bit 1 contains the value of digital input line 2 (IP2); bit 2 contains the value of digital input line 3 (IP3).

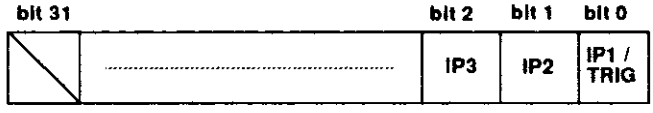


Figure 2-12. Digital Input Bits

A value of 1 in the bit position indicates that the input is high; a value of 0 in the bit position indicates that the input is low. For example, if the value is 5 (00...00101), the input at IP1 / TRIG and IP3 is high and the input at IP2 is low.

Notes: If you are using an external digital trigger, you cannot use the IP1 / TRIG pin (pin 25) for general-purpose digital input operations.

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

- **Digital output** - The DAS-800 Series Function Call Driver provides the **K_DOWrite** function to write a value to digital output channel 0, a 32-bit channel that contains all the digital output lines. The **K_DOWrite** function writes the value to digital output channel 0 as a 32-bit variable, where only bits 0, 1, 2, and 3 are meaningful. As shown in Figure 2-13, bit 0 contains the value written to digital output line 1 (OP1); bit 1 contains the value written to digital output line 2 (OP2); bit 2 contains the value written to digital output line 3 (OP3); bit 3 contains the value written to digital output line 4 (OP4).

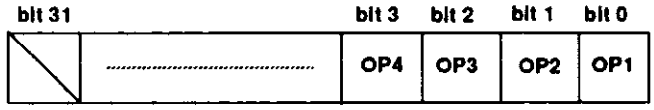


Figure 2-13. Digital Output Bits

A value of 1 in the bit position indicates that the output is high; a value of 0 in the bit position indicates that the output is low. For example, if the value written is 12 (00...01100), the output at OP1 and OP2 is forced low and the output at OP3 and OP4 is forced high.

Notes: The DAS-800 Series Function Call Driver does not provide a function for reading the current state of the digital output lines. To determine the last value written to the digital output lines, check your application program.

If you are using an expansion board for an analog input operation, the driver uses all four digital output lines to specify the expansion board channel that is acquiring data; in this case, you cannot use the digital output lines for general-purpose digital output operations.

Counter/Timer I/O Operations

DAS-800 Series boards contain 8254 counter/timer circuitry; the 8254 contains three counter/timers: C/T0, C/T1, and C/T2. If these counter/timers are not being used for an internal operation, you can use them for another task, such as frequency measurement.

Note: C/T0 is always available for general-purpose tasks. If you are using an internal clock source for an analog input operation, C/T2 and C/T1 are not available for general-purpose tasks. If you are using an external clock source, C/T0, C/T1, and C/T2 are always available for general-purpose tasks. Refer to page 2-13 for more information about the use of the 8254 as an internal clock source.

To configure a counter/timer on the 8254, you can use the **DAS800_Set8254** function. You specify both an initial count value to load into the counter/timer and a counter/timer mode. The initial count value can range from 2 to 65535. The following counter/timer modes are supported:

- Pulse on terminal count
- Programmable one-shot
- Rate generator
- Square-wave generator
- Software-triggered strobe
- Hardware-triggered strobe

Refer to the *DAS-800 Series User's Guide* for more information on the counter/timer modes and on how to program the 8254 counter/timer circuitry.

Use the **DAS800_Get8254** function to obtain the counter/timer mode and the current count value of a counter/timer on the 8254 counter/timer circuitry.

System Operations

This section describes the miscellaneous operations and general maintenance operations that apply to DAS-800 Series boards and to the DAS-800 Series Function Call Driver. It includes information on initializing the driver, initializing a board, retrieving the revision level, and handling errors.

Initializing the Driver

Before you can use any of the functions included in the DAS-800 Series Function Call Driver, you must initialize the driver using one of the following driver initialization functions:

- **Board-specific driver initialization function** - You can use the board-specific driver initialization function **DAS800_DevOpen** to initialize the DAS-800 Series Function Call Driver only. You specify a configuration file; **DAS800_DevOpen** initializes the driver according to the configuration file you specify. Refer to the *DAS-800 Series User's Guide* for information on creating and modifying configuration files.

- **Generic driver initialization function** - If you want to initialize several different DAS Function Call Drivers from the same application program, you can use the generic driver initialization function **K_OpenDriver**. You specify the DAS board you are using and a configuration file; **K_OpenDriver** initializes the driver according to the configuration file you specify. Refer to the *DAS-800 Series User's Guide* for information on creating and modifying configuration files.

  You also specify the name you want to use to identify this particular use of the driver; this name is called the driver handle. You can specify a maximum of 30 driver handles for all the DAS boards accessed from your application program.

  If a particular use of a driver is no longer required and you want to free some memory or if you have used all 30 driver handles, you can use the **K_CloseDriver** function to free a driver handle and close the associated use of the driver. **K_CloseDriver** also frees any system resources associated with the driver handle.

  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.)

Initializing a Board

The DAS-800 Series Function Call Driver supports up to four boards. You must use a board initialization function to specify the board you want to use and the name you want to use to identify the board; this name is called the board handle. Board handles allow you to communicate with more than one board. You use the board handle you specify in the board initialization function in all subsequent function calls related to the board.

The DAS-800 Series Function Call Driver provides the following board initialization functions:

- **Board-specific board initialization function** - You can use the board-specific board initialization function **DAS800_GetDevHandle** to initialize a DAS-800 Series board only.

- **Generic driver initialization function** - If you want to initialize several different DAS boards from the same application program, you can use the generic board initialization function **K_GetDevHandle**. You can specify a maximum of 30 board handles for all the DAS boards accessed from your application program.

  If a board is no longer being used and you want to free some memory or if you have used all 30 board handles, you can use the **K_FreeDevHandle** function to free a board handle. **K_FreeDevHandle** also frees any system resources associated with the board handle.

To reinitialize a board during an operation, you can use the **K_DASDevInit** function. **DAS800_GetDevHandle**, **K_GetDevHandle**, and **K_DASDevInit** perform the following tasks:

- Abort all analog input operations currently in progress that are associated with the board identified by the board handle.

- Verify that the board identified by the board handle is the board specified in the configuration file.

Retrieving the Revision Level

If you are using functions from different DAS Function Call Drivers in the same application program, you may want to verify which versions of

the Function Call Drivers are installed on your board to determine if a particular function is available to you. The **K_GetVer** function allows you to get both the revision number of the DAS-800 Series Function Call Driver and the revision number of the Keithley DAS Driver Specification to which the driver conforms.

Handling Errors

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

For C-language application programs only, the DAS-800 Series Function Call Driver provides the **K_GetErrMsg** function, which gets the address of the string corresponding to an error code.

3

Programming with the Function Call Driver

This chapter contains an overview of the structure of the DAS-800 Series Function Call Driver, as well as programming guidelines and language-specific information to assist you when writing application programs with the DAS-800 Series Function Call Driver.

How the Driver Works

The Function Call Drivers for all DAS boards allow you to perform I/O operations in various operation modes. For single mode, the I/O operation is performed with a single call to a function; the attributes of the I/O operation are specified as arguments to the function and a single value is obtained. For other operation modes, such as synchronous mode and interrupt mode, the driver uses frames to perform the I/O operation. A frame is a data structure whose elements define the particular I/O operation.

Frames help you create structured application programs. You set up the attributes of the I/O operation in advance, using a separate function call for each attribute, and then start the operation at an appropriate point in your program. Frames are useful for operations that have many defining attributes, since providing a separate argument for each attribute could make a function's argument list unmanageably long. In addition, some attributes, such as conversion clock source and trigger source, are only available for I/O operations that use frames.

You indicate that you want to perform an I/O operation by getting an available frame for the driver and specifying the name you want to use to identify the frame; this name is called the frame handle. You then specify the attributes of the I/O operation by using setup functions to define the elements of the frame associated with the operation. For example, to specify the channel on which to perform an I/O operation, you might use the **K_SetChn** setup function.

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

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

When you are ready to perform the I/O operation you have set up, you can start the operation in the appropriate operation mode, referencing the appropriate frame handle.

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

For DAS-800 Series boards, the only operations that use frames are synchronous-mode and interrupt-mode analog input operations. The DAS-800 Series Function Call Driver provides eight identical analog input frames, called A/D (analog-to-digital) frames. You use the **K_GetADFrame** function to access an available A/D frame and specify a frame handle.

Note: Drivers for other DAS boards may provide additional functions for accessing analog output, digital input, or digital output frames.

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

Table 3-1 lists the elements of a DAS-800 A/D frame, the default value of each element, the setup function(s) used to define each element, and the readback function(s) used to read the current definition of the element.

Table 3-1. A/D Frame Elements

Element	Default Value	Setup Function	Readback Function
Buffering Mode	Single-cycle	K_ClrContRun K_SetContRun	K_GetContRun
Buffer[1]	0 (NULL)	K_SetBuf K_SetBufI K_BufListAdd	K_GetBuf
Number of Samples	0	K_SetBuf K_SetBufI K_BufListAdd	K_GetBuf
Start Channel	0	K_SetChn K_SetStartStopChn K_SetStartStopG	K_GetChn K_GetStartStopChn K_GetStartStopG
Stop Channel	0	K_SetStartStopChn K_SetStartStopG	K_GetStartStopChn K_GetStartStopG
Gain	0	K_SetG K_SetStartStopG	K_GetG K_GetStartStopG
Channel-Gain List	0 (NULL)	K_SetChnGAry	K_GetChnGAry
Conversion Clock Source	Internal	K_SetClk	K_GetClk
Conversion Frequency	25 (40 KHz)	K_SetClkRate	K_GetClkRate
Trigger Source	Internal	K_SetTrig	K_GetTrig
Trigger Type	Digital	K_SetADTrig K_SetDITrig	K_GetADTrig K_GetDITrig
Trigger Channel	0 (for analog trigger)	K_SetADTrig	K_GetADTrig
	0 (Channel 0, Bit 0) (for digital trigger)	Not applicable[2]	Not applicable[2]

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

Element	Default Value	Setup Function	Readback Function
Trigger Polarity	Positive (for analog trigger)	K_SetADTrig	K_GetADTrig
	Positive (for digital trigger)	Not applicable[2]	Not applicable[2]
Trigger Sense	Edge (for analog and digital trigger)	Not applicable[2]	Not applicable[2]
Trigger Level	0	K_SetADTrig	K_GetADTrig
Trigger Hysteresis	0	K_SetTrigHyst	K_GetTrigHyst
Trigger Pattern	Not used[3]	Not applicable[2]	Not applicable[2]
Hardware Gate	Disabled	K_SetGate	K_GetGate

Notes
[1] This element must be set.
[2] The default value of this element cannot be changed.
[3] This element is not currently used; it is included for future compatibility.

When you access an A/D frame with **K_GetADFrame**, the elements are set to their default values. You can also use the **K_ClearFrame** function to return all the elements of a frame to their default values.

Note: The DAS-800 Series Function Call Driver provides many other functions that are not related to controlling frames, defining the elements of frames, or reading the values of frame elements. These functions include single-mode operation functions, initialization functions, memory management functions, and other miscellaneous functions.

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

Programming Overview

To write an application program using the DAS-800 Series Function Call Driver, perform the following steps:

1. Define the application's requirements. Refer to Chapter 2 for a description of the board operations supported by the Function Call Driver and the functions that you can use to define each operation.

2. Write your application program. Refer to the following for additional information:

   - Preliminary Tasks, the next section, describes the programming tasks that are common to all application programs.

   - Operation-Specific Programming Tasks, on page 3-6, describes operation-specific programming tasks and the sequence in which these tasks must be performed.

   - Chapter 4 contains detailed descriptions of the FCD functions.

   - The DAS-800 Series standard software package and the ASO-800 software package contain several example programs. The FILES.TXT file in the installation directory lists and describes the example programs.

3. Compile and link the program. Refer to Language-Specific Programming Information, starting on page 3-12, for compile and link statements and other language-specific considerations for each supported language.

Preliminary Tasks

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

1. Include the function and variable type definition file for your language. Depending on the specific language you are using, this file is included in the DAS-800 Series standard software package or the ASO-800 software package.

2. Declare and initialize program variables.

3. Use a driver initialization function (**DAS800_DevOpen** or **K_OpenDriver**) to initialize the driver.

4. Use a board initialization function (**DAS800_GetDevHandle** or **K_GetDevHandle**) to specify the board you want to use and to initialize the board. If you are using more than one board, use the board initialization function once for each board you are using.

Operation-Specific Programming Tasks

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

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

Analog Input Operations

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

Single Mode

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

1. Declare the buffer or variable that will hold the single value to be read.

2. Use the **K_ADRead** function to read the single analog input value; specify the attributes of the operation as arguments to the function.

Synchronous Mode

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

1. Use the **K_GetADFrame** function to access an A/D frame.

2. Allocate or dimension the buffer(s) in which to store the acquired data. Use the **K_IntAlloc** function if you want to allocate the buffer(s) dynamically outside your program's memory area.

3. *If you want to use a channel-gain list to specify the channels acquiring data*, define and assign the appropriate values to the list and note the starting address. Refer to page 2-9 for more information about channel-gain lists.

4. Use the appropriate setup functions to assign values to those elements of the frame that pertain to your application. The setup functions are listed in Table 3-2.

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

Element	Setup Function(s)
Buffer[1]	K_SetBuf K_SetBufI K_BufListAdd
Number of Samples	K_SetBuf K_SetBufI K_BufListAdd

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

Element	Setup Function(s)
Start Channel	K_SetChn K_SetStartStopChn K_StartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStopG
Channel-Gain List	K_SetChnGAry
Conversion Clock Source	K_SetClk
Conversion Frequency	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig
Trigger Polarity	K_SetADTrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst
Hardware Gate	K_SetGate

Notes

[1] You must assign the addresses of all allocated or dimensioned buffers.

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

5. Use the **K_SyncStart** function to start the synchronous operation.

6. *If you are programming in Visual Basic for Windows and you used **K_IntAlloc** to allocate your buffer(s)*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to a local buffer that your program can use.

7. *If you used **K_IntAlloc** to allocate your buffer(s)*, use the **K_IntFree** function to deallocate the buffer(s).

8. *If you used **K_BufListAdd** to specify a list of multiple buffers*, use the **K_BufListReset** function to clear the list.

9. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt Mode

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

1. Use the **K_GetADFrame** function to access an A/D frame.

2. Allocate or dimension the buffer(s) in which to store the acquired data. Use the **K_IntAlloc** function if you want to allocate the buffer(s) dynamically outside your program's memory area.

3. *If you want to use a channel-gain list to specify the channels acquiring data*, define and assign the appropriate values to the list and note the starting address. Refer to page 2-9 for more information about channel-gain lists.

4. Use the appropriate setup functions to assign values to those elements of the frame that pertain to your application. The setup functions are listed in Table 3-3.

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

Element	Setup Function(s)
Buffer[1]	K_SetBuf K_SetBufI K_BufListAdd
Number of Samples	K_SetBuf K_SetBufI K_BufListAdd
Buffering Mode	K_ClrContRun K_SetContRun
Start Channel	K_SetChn K_SetStartStopChn K_StartStopG
Stop Channel	K_SetStartStopChn K_SetStartStopG
Gain	K_SetG K_SetStartStopG
Channel-Gain List	K_SetChnGAry
Conversion Clock Source	K_SetClk
Conversion Frequency	K_SetClkRate
Trigger Source	K_SetTrig
Trigger Type	K_SetADTrig K_SetDITrig
Trigger Channel	K_SetADTrig
Trigger Polarity	K_SetADTrig
Trigger Level	K_SetADTrig
Trigger Hysteresis	K_SetTrigHyst
Hardware Gate	K_SetGate

Notes

[1] You must assign the addresses of all allocated or dimensioned buffers.

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

5. Use the **K_IntStart** function to start the interrupt operation.

6. Use the **K_IntStatus** function to monitor the status of the interrupt operation.

7. *If you specified continuous buffering mode*, use the **K_IntStop** function to stop the interrupt operation when the appropriate number of samples has been acquired.

8. *If you are programming in Visual Basic for Windows and you used **K_IntAlloc** to allocate your buffer(s)*, use the **K_MoveBufToArray** function to transfer the acquired data from the allocated buffer to a local buffer that your program can use.

9. *If you used **K_IntAlloc** to allocate your buffer(s)*, use the **K_IntFree** function to deallocate the buffer(s).

10. *If you used **K_BufListAdd** to specify a list of multiple buffers*, use the **K_BufListReset** function to clear the list.

11. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Digital I/O Operations

You can perform digital I/O operations in single mode only. To perform a single-mode digital I/O operation, perform the following tasks:

1. Declare the buffer or variable that will hold the single value to be read or written.

2. Use one of the following digital I/O single-mode operation functions, specifying the attributes of the operation as arguments to the function:

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

Language-Specific Programming Information

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

Microsoft C/C++

To program in Microsoft C/C++, you need the following files; these files are provided in the ASO-800 software package.

File	Description
DAS800.LIB	Linkable driver.
DASRFACE.LIB	Linkable driver.
DASDECL.H	Include file when compiling in C (.c programs).
DAS800.H	Include file when compiling in C (.c programs).
DASDECL.HPP	Include file when compiling in C++ (.cpp programs).
DAS800.HPP	Include file when compiling in C++ (.cpp programs).
USE800.OBJ	Linkable object.

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

Type of Compile	Compile and Link Statements
C	CL /c *filename*.c LINK *filename*+use800.obj,,,das800+dasrface;
C++	CL /c *filename*.cpp LINK *filename*+use800.obj,,,das800+dasrface;

Borland C/C++

To program in Borland C/C++, you need the following files; these files are provided in the ASO-800 software package.

File	Description
DAS800.LIB	Linkable driver.
DASRFACE.LIB	Linkable driver.
DASDECL.H	Include file when compiling in C (.c programs).
DAS800.H	Include file when compiling in C (.c programs).
DASDECL.HPP	Include file when compiling in C++ (.cpp programs).
DAS800.HPP	Include file when compiling in C++ (.cpp programs).
USE800.OBJ	Linkable object.

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

Type of Compile	Compile and Link Statements[1]
C	BCC -ml *filename*.c use800.obj das800.lib dasrface.lib
C++	BCC -ml *filename*.cpp use800.obj das800.lib dasrface.lib

Notes

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

Microsoft QuickC for Windows

To program in Microsoft QuickC for Windows, you need the following files; these files are provided in the ASO-800 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library of user-interface functions.
DASSUPRT.DLL	Dynamic Link Library used by DASSHELL.DLL.
DAS800.DLL	Dynamic Link Library of DAS-800 board-specific functions.
DASDECL.H	Include file.
DAS800.H	Include file.

To create an executable file in Microsoft QuickC for Windows, perform the following steps:

1. Load *filename*.c into the QuickC for Windows environment, where *filename* indicates the name of your application program.

2. Create a project file. The project file should contain all necessary files, including *filename*.c, *filename*.rc, *filename*.def, and *filename*.h, where *filename* indicates the name of your application program.

3. From the Project menu, choose Build to create a stand-alone executable file (.EXE) that you can execute from within Windows.

Microsoft Visual C++

To program in Microsoft Visual C++, you need the following files; these files are provided in the ASO-800 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library of user-interface functions.
DASSUPRT.DLL	Dynamic Link Library used by DASSHELL.DLL.
DAS800.DLL	Dynamic Link Library of DAS-800 board-specific functions.
DASDECL.HPP	Include file.
DAS800.HPP	Include file.

Refer to the README.TXT file for information about creating an executable file in Visual C++.

Borland Turbo Pascal

To program in Borland Turbo Pascal, you need the following files; these files are provided in the ASO-800 software package.

File	Description
D800TP6.TPU	Turbo Pascal unit for Version 6.0.
D800TP7.TPU	Turbo Pascal unit for Version 7.0.
D800TPU.BAT[1]	Batch file for creating a Turbo Pascal unit.
D800.PAS[1]	Source code for creating a Turbo Pascal unit; required when upgrading the compiler.
D800TPU.INC[1]	Include file for creating a Turbo Pascal unit.
*.TBJ[1]	Object files used for creating a Turbo Pascal unit.

Notes

[1] Used for creating a new Turbo Pascal unit when compiling in Borland Turbo Pascal for versions higher than 7.0.

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 application program.

Refer to page 3-18 for information about specifying the buffer address when programming in Borland Turbo Pascal. Refer to page 3-19 for information about specifying the channel-gain list starting address when programming in Borland Turbo Pascal.

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-800 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library of user-interface functions.
DASSUPRT.DLL	Dynamic Link Library used by DASSHELL.DLL.
DAS800.DLL	Dynamic Link Library of DAS-800 board-specific functions.
DASDECL.INC	Include file.
DAS800.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 application program.

2. From the Compile menu, choose Make.

Refer to the next section for information about specifying the buffer address when programming in Borland Turbo Pascal for Windows. Refer to page 3-19 for information about specifying the channel-gain list starting address when programming in Borland Turbo Pascal for Windows.

Specifying the Buffer Address (Pascal)

If you are writing your application program in Borland Turbo Pascal or Borland Turbo Pascal for Windows, perform the following steps to specify a buffer address:

1. Reduce the memory heap reserved by Pascal by entering the following:

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

2. Declare a dummy type array of ^Integer, as in the following example:

```
Type
   IntArray = Array[0..1] of ^Integer;
   . . .
```

   The dimension of this array is irrelevant; it is used only to satisfy Pascal's type-checking requirements.

3. Declare an array of the dummy type, as in the following example:

```
Var
   acqBuf : IntArray;
   . . .
```

4. *If you are allocating your buffer dynamically using* ***K_IntAlloc***, use Pascal's Addr() function, as in the following example:

```
err : = K_IntAlloc (frameHandle, samples,
   Addr(acqBuf), memHandle);
```

5. Use **K_SetBuf** to specify the buffer address, as in the following example:

```
err : = K_SetBuf (frameHandle, acqBuf, samples);
```

This procedure allows you to directly access data stored in the buffer. You can retrieve data from the buffer, as in the following example:

```
For I := 0 to (samples - 1) do
Begin;
   data := acqBuf^[I];
End;
```

Specifying the Channel-Gain List Starting Address (Pascal)

If you are writing your application program in Borland Turbo Pascal or Borland Turbo Pascal for Windows, perform the following steps to specify a channel-gain list starting address:

1. Define a record type for the channel-gain list, as in the following example:

```
Type
   ChanGainArray = Record;
      num_of_codes : Integer;
      queue : Array[0..15] of Byte;
end;
```

2. Define an array of type ChanGainArray, as in the following example:

```
Var
   CGList : ChanGainArray;
   . . .
```

3. After this is initialized, the array can be passed to the function, as in the following example:

```
err : = K_SetChnGAry (ADFrame1, CGList.num_of_codes);
```

Microsoft QuickBASIC (Version 4.0)

To program in Microsoft QuickBASIC (Version 4.0), you need the following files; these files are provided in the DAS-800 Series standard software package.

File	Description
D800QB40.LIB	Linkable driver for QuickBASIC, Version 4.0, stand-alone, executable (.EXE) programs.
D800QB40.QLB	Command-line loadable driver for the QuickBASIC, Version 4.0, integrated environment.
QB4DECL.BI	Include file.
DASDECL.BI	Include file.
DAS800.BI	Include file.

For Microsoft QuickBASIC (Version 4.0), you can create an executable file from within the programming environment, or you can use a compile and link statement.

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

1. Enter the following to invoke the environment:

```
QB /L D800QB40 filename.bas
```

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

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

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

```
BC filename.bas /O
Link filename.obj,,,D800QB40.lib+BCOM40.lib;
```

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

Refer to page 3-25 for information about specifying the buffer address when programming in Microsoft QuickBASIC (Version 4.0).

Microsoft QuickBasic (Version 4.5)

To program in Microsoft QuickBasic (Version 4.5), you need the following files; these files are provided in the DAS-800 Series standard software package.

File	Description
D800QB45.LIB	Linkable driver for QuickBasic, Version 4.5, stand-alone, executable (.EXE) programs.
D800QB45.QLB	Command-line loadable driver for the QuickBasic, Version 4.5, integrated environment.
QB4DECL.BI	Include file.
DASDECL.BI	Include file.
DAS800.BI	Include file.

For Microsoft QuickBasic (Version 4.5), you can create an executable file from within the programming environment, or you can use a compile and link statement.

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

1. Enter the following to invoke the environment:

   ```
   QB /L D800QB45 filename.bas
   ```

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

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

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

```
BC filename.bas /O
Link filename.obj,,,D800QB45.lib+BCOM45.lib;
```

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

Refer to page 3-25 for information about specifying the buffer address when programming in Microsoft QuickBasic (Version 4.5).

Microsoft Professional Basic (Version 7.0)

To program in Microsoft Professional Basic (Version 7.0), you need the following files; these files are provided in the DAS-800 Series standard software package.

File	Description
D800QBX.LIB	Linkable driver for Professional Basic, Versions 7.0 and higher, stand-alone, executable (.EXE) programs.
D800QBX.QLB	Command-line loadable driver for the Professional Basic, Versions 7.0 and higher, integrated environment.
DASDECL.BI	Include file.
DAS800.BI	Include file.

For Microsoft Professional Basic (Version 7.0), you can create an executable file from within the programming environment, or you can use a compile and link statement.

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

1. Enter the following to invoke the environment:

   ```
   QBX /L D800QBX filename.bas
   ```

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

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

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

```
BC filename.bas /o;
Link filename.obj,,,D800QBX.lib;
```

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

Refer to page 3-25 for information about specifying the buffer address when programming in Microsoft Professional Basic (Version 7.0).

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-800 Series standard software package.

File	Description
DASDECL.BI	Include file.
DAS800.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 D800VBD.QLB filename.BAS
   ```

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

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

Refer to page 3-25 for information about specifying the buffer address when programming in Microsoft Visual Basic for DOS.

Microsoft Visual Basic for Windows

To program in Microsoft Visual Basic for Windows, you need the following files; these files are provided in the ASO-800 software package.

File	Description
DASSHELL.DLL	Dynamic Link Library of user-interface functions.
DASSUPRT.DLL	Dynamic Link Library used by DASSHELL.DLL.
DAS800.DLL	Dynamic Link Library of DAS-800 board-specific functions.
DASDECL.BAS	Include file; must be added to the Project List.
DAS800.BAS	Include file; must be added to the Project List.

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

Refer to the next section for information about specifying the buffer address when programming in Microsoft Visual Basic for Windows.

Specifying the Buffer Address (All BASIC Languages)

This section describes how to specify a buffer address when programming in BASIC and Visual Basic for Windows.

For Visual Basic for Windows, if you are allocating your buffer dynamically using ***K_IntAlloc****,* perform the following steps to specify the buffer address:

1. Declare the allocated buffer pointer, as in the following example:

```
Global AllocBuf As Long
```

2. Allocate the buffer, as in the following example:

```
errnum = K_IntAlloc (frameHandle, samples,
   AllocBuf, memHandle)
```

   Refer to page 4-78 for more information about the **K_IntAlloc** function.

3. In defining the elements of your frame, specify the buffer address, as in the following example:

```
errnum = K_SetBuf (frameHandle, AllocBuf, samples)
```

   Refer to page 4-95 for more information about the **K_SetBuf** function.

4. After all your data is acquired, move the data from the allocated buffer to a local storage buffer that your program can access, as in the following example:

```
errnum = K_MoveBufToArray (Buffer(0), AllocBuf,
   samples)
```

   Refer to page 4-88 for more information about the **K_MoveBufToArray** function.

For BASIC and Visual Basic for Windows, if you are dimensioning your buffer locally, perform the following steps to specify the buffer address:

1. Declare the local buffer, as in the following example:

```
Global Buffer(20000) As Integer
```

2. In defining the elements of your frame, specify the buffer address, as in the following example:

```
errnum = K_SetBufI (frameHandle, Buffer(0),
   samples)
```

   Refer to page 4-97 for more information about the **K_SetBufI** function.

Notes: The local buffer is accessible to your program; you do not have to use **K_MoveBufToArray** to move it.

Do not use underscores in the BASIC languages.

4

Function Reference

The FCD functions are organized into the following groups:

- Initialization functions
- Operation functions
- Frame management functions
- Memory management functions
- Buffer address functions
- Buffering mode functions
- Channel and gain functions
- Conversion clock functions
- Trigger functions
- 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. FCD Functions

Function Type	Function Name	Page Number
Initialization	DAS800_DevOpen	page 4-6
	K_OpenDriver	page 4-89
	K_CloseDriver	page 4-28
	DAS800_GetDevHandle	page 4-11
	K_GetDevHandle	page 4-56
	K_FreeDevHandle	page 4-38
	K_DASDevInit	page 4-32
Operation	K_ADRead	page 4-19
	K_DIRead	page 4-33
	K_DOWrite	page 4-35
	K_SyncStart	page 4-124
	K_IntStart	page 4-81
	K_IntStatus	page 4-83
	K_IntStop	page 4-86
Frame Management	K_GetADFrame	page 4-40
	K_FreeFrame	page 4-39
	K_ClearFrame	page 4-26
Memory Management	K_IntAlloc	page 4-78
	K_IntFree	page 4-80
	K_MoveBufToArray	page 4-88

Table 4-1. FCD Functions (cont.)

Function Type	Function Name	Page Number
Buffer Address	K_SetBuf	page 4-95
	K_SetBufI	page 4-97
	K_GetBuf	page 4-44
	K_BufListAdd	page 4-22
	K_BufListReset	page 4-24
Buffering Mode	K_ClrContRun	page 4-30
	K_SetContRun	page 4-107
	K_GetContRun	page 4-54
Channel and Gain	K_SetChn	page 4-99
	K_SetStartStopChn	page 4-115
	K_SetG	page 4-111
	K_SetStartStopG	page 4-117
	K_SetChnGAry	page 4-101
	K_FormatChanGAry	page 4-37
	K_RestoreChanGAry	page 4-92
	K_GetChn	page 4-46
	K_GetStartStopChn	page 4-65
	K_GetG	page 4-61
	K_GetStartStopG	page 4-67
	K_GetChnGAry	page 4-48
	DAS800_SetADGainMode	page 4-15
	DAS800_GetADGainMode	page 4-9
Conversion Clock	K_SetClk	page 4-103
	K_SetClkRate	page 4-105
	K_GetClk	page 4-50
	K_GetClkRate	page 4-52

Table 4-1. FCD Functions (cont.)

Function Type	Function Name	Page Number
Trigger	K_SetTrig	page 4-120
	K_SetADTrig	page 4-93
	K_SetTrigHyst	page 4-122
	K_SetDITrig	page 4-109
	K_GetTrig	page 4-70
	K_GetADTrig	page 4-42
	K_GetTrigHyst	page 4-72
	K_GetDITrig	page 4-58
Gate	K_SetGate	page 4-113
	K_GetGate	page 4-63
Counter/Timer	DAS800_Set8254	page 4-17
	DAS800_Get8254	page 4-13
Miscellaneous	K_GetErrMsg	page 4-60
	K_GetVer	page 4-74
	K_InitFrame	page 4-76

Keep the following conventions in mind throughout this chapter:

- Although the function names are shown with underscores, do not use the underscores in the BASIC languages.
- The data types DDH, FRAMEH, DWORD, WORD, and BYTE are defined in the language-specific include files.
- Variable names are shown in italics.
- The return value for all FCD functions is the error/status code. Refer to Appendix A for more information.
- The syntax shows the format of the function and the data types of the parameters. This line of code is not necessarily the exact line of code you would enter in your application program. In addition, data types must be defined before you enter the line of code.
- Entry parameters are parameters that are passed to the function but not changed by the function.
- Exit parameters are parameters that are modified by the function.
- In the examples, the variables are not defined. It is assumed that they are defined as shown in the syntax.

DAS800_DevOpen

Purpose

Initializes the DAS-800 Series Function Call Driver.

Syntax

C

DAS800_DevOpen (*cfgFile*, *numDevices*);
char **cfgFile*;
char **numDevices*;

Pascal

DAS800_DevOpen (*cfgFile*, *numDevices*) : Word;
cfgFile : String;
numDevices : Integer;

Visual Basic for Windows

DAS800_DevOpen (*cfgFile*, *numDevices*) As Integer
Dim *cfgFile* As String
Dim *numDevices* As Integer

BASIC

DAS800DevOpen% (*cfgfile*, *numDevices*)
Dim *cfgFile* As String
Dim *numDevices* As Integer

Entry Parameters

cfgFile — Driver configuration file.
Valid values: **0** = DAS800.CFG
–1 = Default configuration
filename = Any configuration file

Exit Parameters

numDevices — Number of boards defined in *cfgFile*.
Valid values: 1 to 4

Notes

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

You create a configuration file using the D800CFG.EXE utility. Refer to the *DAS-800 Series User's Guide* for more information.

If *cfgFile* = 0, **DAS800_DevOpen** looks for the DAS800.CFG configuration file in the current directory and uses those settings, if available. If *cfgFile* = −1, **DAS800_DevOpen** initializes the driver to its default configuration; the default configuration is shown in Table 4-2.

Table 4-2. Default Configuration

Attribute	Default Configuration
Board type	DAS-800
Base address	300H[1]
8254 C/T2 usage	Cascaded
Input range type	Bipolar
Channel 0 input configuration	Single-ended
Channel 1 input configuration	Single-ended
Channel 2 input configuration	Single-ended
Channel 3 input configuration	Single-ended
Channel 4 input configuration	Single-ended
Channel 5 input configuration	Single-ended
Channel 6 input configuration	Single-ended
Channel 7 input configuration	Single-ended
Interrupt level	X (Disabled)
Number of EXP-16s	0
Gain of EXP-16s	[N/A]
Number of EXP-GPs	0
Gain of EXP-GPs	[N/A]
CJR channel	−1 (Disabled)

Notes

[1] The default base address for board 0 is 300H. If you are using multiple DAS-800 Series boards, the default base address for board 1 is 308H, the default base address for board 2 is 310H, and the default base address for board 3 is 318H.

The Function Call Driver requires null terminated strings. To create null terminated strings in Pascal, BASIC, and Visual Basic for Windows, refer to the following examples. These examples assume that the configuration file (*cfgFile*) is DAS800.CFG.

Pascal: *cfgFile* : = 'DAS800.CFG' + #0;

BASIC and Visual Basic for Windows:

cfgFile = "DAS800.CFG" + CHR$(0)

Example

After you set up your DAS-801 board, you created a configuration file to reflect the settings of the jumper and switches on the board. The name of the configuration file is stored in the memory location pointed to by CONF801. You want to initialize the DAS-800 Series Function Call Driver according to this configuration file and store the number of boards defined in the configuration file in a variable called NumberOfBoards.

C

```
char NumberOfBoards;
err = DAS800_DevOpen (CONF801, &NumberOfBoards);
```

Pascal

```
err : = DAS800_DevOpen (CONF801[1], NumberOfBoards);
```

Visual Basic for Windows

```
errnum = DAS800_DevOpen (CONF801, NumberOfBoards)
```

BASIC

```
errnum = DAS800DevOpen% (CONF801, NumberOfBoards)
```

DAS800_GetADGainMode

Purpose

Gets the current input range type (unipolar or bipolar).

Syntax

C

DAS800_GetADGainMode (*devNumber*, *mode*);
short *devNumber*;
short **mode*;

Pascal

DAS800_GetADGainMode (*devNumber*, *mode*) : Word;
devNumber : Integer;
mode : Integer;

Visual Basic for Windows

DAS800_GetADGainMode (*devNumber*, *mode*) As Integer
Dim *devNumber* As Integer
Dim *mode* As Integer

BASIC

DAS800GetADGainMode% (*devNumber*, *mode*)
Dim *devNumber* As Integer
Dim *mode* As Integer

Entry Parameters

devNumber	Board number. Valid values: **0** to **3**

Exit Parameters

mode	Input range type. Value stored: **0** = Unipolar 1 = Bipolar

Notes

For the board specified by *devNumber*, this function gets the current input range type and stores it in *mode*.

Example

You want to store the current input range type for board 1 in a variable called ADMode1.

C

```
short ADMode1;
err = DAS800_GetADGainMode (1, &ADMode1);
```

Pascal

```
err : = DAS800_GetADGainMode (1, ADMode1);
```

Visual Basic for Windows

```
errnum = DAS800_GetADGainMode% (1, ADMode1)
```

BASIC

```
errnum = DAS800GetADGainMode% (1, ADMode1)
```

DAS800_GetDevHandle

Purpose

Initializes a DAS-800 Series board.

Syntax

C

DAS800_GetDevHandle (*devNumber*, *devHandle*);
short *devNumber*;
DDH **devHandle*;

Pascal

DAS800_GetDevHandle (*devNumber*, *devHandle*) : Word;
devNumber : Integer;
devHandle : Longint;

Visual Basic for Windows

DAS800_GetDevHandle (*devNumber*, *devHandle*) As Integer
Dim *devNumber* As Integer
Dim *devHandle* As Long

BASIC

DAS800GetDevHandle% (*devNumber*, *devHandle*)
Dim *devNumber* As Integer
Dim *devHandle* As Long

Entry Parameters

devNumber	Board number. Valid values: **0** to **3**

Exit Parameters

devHandle	Handle associated with the board.

Notes

This function initializes the board specified by *devNumber*, and stores the board handle of the specified board in *devHandle*.

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

Example

You want to initialize board 1 and to associate board 1 with a board handle called BrdHd1.

C

```
DDH BrdHd1;
err = DAS800_GetDevHandle (1, &BrdHd1);
```

Pascal

```
err : = DAS800_GetDevHandle (1, BrdHd1);
```

Visual Basic for Windows

```
errnum = DAS800_GetDevHandle (1, BrdHd1)
```

BASIC

```
errnum = DAS800GetDevHandle% (1, BrdHd1)
```

DAS800_Get8254

Purpose

Gets status of the 8254 counter/timer circuitry.

Syntax

C

DAS800_Get8254 (*devNumber*, *counter*, *mode*, *count*);
short *devNumber*;
short *counter*;
short **mode*;
unsigned long **count*;

Pascal

DAS800_Get8254 (*devNumber*, *counter*, *mode*, *count*) : Word;
devNumber : Integer;
counter : Integer;
mode : Integer;
count : Longint;

Visual Basic for Windows

DAS800_Get8254 (*devNumber*, *counter*, *mode*, *count*) As Integer
Dim *devNumber* As Integer
Dim *counter* As Integer
Dim *mode* As Integer
Dim *count* As Long

BASIC

DAS800Get8254% (*devNumber*, *counter*, *mode*, *count*)
Dim *devNumber* As Integer
Dim *counter* As Integer
Dim *mode* As Integer
Dim *count* As Long

Entry Parameters

devNumber — Board number.
Valid values: **0** to **3**

counter — Counter/timer.
Valid values: **0** = C/T0
1 = C/T1
2 = C/T2

Exit Parameters

mode — Counter/timer mode.
Value stored: **0** = Pulse on terminal count
1 = Programmable one-shot
2 = Rate generator
3 = Square-wave generator
4 = Software-triggered strobe
5 = Hardware-triggered strobe

count — Value of counter/timer.
Value stored: **0** to **65535**

Notes

For the counter/timer specified by *counter* on the 8254 counter/timer circuitry on the board specified by *devNumber*, this function stores the counter/timer mode in *mode* and the current value of the counter/timer in *count*.

Refer to the *DAS-800 Series User's Guide* for an explanation of the counter/timer modes.

Example

You want to store the counter/timer mode of C/T0 on board 1 in a variable called CT0Mode and the value currently loaded in C/T0 on board 1 in a variable called CT0Count.

C

```
short CT0Mode;
unsigned long CT0Count;
err = DAS800_Get8254 (1, 0, &CT0Mode, &CT0Count);
```

Pascal

```
err : = DAS800_Get8254 (1, 0, CT0Mode, CT0Count);
```

Visual Basic for Windows

```
errnum = DAS800_Get8254 (1, 0, CT0Mode, CT0Count)
```

BASIC

```
errnum = DAS800Get8254% (1, 0, CT0Mode, CT0Count)
```

DAS800_SetADGainMode

Purpose

Sets the input range type (unipolar or bipolar).

Syntax

C

DAS800_SetADGainMode (*devNumber*, *mode*);
short *devNumber*;
short *mode*;

Pascal

DAS800_SetADGainMode (*devNumber*, *mode*) : Word;
devNumber : Integer;
mode : Integer;

Visual Basic for Windows

DAS800_SetADGainMode (*devNumber*, *mode*) As Integer
Dim *devNumber* As Integer
Dim *mode* As Integer

BASIC

DAS800SetADGainMode% (*devNumber*, *mode*)
Dim *devNumber* As Integer
Dim *mode* As Integer

Entry Parameters

devNumber	Board number. Valid values: **0** to **3**
mode	Input range type. Valid values: **0** = Unipolar **1** = Bipolar

Notes

For the board specified by *devNumber*, this function sets the input range type to *mode*.

This function is appropriate for DAS-801 and DAS-802 boards only. The DAS-800 board is always configured for a ±5 V bipolar analog input range type.

Example

The configuration file for board 1 specifies a bipolar input range type. You want to change the input range type to unipolar.

C

```
err = DAS800_SetADGainMode (1, 0);
```

Pascal

```
err : = DAS800_SetADGainMode (1, 0);
```

Visual Basic for Windows

```
errnum = DAS800_SetADGainMode (1, 0)
```

BASIC

```
errnum = DAS800SetADGainMode% (1, 0)
```

DAS800_Set8254

Purpose

Sets up the 8254 counter/timer circuitry.

Syntax

C

DAS800_Set8254 (*devNumber*, *counter*, *mode*, *count*);
short *devNumber*;
short *counter*;
short *mode*;
unsigned long *count*;

Pascal

DAS800_Set8254 (*devNumber*, *counter*, *mode*, *count*) : Word;
devNumber : Integer;
counter : Integer;
mode : Integer;
count : Longint;

Visual Basic for Windows

DAS800_Set8254 (*devNumber*, *counter*, *mode*, *count*) As Integer
Dim *devNumber* As Integer
Dim *counter* As Integer
Dim *mode* As Integer
Dim *count* As Long

BASIC

DAS800Set8254% (*devNumber*, *counter*, *mode*, *count*)
Dim *devNumber* As Integer
Dim *counter* As Integer
Dim *mode* As Integer
Dim *count* As Long

Entry Parameters

devNumber — Board number.
Valid values: **0** to **3**

counter — Counter/timer.
Valid values: **0** = C/T0
1 = C/T1
2 = C/T2

mode — Counter/timer mode.
Valid values: **0** = Pulse on terminal count
1 = Programmable one-shot
2 = Rate generator
3 = Square-wave generator
4 = Software-triggered strobe
5 = Hardware-triggered strobe

count — Value of counter/timer.
Valid values: **2** to **65535**

Notes

For the counter/timer specified by *counter* on the 8254 counter/timer circuitry on the board specified by *devNumber*, this function sets the counter/timer mode to *mode* and the initial count value to *count*.

Refer to the *DAS-800 Series User's Guide* for an explanation of the counter/timer modes and for more information about the 8254 counter/timer circuitry.

Example

You want to configure C/T0 on board 1 as a software-triggered strobe and load an initial count value of 100 into C/T0.

C

```
err = DAS800_Set8254 (1, 0, 4, 100);
```

Pascal

```
err : = DAS800_Set8254 (1, 0, 4, 100);
```

Visual Basic for Windows

```
errnum = DAS800_Set8254 (1, 0, 4, 100)
```

BASIC

```
errnum = DAS800Set8254% (1, 0, 4, 100)
```

K_ADRead

Purpose

Reads a single analog input value.

Syntax

C

K_ADRead (*devHandle*, *chan*, *gainCode*, *ADvalue*);
DDH *devHandle*;
unsigned char *chan*;
unsigned char *gainCode*;
void **ADvalue*;

Pascal

K_ADRead (*devHandle*, *chan*, *gainCode*, *ADvalue*) : Word;
devHandle : Longint;
chan : Byte;
gainCode : Byte;
ADvalue : Pointer;

Visual Basic for Windows

K_ADRead (*devHandle*, *chan*, *gainCode*, *ADvalue*) As Integer
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *gainCode* As Integer
Dim *ADvalue* As Long

BASIC

KADRead% (*devHandle*, *chan*, *gainCode*, *ADvalue*)
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *gainCode* As Integer
Dim *ADvalue* As Long

Entry Parameters

devHandle	Handle associated with the board.
chan	Analog input channel. Valid values: **0** to **127**

gainCode Gain code.
Valid values:

Gain Code	DAS-801 Gain	DAS-802 Gain
0	1	1
1	0.5	0.5
2	10	2
3	100	4
4	500	8

Exit Parameters *ADvalue* Acquired analog input value.

Notes

This function reads the analog input channel *chan* on the board specified by *devHandle* at the gain represented by *gainCode*, and stores the raw count in *ADvalue*.

The range of valid values for *chan* depends on the number of expansion boards you are using. Refer to page 2-6 for more information.

A gain of 0.5 (*gainCode* = 1) is valid only for boards configured with a bipolar input range type. The DAS-800 board supports a gain of 1 only (*gainCode* must equal 0). Refer to Table 2-2 on page 2-6 for a list of the voltage ranges associated with each gain.

Make sure that the variable used to store *ADvalue* is dimensioned as a 16-bit integer.

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

Example

You want to perform an analog input operation on a DAS-801 board that was assigned the board handle BrdHd1. You want to read the value of the signal connected to analog input channel 3 at a gain of 10 and store the raw count in a variable called Chn3Val.

C

```
short Chn3Val;
err = K_ADRead (BrdHd1, 3, 2, &Chn3Val);
```

Pascal

```
err : = K_ADRead (BrdHd1, 3, 2, Chn3Val);
```

Visual Basic for Windows

```
errnum = K_ADRead (BrdHd1, 3, 2, Chn3Val)
```

BASIC

```
errnum = KADRead% (BrdHd1, 3, 2, Chn3Val)
```

K_BufListAdd

Purpose

Adds a buffer to the list of multiple buffers.

Syntax

C

K_BufListAdd (*frameHandle*, *acqBuf*, *samples*);
FRAMEH *frameHandle*;
void **acqBuf*;
DWORD *samples*;

Pascal

K_BufListAdd (*frameHandle*, *acqBuf*, *samples*) : Word;
frameHandle : Longint;
acqBuf : Pointer;
samples : Longint;

Visual Basic for Windows

K_BufListAdd (*frameHandle*, *acqBuf*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *acqBuf* As Long
Dim *samples* As Long

BASIC

KBufListAdd% (*frameHandle*, *acqBuf*, *samples*)
Dim *frameHandle* As Long
Dim *acqBuf* As Long
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
acqBuf	Starting address of buffer.
samples	Number of samples in the buffer.

Notes

For the operation defined by *frameHandle*, this function adds the buffer at the address pointed to by *acqBuf* to the list of multiple buffers; the number of samples in the buffer is specified in *samples*.

You must either allocate the buffer dynamically using **K_IntAlloc** or dimension the buffer locally before you add the buffer to the multiple-buffer list.

Make sure that you add buffers to the multiple-buffer list in the order in which you want to use them. The first buffer you add is Buffer 1, the second buffer you add is Buffer 2, and so on. You can add up to 50 buffers. For interrupt-mode operations, you can use **K_IntStatus** to determine which buffer is currently in use; refer to page 4-83 for more information.

Example

You allocated a 1000-sample buffer to store data for an analog input operation defined by the frame ADFrame1; the buffer starts at the memory location pointed to by Buffer. You want to add this buffer to the list of multiple buffers.

C

```
err = K_BufListAdd (ADFrame1, Buffer, 1000);
```

Pascal

```
err : = K_BufListAdd (ADFrame1, Buffer, 1000);
```

Visual Basic for Windows

```
errnum = K_BufListAdd (ADFrame1, Buffer, 1000)
```

BASIC

```
errnum = KBufListAdd% (ADFrame1, Buffer, 1000)
```

K_BufListReset

Purpose

Clears the list of multiple buffers.

Syntax

C
K_BufListReset (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_BufListReset (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_BufListReset (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KBufListReset% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to the frame that defines the A/D operation.

Notes

For the operation defined by *frameHandle*, this function clears all buffers from the list of multiple buffers.

This function does not deallocate the buffers in the list. If dynamically allocated buffers are no longer needed, you can use **K_IntFree** to free the buffers. Refer to page 4-80 for more information.

Example

You want to clear all buffers from the multiple-buffer list associated with the analog input operation defined by the frame ADFrame1.

C

```
err = K_BufListReset (ADFrame1);
```

Pascal

```
err : = K_BufListReset (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_BufListReset (ADFrame1)
```

BASIC

```
errnum = KBufListReset% (ADFrame1)
```

K_ClearFrame

Purpose Sets the elements of a frame to their default values.

Syntax

C
K_ClearFrame (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_ClearFrame (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_ClearFrame (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KClearFrame% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters *frameHandle* Handle to the frame that defines the A/D operation.

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

Refer to Table 3-1 on page 3-3 for a list of the default values for the elements of an A/D frame.

Example

You want to return all the elements of an A/D frame called ADFrame1 to their default values.

C

```
err = K_ClearFrame (ADFrame1);
```

Pascal

```
err : = K_ClearFrame (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_ClearFrame (ADFrame1)
```

BASIC

```
errnum = KClearFrame% (ADFrame1)
```

K_CloseDriver

Purpose

Closes a previously initialized DAS Function Call Driver.

Syntax

C

K_CloseDriver (*driverHandle*);
DWORD *driverHandle*;

Pascal (Windows Only)

K_CloseDriver (*driverHandle*) : Word;
driverHandle : Longint;

Visual Basic for Windows

K_CloseDriver (*driverHandle*) As Integer
Dim *driverHandle* As Long

Entry Parameters

driverHandle Driver handle you want to free.

Notes

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

If *driverHandle* 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).

You cannot use this function in BASIC or Borland Turbo Pascal for DOS.

Example

You have already initialized the DAS-800 Series Function Call Driver and associated it with a driver handle called Drv800 and now want to reinitialize the driver according to a different configuration file. You want to first close 800Drv1 to free the memory used by Drv800 for another use.

C

```
err = K_CloseDriver (Drv800);
```

Pascal (Windows Only)

```
err : = K_CloseDriver (Drv800);
```

Visual Basic for Windows

```
errnum = K_CloseDriver (Drv800)
```

K_ClrContRun

Purpose

Specifies single-cycle buffering mode.

Syntax

C
K_ClrContRun (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_ClrContRun (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_ClrContRun (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KClrContRun% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to the frame that defines the A/D operation.

Notes

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

Refer to page 2-16 for more information about buffering modes.

The Buffering Mode element is meaningful for interrupt operations only.

Example

You want to specify single-cycle buffering mode for the analog input operation defined by a frame called ADFrame1.

C

```
err = K_ClrContRun (ADFrame1);
```

Pascal

```
err : = K_ClrContRun (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_ClrContRun (ADFrame1)
```

BASIC

```
errnum = KClrContRun% (ADFrame1)
```

K_DASDevInit

Purpose Reinitializes a board.

Syntax

C
K_DASDevInit (*devHandle*);
DDH *devHandle*;

Pascal
K_DASDevInit (*devHandle*) : Word;
devHandle : Longint;

Visual Basic for Windows
K_DASDevInit (*devHandle*) As Integer
Dim *devHandle* As Long

BASIC
KDASDevInit% (*devHandle*)
Dim *devHandle* As Long

Entry Parameters *devHandle* Handle associated with the board.

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

Example You want to reinitialize the board associated with a board handle called BrdHd1.

C
err = K_DASDevInit (BrdHd1);

Pascal
err : = K_DASDevInit (BrdHd1);

Visual Basic for Windows
errnum = K_DASDevInit (BrdHd1)

BASIC
errnum = KDASDevInit% (BrdHd1)

K_DIRead

Purpose

Reads a single digital input value.

Syntax

C

K_DIRead (*devHandle*, *chan*, *DIvalue*);
DDH *devHandle*;
unsigned char *chan*;
void **DIvalue*;

Pascal

K_DIRead (*devHandle*, *chan*, *DIvalue*) : Word;
devHandle : Longint;
chan : Byte;
DIvalue : Pointer;

Visual Basic for Windows

K_DIRead (*devHandle*, *chan*, *DIvalue*) As Integer
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *DIvalue* As Long

BASIC

KDIRead% (*devHandle*, *chan*, *DIvalue*)
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *DIvalue* As Long

Entry Parameters

devHandle	Handle associated with the board.
chan	Digital input channel. Valid value: **0**

Exit Parameters

DIvalue	Digital input value.

Notes

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

DIvalue is a 32-bit variable. The acquired digital value is stored in bits 0, 1, and 2; the values in the remaining bits of *DIvalue* are not defined. Refer to page 2-24 for more information.

Example

You want to perform a digital input operation on a board that was assigned the board handle BrdHd1. You want to read the value of all the bits in digital input channel 0 and store the value in a variable called DIVal.

C

```
long DIVal;
err = K_DIRead (BrdHd1, 0, &DIVal);
```

Pascal

```
err : = K_DIRead (BrdHd1, 0, DIVal);
```

Visual Basic for Windows

```
errnum = K_DIRead (BrdHd1, 0, DIVal)
```

BASIC

```
errnum = KDIRead% (BrdHd1, 0, DIVal)
```

K_DOWrite

Purpose

Writes a single digital output value.

Syntax

C

K_DOWrite (*devHandle*, *chan*, *DOvalue*);
DDH *devHandle*;
unsigned char *chan*;
long *DOvalue*;

Pascal

K_DOWrite (*devHandle*, *chan*, *DOvalue*) : Word;
devHandle : Longint;
chan : Byte;
DOvalue : Longint;

Visual Basic for Windows

K_DOWrite (*devHandle*, *chan*, *DOvalue*) As Integer
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *DOvalue* As Long

BASIC

KDOWrite% (*devHandle*, *chan*, *DOvalue*)
Dim *devHandle* As Long
Dim *chan* As Integer
Dim *DOvalue* As Long

Entry Parameters

devHandle	Handle associated with the board.
chan	Digital output channel. Valid value: **0**
DOvalue	Digital output value. Valid values: **0** to **15**

Notes

This function writes the value *DOvalue* to the digital output channel lines on the board specified by *devHandle*.

DOvalue is a 32-bit variable. The value written is stored in bits 0, 1, 2, and 3; the values in the remaining bits of *DOvalue* are not defined. Refer to page 2-25 for more information.

If you are using an expansion board for an analog input operation, you cannot use this function because the driver uses all four digital output lines to specify the expansion board channel that is acquiring data.

Example

You want to perform a digital output operation on a board that was assigned the board handle BrdHd1. To force the output high on OP1 and OP2 and low on OP3 and OP4, you must write a value of 3 (00..00011) to the digital output lines.

C

```
err = K_DOWrite (BrdHd1, 0, 3);
```

Pascal

```
err : = K_DOWrite (BrdHd1, 0, 3);
```

Visual Basic for Windows

```
errnum = K_DOWrite (BrdHd1, 0, 3)
```

BASIC

```
errnum = KDOWrite% (BrdHd1, 0, 3)
```

K_FormatChanGAry

Purpose

Converts the format of a channel-gain list.

Syntax

Visual Basic For Windows

K_FormatChanGAry (*chanGainArray*) As Integer
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

BASIC

KFormatChanGAry% (*chanGainArray*)
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

Entry Parameters

chanGainArray(0) Channel-gain list starting address.

Notes

This function converts a channel-gain list created in BASIC or Visual Basic for Windows using double-byte (16-bit) values to a channel-gain list of single-byte (8-bit) values that the **K_SetChnGAry** function can use.

After you use this function, your program can no longer read the converted list. You must use the **K_RestoreChanGAry** function to return the list to its original format. Refer to page 4-92 for more information.

Example

You created a channel-gain list in BASIC and named it CGList. You want to convert the channel-gain list to single-byte values.

Visual Basic For Windows

```
errnum = K_FormatChanGAry (CGList(0))
```

BASIC

```
errnum = KFormatChanGAry% (CGList(0))
```

K_FreeDevHandle

Purpose

Frees a previously specified board handle.

Syntax

C
K_FreeDevHandle (*devHandle*);
DWORD *devHandle*;

Pascal (Windows Only)
K_FreeDevHandle (*devHandle*) : Word;
devHandle : Longint;

Visual Basic for Windows
K_FreeDevHandle (*devHandle*) As Integer
Dim *devHandle* As Long

Entry Parameters

devHandle Board handle you want to free.

Notes

This function frees the board handle specified by *devHandle*. This function also frees all frame handles associated with *devHandle*.

You cannot use this function in BASIC or Borland Turbo Pascal for DOS.

Example

You have initialized your DAS-801 board 1 and associated it with a board handle called BrdHdl. You now want to free the board handle so that it can be used again.

C
err = K_FreeDevHandle (BrdHdl);

Pascal
err : = K_FreeDevHandle (BrdHdl);

Visual Basic for Windows
errnum = K_FreeDevHandle (BrdHdl)

BASIC
errnum = KFreeDevHandle% (BrdHdl)

K_FreeFrame

Purpose

Frees a frame.

Syntax

C
K_FreeFrame (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_FreeFrame (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_FreeFrame (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KFreeFrame% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to frame you want to free.

Notes

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

Example

You want to perform an analog input operation, but no frames are available. The analog input operation defined by the frame ADFrame1 is complete. You can free ADFrame1 and redefine it for your new operation.

C
```
err = K_FreeFrame (ADFrame1);
```

Pascal
```
err : = K_FreeFrame (ADFrame1);
```

Visual Basic for Windows
```
errnum = K_FreeFrame (ADFrame1)
```

BASIC
```
errnum = KFreeFrame% (ADFrame1)
```

K_GetADFrame

Purpose

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

Prototype

C

K_GetADFrame (*devHandle*, *frameHandle*);
DDH *devHandle*;
FRAMEH **frameHandle*;

Pascal

K_GetADFrame (*devHandle*, *frameHandle*) : Word;
devHandle : Longint;
frameHandle : Longint;

Visual Basic for Windows

K_GetADFrame (*devHandle*, *frameHandle*) As Integer
Dim *devHandle* As Long
Dim *frameHandle* As Long

BASIC

KGetADFrame% (*devHandle*, *frameHandle*)
Dim *devHandle* As Long
Dim *frameHandle* As Long

Entry Parameters

devHandle	Handle associated with the board.

Exit Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Notes

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

Example

You want to perform a frame-based analog input operation on a board that was assigned the board handle BrdHd1 and assign the frame handle ADFrame1 to the frame that will define the operation.

C

```
FRAMEH ADFrame1;
err = K_GetADFrame (BrdHd1, &ADFrame1);
```

Pascal

```
err : = K_GetADFrame (BrdHd1, ADFrame1);
```

Visual Basic for Windows

```
errnum = K_GetADFrame (BrdHd1, ADFrame1)
```

BASIC

```
errnum = KGetADFrame% (BrdHd1, ADFrame1)
```

K_GetADTrig

Purpose

Reads the current analog trigger conditions.

Syntax

C

K_GetADTrig (*framehandle*, *trigOption*, *chan*, *level*);
FRAMEH *framehandle*;
short **trigOption*;
short **chan*;
long **level*;

Pascal

K_GetADTrig (*frameHandle*, *trigOption*, *chan*, *level*) : Word;
frameHandle : Longint;
trigOption : Word;
chan : Word;
level : Longint;

Visual Basic for Windows

K_GetADTrig (*frameHandle*, *trigOption*, *chan*, *level*) As Integer
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *level* As Long

BASIC

KGetADTrig% (*frameHandle*, *trigOption*, *chan*, *level*)
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *level* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

trigOption	Analog trigger polarity and sense. Value stored: **0** = Positive edge **2** = Negative edge
chan	Analog channel used as trigger channel. Value stored: **0** to **127**

level Level at which the trigger event occurs.
Value stored: **0** to **4095**

Notes

For the operation defined by *frameHandle*, this function stores the channel used for an analog trigger in *chan*, the level used for the analog trigger in *level*, and the trigger polarity and trigger sense in *trigOption*.

The *trigOption* variable contains the value of the Trigger Polarity and Trigger Sense elements.

The *chan* variable contains the value of the Trigger Channel element. The location of the channel stored in *chan* depends on the expansion boards you are using. Refer to page 2-6 for more information.

The *level* variable contains the value of the Trigger Level element. The value of *level* is represented in raw counts. Refer to Appendix B for information on converting the raw count stored in *level* to voltage.

Example

You are using an analog trigger to trigger the analog input operation defined by the frame ADFrame1. You want to store the trigger polarity and sense in a variable called TrigSens, the channel used for the analog trigger in a variable called TrigChan, and the raw count associated with voltage that will trigger the operation in a variable called TrigLvl.

C

```
short TrigSens;
short TrigChan;
long TrigLvl;
err = K_GetADTrig (ADFrame1, &TrigSens, &TrigChan, &TrigLvl);
```

Pascal

```
err : = K_GetADTrig (ADFrame1, TrigSens, TrigChan, TrigLvl);
```

Visual Basic for Windows

```
errnum = K_GetADTrig (ADFrame1, TrigSens, TrigChan, TrigLvl)
```

BASIC

```
errnum = KGetADTrig% (ADFrame1, TrigSens, TrigChan, TrigLvl)
```

K_GetBuf

Purpose

Reads the address of a buffer.

Syntax

C

K_GetBuf (*frameHandle*, *acqBuf*, *samples*);
FRAMEH *frameHandle*;
void **acqBuf*;
long **samples*;

Pascal

K_GetBuf (*frameHandle*, *acqBuf*, *samples*) : Word;
frameHandle : Longint;
acqBuf : Pointer;
samples : Longint;

Visual Basic for Windows

K_GetBuf (*frameHandle*, *acqBuf*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *acqBuf* As Long
Dim *samples* As Long

BASIC

KGetBuf% (*frameHandle*, *acqBuf*, *samples*)
Dim *frameHandle* As Long
Dim *acqBuf* As Long
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

acqBuf	Starting address of buffer.
samples	Number of samples.

Notes

For the operation specified by *frameHandle*, this function stores the address of the currently allocated buffer in *acqBuf* and the number of samples stored in the buffer in *samples*.

Use this function to read the address of a single buffer whose address was specified by **K_SetBuf** or **K_SetBufI**.

The *acqBuf* variable contains the value of the Buffer element.

The *samples* variable contains the value of the Number of Samples element.

Example

You defined an analog input operation in a frame called ADFrame1. You want to store the starting address of the buffer used to store the acquired data in a variable called BufAddr and the number of samples acquired in a variable called NumSamps.

C

```
void *BufAddr;
long NumSamps;
err = K_GetBuf (ADFrame1, &BufAddr, &NumSamps);
```

Pascal

```
err : = K_GetBuf (ADFrame1, BufAddr, NumSamps);
```

Visual Basic for Windows

```
errnum = K_GetBuf (ADFrame1, BufAddr, NumSamps)
```

BASIC

```
errnum = KGetBuf% (ADFrame1, BufAddr, NumSamps)
```

K_GetChn

Purpose

Gets a single channel number.

Syntax

C

K_GetChn (*frameHandle*, *chan*);
FRAMEH *frameHandle*;
short **chan*;

Pascal

K_GetChn (*frameHandle*, *chan*) : Word;
frameHandle : Longint;
chan : Word;

Visual Basic for Windows

K_GetChn (*frameHandle*, *chan*) As Integer
Dim *frameHandle* As Long
Dim *chan* As Integer

BASIC

KGetChn% (*frameHandle*, *chan*)
Dim *frameHandle* As Long
Dim *chan* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

chan	Channel on which to perform operation. Value stored: **0** to **127**

Notes

For the operation defined by *frameHandle*, this function stores the single channel number in *chan*.

The *chan* variable contains the value of the Start Channel element. The location of the channel stored in *chan* depends on the expansion boards you are using. Refer to page 2-6 for more information.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the number of the channel on which you are acquiring data in a variable called SingChan.

C

```
short SingChan;
err = K_GetChn (ADFrame1, &SingChan);
```

Pascal

```
err : = K_GetChn (ADFrame1, SingChan);
```

Visual Basic for Windows

```
errnum = K_GetChn (ADFrame1, SingChan)
```

BASIC

```
errnum = KGetChn% (ADFrame1, SingChan)
```

K_GetChnGAry

Purpose

Gets the starting address of a channel-gain list.

Syntax

C

K_GetChnGAry (*frameHandle*, *chanGainArray*);
FRAMEH *frameHandle*;
void **chanGainArray*;

Pascal

K_GetChnGAry (*frameHandle*, *chanGainArray*) : Word;
frameHandle : Longint;
chanGainArray : Longint;

Visual Basic for Windows

K_GetChnGAry (*frameHandle*, *chanGainArray*) As Integer
Dim *frameHandle* As Long
Dim *chanGainArray* As Long

BASIC

KGetChnGAry% (*frameHandle*, *chanGainArray*)
Dim *frameHandle* As Long
Dim *chanGainArray* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

chanGainArray	Channel-gain list starting address.

Notes

For the operation defined by *frameHandle*, this function stores the starting address of the channel-gain list in *chanGainArray*.

The *chanGainArray* variable contains the value of the Channel-Gain List element.

Refer to page 2-9 for information on setting up a channel-gain list.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the starting address of the channel-gain list in a variable called AryAddr.

C

```
err = K_GetChnGAry (ADFrame1, AryAddr);
```

Pascal

```
err : = K_GetChnGAry (ADFrame1, AryAddr);
```

Visual Basic for Windows

```
errnum = K_GetChnGAry (ADFrame1, AryAddr)
```

BASIC

```
errnum = KGetChnGAry% (ADFrame1, AryAddr)
```

K_GetClk

Purpose

Gets the conversion clock source.

Syntax

C

K_GetClk (*frameHandle*, *clkSource*);
FRAMEH *frameHandle*;
short **clkSource*;

Pascal

K_GetClk (*frameHandle*, *clkSource*) : Word;
frameHandle : Longint;
clkSource : Word;

Visual Basic for Windows

K_GetClk (*frameHandle*, *clkSource*) As Integer
Dim *frameHandle* As Long
Dim *clkSource* As Integer

BASIC

KGetClk% (*frameHandle*, *clkSource*)
Dim *frameHandle* As Long
Dim *clkSource* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

clkSource	Conversion clock source. Value stored: **0** = Internal **1** = External

Notes

For the operation defined by *frameHandle*, this function stores the conversion clock source in *clkSource*.

An internal clock source is the 1 MHz time base of the 8254 counter/timer circuitry; an external clock source is an external signal connected to the INT_IN / XCLK pin. Refer to page 2-13 for more information about conversion clock sources.

The *clkSource* variable contains the value of the Conversion Clock Source element.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the conversion clock source in a variable called Clock.

C

```
short Clock;
err = K_GetClk (ADFrame1, &Clock);
```

Pascal

```
err : = K_GetClk (ADFrame1, Clock);
```

Visual Basic for Windows

```
errnum = K_GetClk (ADFrame1, Clock)
```

BASIC

```
errnum = KGetClk% (ADFrame1, Clock)
```

K_GetClkRate

Purpose

Gets the clock rate (conversion frequency).

Syntax

C

K_GetClkRate (*frameHandle*, *clkTicks*);
FRAMEH *frameHandle*;
long **clkTicks*;

Pascal

K_GetClkRate (*frameHandle*, *clkTicks*) : Word;
frameHandle : Longint;
clkTicks : Longint;

Visual Basic for Windows

K_GetClkRate (*frameHandle*, *clkTicks*) As Integer
Dim *frameHandle* As Long
Dim *clkTicks* As Long

BASIC

KGetClkRate% (*frameHandle*, *clkTicks*)
Dim *frameHandle* As Long
Dim *clkTicks* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

clkTicks	Number of clock ticks between conversions. Value stored: **25** to **65,535** (normal mode) **25** to **4,294,967,295** (cascaded mode)

Notes

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

The *clkTicks* variable contains the value of the Conversion Frequency element.

This function applies to an internal clock source only.

After a synchronous or interrupt operation, the value stored in *clkTicks* represents the actual count, not necessarily the count set by **K_SetClkRate**. The counts are different if you use cascaded mode and specify a count in **K_SetClkRate** that cannot be divided between C/T1 and C/T2; in this case, the driver loads C/T1 and C/T2 as accurately as possible.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the number of clock ticks between conversions in a variable called Ticks.

C

```
long Ticks;
err = K_GetClkRate (ADFrame1, &Ticks);
```

Pascal

```
err : = K_GetClkRate (ADFrame1, Ticks);
```

Visual Basic for Windows

```
errnum = K_GetClkRate (ADFrame1, Ticks)
```

BASIC

```
errnum = KGetClkRate% (ADFrame1, Ticks)
```

K_GetContRun

Purpose

Gets the buffering mode.

Syntax

C

K_GetContRun (*frameHandle*, *mode*);
FRAMEH *frameHandle*;
short **mode*;

Pascal

K_GetContRun (*frameHandle*, *mode*) : Word;
frameHandle : Longint;
mode : Word;

Visual Basic for Windows

K_GetContRun (*frameHandle*, *mode*) As Integer
Dim *frameHandle* As Long
Dim *mode* As Integer

BASIC

KGetContRun% (*frameHandle*, *mode*)
Dim *frameHandle* As Long
Dim *mode* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

mode	Buffering mode. Value stored: **0** = Single-cycle 1 = Continuous

Notes

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

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

Refer to page 2-16 for a description of buffering modes.

The Buffering Mode element is meaningful for interrupt operations only.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the buffering mode in a variable called BufMode.

C

```
short BufMode;
err = K_GetContRun (ADFrame1, &BufMode);
```

Pascal

```
err : = K_GetContRun (ADFrame1, BufMode);
```

Visual Basic for Windows

```
errnum = K_GetContRun (ADFrame1, BufMode)
```

BASIC

```
errnum = KGetContRun% (ADFrame1, BufMode)
```

K_GetDevHandle

Purpose

Initializes any DAS board.

Syntax

C

K_GetDevHandle (*driverHandle*, *devNumber*, *devHandle*);
DWORD *driverHandle*;
WORD *devNumber*;
DDH **devHandle*;

Pascal (Windows Only)

K_GetDevHandle (*driverHandle*, *devNumber*, *devHandle*) : Word;
driverHandle : Longint;
devNumber : Integer;
devHandle : Longint;

Visual Basic for Windows

K_GetDevHandle (*driverHandle*, *devNumber*, *devHandle*) As Integer
Dim *driverHandle* As Long
Dim *devNumber* As Integer
Dim *devHandle* As Long

Entry Parameters

driverHandle	Driver handle of the associated Function Call Driver.
devNumber	Board number. Valid values: **0** to **3**

Exit Parameters

devHandle	Handle associated with the board.

Notes

This function initializes the board associated with *driverHandle* and specified by *devNumber*, and stores the board handle of the specified board in *devHandle*.

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

You cannot use this function in BASIC or Borland Turbo Pascal for DOS.

Example

You want to initialize your DAS-801 board 1, which is associated with the driver handle called Drv800, and associate this board with a board handle called BrdHd1.

C

```
err = K_GetDevHandle (Drv800, 1, &BrdHd1);
```

Pascal

```
err : = K_GetDevHandle (Drv800, 1, BrdHd1);
```

Visual Basic for Windows

```
errnum = K_GetDevHandle (Drv800, 1, BrdHd1)
```

BASIC

```
errnum = KGetDevHandle% (Drv800, 1, BrdHd1)
```

K_GetDITrig

Purpose

Reads the current digital trigger conditions.

Syntax

C

K_GetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*);
FRAMEH *frameHandle*;
short **trigOption*;
short **chan*;
long **pattern*;

Pascal

K_GetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*) : Word;
frameHandle : Longint;
trigOption : Word;
chan : Word;
pattern : Longint;

Visual Basic for Windows

K_GetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*) As Integer
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *pattern* As Long

BASIC

KGetDITrig% (*frameHandle*, *trigOption*, *chan*, *pattern*)
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *pattern* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

trigOption	Trigger polarity and sense. Value stored: **0** = Positive, edge-sensitive
chan	Digital input channel. Value stored: **0**

pattern Trigger pattern.

Notes

For the operation defined by *frameHandle*, this function stores the trigger polarity and sense in *trigOption*, the channel used for the digital trigger in *chan*, and the trigger pattern in *pattern*.

Since the DAS-800 Series Function Call Driver does not currently support digital pattern triggering, the value of *pattern* is meaningless; the *pattern* parameter is provided for future compatibility.

The *trigOption* variable contains the value of the Trigger Polarity and Trigger Sense elements.

The *chan* variable contains the value of the Trigger Channel element.

Example

You are using a digital trigger to trigger the analog input operation defined by the frame ADFrame1. You want to store the trigger polarity and sense in a variable called TrigSens and the channel used for the analog trigger in a variable called TrigChan. (Reserved is a placeholder for the trigger pattern, which is not supported at this time.)

C

```
short TrigSens;
short TrigChan;
long Reserved;
err = K_GetDITrig (ADFrame1, &TrigSens, &TrigChan, &Reserved);
```

Pascal

```
err : = K_GetDITrig (ADFrame1, TrigSens, TrigChan, Reserved);
```

Visual Basic for Windows

```
errnum = K_GetDITrig (ADFrame1, TrigSens, TrigChan, Reserved)
```

BASIC

```
errnum = KGetDITrig% (ADFrame1, TrigSens, TrigChan, Reserved)
```

K_GetErrMsg

Purpose

Gets the address of an error message string.

Syntax

C

K_GetErrMsg (*devHandle*, *msgNum*, *errMsg*);
DDH *devHandle*;
short *msgNum*;
char far **errMsg*;

Entry Parameters

devHandle	Handle associated with the board.
msgNum	Error message number.

Exit Parameters

errMsg	Address of error message string.

Notes

For the board specified by *devHandle*, this function stores the address of the string corresponding to error message number *msgNum* in *errMsg*.

Refer to page 2-30 for more information about error handling. Refer to Appendix A for a list of error codes and their meanings.

This function is available for C only.

Example

You are writing a program in C for a board that was assigned the board handle BrdHd1 and want to store the address of the string corresponding to error message 7801H in a variable called ErrStr.

```
err = K_GetErrMsg (BrdHd1, 0x7801, &ErrStr);
```

K_GetG

Purpose

Gets the gain.

Syntax

C

K_GetG (*frameHandle*, *gainCode*);
FRAMEH *frameHandle*;
short **gainCode*;

Pascal

K_GetG (*frameHandle*, *gainCode*) : Word;
frameHandle : Longint;
gainCode : Word;

Visual Basic for Windows

K_GetG (*frameHandle*, *gainCode*) As Integer
Dim *frameHandle* As Long
Dim *gainCode* As Integer

BASIC

KGetG% (*frameHandle*, *gainCode*)
Dim *frameHandle* As Long
Dim *gainCode* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

Exit Parameters *gainCode* Gain code.
Value stored:

Value of *gainCode*	DAS-801 Gain	DAS-802 Gain
0	1	1
1	0.5	0.5
2	10	2
3	100	4
4	500	8

Notes

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

The *gainCode* variable contains the value of the Gain element.

A gain of 0.5 (*gainCode* = 1) is valid only for boards configured with a bipolar input range type. The DAS-800 board supports a gain of 1 only (*gainCode* must equal 0). Refer to Table 2-2 on page 2-6 for a list of the voltage ranges associated with each gain.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the gain of the channel on which you are acquiring data in a variable called SingGain.

C

```
short SingGain;
err = K_GetG (ADFrame1, &SingGain);
```

Pascal

```
err : = K_GetG (ADFrame1, SingGain);
```

Visual Basic for Windows

```
errnum = K_GetG (ADFrame1, SingGain)
```

BASIC

```
errnum = KGetG% (ADFrame1, SingGain)
```

K_GetGate

Purpose

Gets the status of the hardware gate.

Syntax

C

K_GetGate (*frameHandle*, *gateOpt*);
FRAMEH *frameHandle*;
short **gateOpt*;

Pascal

K_GetGate (*frameHandle*, *gateOpt*) : Word;
frameHandle : Longint;
gateOpt : Integer;

Visual Basic for Windows

K_GetGate (*frameHandle*, *gateOpt*) As Integer
Dim *frameHandle* As Long
Dim *gateOpt* As Integer

BASIC

KGetGate% (*frameHandle*, *gateOpt*)
Dim *frameHandle* As Long
Dim *gateOpt* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

Exit Parameters

gateOpt — Status of the hardware gate.
Value stored: **0** = Disabled
1 = Enabled

Notes

For the operation defined by *frameHandle*, this function stores the status of the hardware gate in *gateOpt*.

The *gateOpt* variable contains the value of the Hardware Gate element.

DAS-800 Series boards support a positive gate only. When the hardware gate is enabled, conversions occur only while the gate signal is high.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the status of the hardware gate in a variable called Gate.

C

```
short Gate;
err = K_GetGate (ADFrame1, &Gate);
```

Pascal

```
err : = K_GetGate (ADFrame1, Gate);
```

Visual Basic for Windows

```
errnum = K_GetGate (ADFrame1, Gate)
```

BASIC

```
errnum = KGetGate% (ADFrame1, Gate)
```

K_GetStartStopChn

Purpose

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

Syntax

C

K_GetStartStopChn (*frameHandle*, *start*, *stop*);
FRAMEH *frameHandle*;
short **start*;
short **stop*;

Pascal

K_GetStartStopChn (*frameHandle*, *start*, *stop*) : Word;
frameHandle : Longint;
start : Word;
stop : Word;

Visual Basic for Windows

K_GetStartStopChn (*frameHandle*, *start*, *stop*) As Integer
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer

BASIC

KGetStartStopChn% (*frameHandle*, *start*, *stop*)
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

start	First channel in a group of consecutive channels. Value stored: **0** to **127**
stop	Last channel in a group of consecutive channels. Value stored: **0** to **127**

Notes

For the operation defined by *frameHandle*, this function stores the first channel in a group of consecutive channels in *start* and the last channel in the group of consecutive channels in *stop*.

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

The *stop* variable contains the value of the Stop Channel element.

The locations of the channels stored in *start* and *stop* depend on the number of expansion boards you are using. Refer to page 2-6 for more information.

Example

You defined an analog input operation in a frame called ADFrame1. You want to store the first channel in your group of consecutive channels in a variable called First and the last channel in your group of consecutive channels in a variable called Last.

C

```
short First;
short Last;
err = K_GetStartStopChn (ADFrame1, &First, &Last);
```

Pascal

```
err : = K_GetStartStopChn (ADFrame1, First, Last);
```

Visual Basic for Windows

```
errnum = K_GetStartStopChn (ADFrame1, First, Last)
```

BASIC

```
errnum = KGetStartStopChn% (ADFrame1, First, Last)
```

K_GetStartStopG

Purpose

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

Syntax

C

K_GetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*);
FRAMEH *frameHandle*;
short **start*;
short **stop;*
short **gainCode*;

Pascal

K_GetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*) : Word;
frameHandle : Longint;
start : Word;
stop : Word;
gainCode : Word;

Visual Basic for Windows

K_GetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*) As Integer
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer
Dim *gainCode* As Integer

BASIC

KGetStartStopG% (*frameHandle*, *start*, *stop*, *gainCode*)
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer
Dim *gainCode* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

Exit Parameters

start — First channel in a group of consecutive channels.
Value stored: **0** to **127**

stop — Last channel in a group of consecutive channels.
Value stored: **0** to **127**

gainCode Gain code.
Value stored:

Value of *gainCode*	DAS-801 Gain	DAS-802 Gain
0	1	1
1	0.5	0.5
2	10	2
3	100	4
4	500	8

Notes

For the operation defined by *frameHandle*, this function stores the first channel in a group of consecutive channels in *start*, the last channel in the group of consecutive channels in *stop*, and the gain code for all channels in the group in *gainCode*.

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

The *stop* variable contains the value of the Stop Channel element.

The locations of the channels stored in *start* and *stop* depend on the number of expansion boards you are using. Refer to page 2-6 for more information.

The *gainCode* variable contains the value of the Gain element.

A gain of 0.5 (*gainCode* = 1) is valid only for boards configured with a bipolar input range type. The DAS-800 board supports a gain of 1 only (*gainCode* must equal 0). Refer to Table 2-2 on page 2-6 for a list of the voltage ranges associated with each gain.

Example

You defined an analog input operation in a frame called ADFrame1. You want to store the first channel in your group of consecutive channels in a variable called First, the last channel in your group of consecutive channels in a variable called Last, and the gain for all channels in the group in a variable called ListGain.

C

```
short First;
short Last;
short ListGain;
err = K_GetStartStopG (ADFrame1, &First, &Last, &ListGain);
```

Pascal

```
err : = K_GetStartStopG (ADFrame1, First, Last, ListGain);
```

Visual Basic for Windows

```
errnum = K_GetStartStopG (ADFrame1, First, Last, ListGain)
```

BASIC

```
errnum = KGetStartStopG% (ADFrame1, First, Last, ListGain)
```

K_GetTrig

Purpose

Gets the trigger source.

Syntax

C

K_GetTrig (*frameHandle*, *trigSource*);
FRAMEH *frameHandle*;
short **trigSource*;

Pascal

K_GetTrig (*frameHandle*, *trigSource*) : Word;
frameHandle : Longint;
trigSource : Word;

Visual Basic for Windows

K_GetTrig (*frameHandle*, *trigSource*) As Integer
Dim *frameHandle* As Long
Dim *trigSource* As Integer

BASIC

KGetTrig% (*frameHandle*, *trigSource*)
Dim *frameHandle* As Long
Dim *trigSource* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

Exit Parameters

trigSource — Trigger source.
Value stored: **0** = Internal trigger
1 = External trigger

Notes

For the operation defined by *frameHandle*, this function stores the trigger source in *trigSource*.

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

An internal trigger is a software trigger; conversions begin when the operation is started. An external trigger is either an analog trigger or a digital trigger; conversions begin when the trigger event occurs. Refer to page 2-16 for more information about internal and external trigger sources.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the source of the trigger that will start the operation in a variable called Trigger.

C

```
short Trigger;
err = K_GetTrig (ADFrame1, &Trigger);
```

Pascal

```
err : = K_GetTrig (ADFrame1, Trigger);
```

Visual Basic for Windows

```
errnum = K_GetTrig (ADFrame1, Trigger)
```

BASIC

```
errnum = KGetTrig% (ADFrame1, Trigger)
```

K_GetTrigHyst

Purpose

Gets the hysteresis value.

Syntax

C

K_GetTrigHyst (*frameHandle*, *hyst*);
FRAMEH *frameHandle*;
short **hyst*;

Pascal

K_GetTrigHyst (*frameHandle*, *hyst*) : Word;
frameHandle : Longint;
hyst : Word;

Visual Basic for Windows

K_GetTrigHyst (*frameHandle*, *hyst*) As Integer
Dim *frameHandle* As Long
Dim *hyst* As Integer

BASIC

KGetTrigHyst% (*frameHandle*, *hyst*)
Dim *frameHandle* As Long
Dim *hyst* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

hyst	Hysteresis value. Value stored: **0** to **4095**

Notes

For the operation defined by *frameHandle*, this function stores the hysteresis value used for an analog trigger in *hyst*. The value is represented in raw counts; refer to Appendix B for information on converting the raw count to voltage.

The *hyst* variable contains the value of the Trigger Hysteresis element.

Refer to page 2-17 for more information about analog triggers.

Example

You defined an analog input operation in a frame called ADFrame1 and want to store the hysteresis value used by the analog trigger in a variable called HystVal.

C

```
short HystVal;
err = K_GetTrigHyst (ADFrame1, &HystVal);
```

Pascal

```
err : = K_GetTrigHyst (ADFrame1, HystVal);
```

Visual Basic for Windows

```
errnum = K_GetTrigHyst (ADFrame1, HystVal)
```

BASIC

```
errnum = KGetTrigHyst% (ADFrame1, HystVal)
```

K_GetVer

Purpose

Gets revision numbers.

Syntax

C

K_GetVer (*devHandle*, *spec*, *version*);
DDH *devHandle*;
short **spec*;
short **version*;

Pascal

K_GetVer (*devHandle*, *spec*, *version*) : Word;
devHandle : Longint;
spec : Word;
version : Word;

Visual Basic for Windows

K_GetVer (*devHandle*, *spec*, *version*) As Integer
Dim *devHandle* As Long
Dim *spec* As Integer
Dim *version* As Integer

BASIC

KGetVer% (*devHandle*, *spec*, *version*)
Dim *devHandle* As Long
Dim *spec* As Integer
Dim *version* As Integer

Entry Parameters

devHandle	Handle associated with the board.

Exit Parameters

spec	Revision number of the Keithley DAS Driver Specification to which the driver conforms.
version	Driver version number.

Notes

For the board specified by *devHandle*, this function stores the revision number of the DAS-800 Series Function Call Driver in *version* and the revision number of the driver specification in *spec*.

The values stored in *spec* and *version* are two-byte (16-bit) integers; the high byte of each contains the major revision level and the low byte of each contains the minor revision level. For example, if the driver version number is 2.1, the major revision level is 2 and the minor revision level is 1; therefore, the high byte of *version* contains the value of **2** (512) and the low byte of *version* contains the value of **1**; the value of both bytes is 513.

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

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

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

Example

You are using functions from different DAS Function Call Drivers in your application program. Before you include a particular function in your program, you want to check the revision of the Function Call Driver associated with a particular board. The board is associated with the board handle BrdHd1. You want to store the revision number of the driver in a variable called Brd1Rev and the revision number of the driver specification in a variable called Brd1Spec.

C

```
short Brd1Spec;
short Brd1Rev;
err = K_GetVer (BrdHd1, &Brd1Spec, &Brd1Rev);
```

Pascal

```
err : = K_GetVer (BrdHd1, Brd1Spec, Brd1Rev);
```

Visual Basic for Windows

```
errnum = K_GetVer (BrdHd1, Brd1Spec, Brd1Rev)
```

BASIC

```
errnum = KGetVer% (BrdHd1, Brd1Spec, Brd1Rev)
```

K_InitFrame

Purpose

Checks the interrupt status.

Syntax

C
K_InitFrame (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_InitFrame (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_InitFrame (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KInitFrame% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to the frame that defines the A/D operation.

Notes

This function checks the status of interrupt operations on the board associated with *frameHandle*.

If no interrupt operation is active, **K_InitFrame** checks the validity of the board associated with *frameHandle* and, if the board is valid, enables A/D operations.

If an interrupt operation is active, **K_InitFrame** returns an error indicating that the board is busy.

Example

You defined an analog input operation in a frame called ADFrame1. ADFrame1 is associated with a board that was assigned the board handle BrdHd1. You want to check the status of interrupt operations on the board before starting a new analog input operation.

C

```
err = K_InitFrame (ADFrame1);
```

Pascal

```
err : = K_InitFrame (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_InitFrame (ADFrame1)
```

BASIC

```
errnum = KInitFrame% (ADFrame1)
```

K_IntAlloc

Purpose

Allocates a buffer.

Syntax

C

K_IntAlloc (*frameHandle*, *samples*, *acqBuf*, *memHandle*);
FRAMEH *frameHandle*;
DWORD *samples*;
void **acqBuf*;
WORD **memHandle*;

Pascal

K_IntAlloc (*frameHandle*, *samples*, *acqBuf*, *memHandle*) : Word;
frameHandle : Longint;
samples : LongInt;
acqBuf : Pointer;
memHandle : Word;

Visual Basic for Windows

K_IntAlloc (*frameHandle*, *samples*, *acqBuf*, *memHandle*) As Integer
Dim *frameHandle* As Long
Dim *samples* As Long
Dim *acqBuf* As Long
Dim *memHandle* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
samples	Number of samples. Valid values: **0** to **32767**

Exit Parameters

acqBuf	Starting address of the allocated buffer.
memHandle	Handle associated with the allocated buffer.

Notes

For the operation defined by *frameHandle*, this function allocates a buffer of the size specified by *samples*, and stores the starting address of the buffer in *acqBuf* and the handle of the buffer in *memHandle*.

Do not use this function for BASIC; for the BASIC languages, you must dimension your buffer locally.

Example

You defined an analog input operation in a frame called ADFrame1. You want to allocate a buffer that will store 1000 samples, store the starting address of this buffer in a variable called Buffer1, and associate this buffer with a memory handle called Handle1.

C

```
err = K_IntAlloc (ADFrame1, 1000, Buffer1, Handle1);
```

Pascal

Refer to page 3-18 for an example of using **K_IntAlloc** in Pascal.

Visual Basic for Windows

```
errnum = K_IntAlloc (ADFrame1, 1000, Buffer1, Handle1)
```

K_IntFree

Purpose

Frees a buffer.

Syntax

C
K_IntFree (*memHandle*);
WORD *memHandle*;

Pascal
K_IntFree (*memHandle*) : Word;
memHandle : Word;

Visual Basic for Windows
K_IntFree (*memHandle*) As Integer
Dim *memHandle* As Integer

Entry Parameters

memHandle Handle to interrupt buffer.

Notes

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

Example

You defined an analog input operation in a frame called ADFrame1 and allocated a buffer associated with the memory handle Handle1. You want to free this buffer for another use.

C
```
err = K_IntFree (Handle1);
```

Pascal
```
err : = K_IntFree (Handle1);
```

Visual Basic for Windows
```
errnum = K_IntFree (Handle1)
```

K_IntStart

Purpose

Starts an interrupt operation.

Syntax

C

K_IntStart (*frameHandle*);
FRAMEH *frameHandle*;

Pascal

K_IntStart (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows

K_IntStart (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC

KIntStart% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to the frame that defines the A/D operation.

Notes

This function starts the interrupt operation defined by *frameHandle*.

Refer to page 3-9 for a discussion of the programming tasks associated with interrupt operations.

Example

You defined an analog input operation in a frame called ADFrame1 and want to start the operation in interrupt mode.

C

```
err = K_IntStart (ADFrame1);
```

Pascal

```
err : = K_IntStart (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_IntStart (ADFrame1)
```

BASIC

```
errnum = KIntStart% (ADFrame1)
```

K_IntStatus

Purpose

Gets status of interrupt operation.

Syntax

C

K_IntStatus (*frameHandle*, *status*, *samples*);
FRAMEH *frameHandle*;
short **status*;
long **samples*;

Pascal

K_IntStatus (*frameHandle*, *status*, *samples*) : Word;
frameHandle : Longint;
status : Word;
samples : Longint;

Visual Basic for Windows

K_IntStatus (*frameHandle*, *status*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *status* As Integer
Dim *samples* As Long

BASIC

KIntStatus% (*frameHandle*, *status*, *samples*)
Dim *frameHandle* As Long
Dim *status* As Integer
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

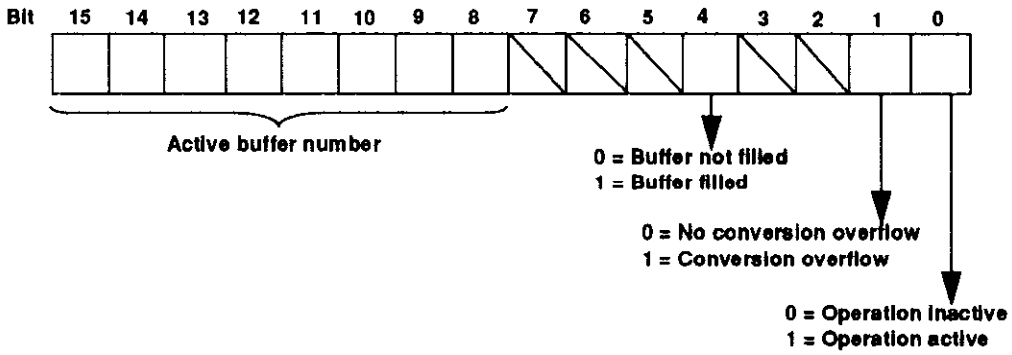
Exit Parameters

status	Status of interrupt operation.
samples	Number of samples that were acquired.

Notes

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

The value stored in *status* depends on the settings in the Status Word, as shown below:



The bits are described as follows:

- Bit 0 indicates whether an interrupt-mode operation is in progress.
- Bit 1 indicates whether a conversion overflow occurred because the transfer of data between the board and the computer's memory was slower than the rate at which the board was acquiring data. When this bit is set, all conversions stop.
- Bit 4 indicates whether the buffer(s) used for an interrupt-mode operation running in continuous buffering mode have been filled. If this bit is set, the buffer(s) have been filled at least once.
- Bits 8 through 15 indicate which buffer in a multiple-buffer list is currently active. To determine the active buffer number, divide the value of the Status word by 256. The first buffer added to the list is Buffer 1, the second buffer added to the list is Buffer 2, and so on.

Example

You defined an analog input operation in a frame called ADFrame1 and started the operation in interrupt mode. You want to store the status of the interrupt operation in a variable called IntStat and the number of samples already acquired in a variable called IntSamp.

C

```
short IntStat;
long IntSamp;
err = K_IntStatus (ADFrame1, &IntStat, &IntSamp);
```

Pascal

```
err : = K_IntStatus (ADFrame1, IntStat, IntSamp);
```

Visual Basic for Windows

```
errnum = K_IntStatus (ADFrame1, IntStat, IntSamp)
```

BASIC

```
errnum = KIntStatus% (ADFrame1, IntStat, IntSamp)
```

K_IntStop

Purpose

Stops an interrupt operation.

Syntax

C

K_IntStop (*frameHandle*, *status*, *samples*);
FRAMEH *frameHandle*;
short **status*;
long **samples*;

Pascal

K_IntStop (*frameHandle*, *status*, *samples*) : Word;
frameHandle : Longint;
status : Word;
samples : Longint;

Visual Basic for Windows

K_IntStop (*frameHandle*, *status*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *status* As Integer
Dim *samples* As Long

BASIC

KIntStop% (*frameHandle*, *status*, *samples*)
Dim *frameHandle* As Long
Dim *status* As Integer
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.

Exit Parameters

status	Status of interrupt operation. Value stored: **0** to **65535**
samples	Number of samples that were acquired.

Notes

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

Refer to page 4-84 for the meaning of the value stored in *status*.

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

Example

You defined an analog input operation in a frame called ADFrame1 and started the operation in interrupt mode. You want to stop the interrupt operation, store the status of the interrupt operation in a variable called IntStat, and store the number of samples already acquired in a variable called IntSamp.

C

```
short IntStat;
long IntSamp;
err = K_IntStop (ADFrame1, &IntStat, &IntSamp);
```

Pascal

```
err : = K_IntStop (ADFrame1, IntStat, IntSamp);
```

Visual Basic for Windows

```
errnum = K_IntStop (ADFrame1, IntStat, IntSamp)
```

BASIC

```
errnum = KIntStop% (ADFrame1, IntStat, IntSamp)
```

K_MoveBufToArray

Purpose

Transfers data from a buffer allocated through **K_IntAlloc** to a locally dimensioned buffer.

Syntax

Visual Basic for Windows

K_MoveBufToArray (*dest*, *source*, *samples*) As Integer
Dim *dest* As Integer
Dim *source* As Long
Dim *samples* As Integer

Entry Parameters

dest	Address of destination buffer.
source	Address of source buffer.
samples	Number of samples to transfer.

Notes

This function transfers the number of samples specified by *samples* from the buffer at address *source* to the buffer at address *dest*.

If the buffer used to store acquired data for your Visual Basic for Windows program was allocated through **K_IntAlloc**, the buffer is not accessible to your program and you must use this function to move the data to an accessible buffer. If the buffer used to store acquired data for your Visual Basic for Windows program was dimensioned locally within the program's memory area, the buffer is accessible to your program and you do not have to use this function. This function is intended for Visual Basic for Windows only, since other languages can access dynamically allocated buffers.

Example

You used **K_IntAlloc** to allocate a buffer to store acquired data for your Visual Basic for Windows program; this buffer starts at the memory location pointed to by AllocBuf. You must move the data to another buffer that is accessible to your program. You want to move 1000 samples from this buffer to another buffer starting at the memory location pointed to by BasicBuf.

Visual Basic for Windows

```
errnum = K_MoveBufToArray (BasicBuf(0), AllocBuf, 1000)
```

K_OpenDriver

Purpose

Initializes any DAS Function Call Driver.

Syntax

C

K_OpenDriver (*deviceName*, *cfgFile*, *driverHandle*);
char **deviceName*;
char **cfgFile*;
DWORD **driverHandle*;

Pascal (Windows Only)

K_OpenDriver (*deviceName*, *cfgFile*, *driverHandle*) : Word;
deviceName : String;
cfgFile : String;
driverHandle : Longint;

Visual Basic for Windows

K_OpenDriver (*deviceName*, *cfgFile*, *driverHandle*) As Integer
Dim *deviceName* As String
Dim *cfgFile* As String
Dim *driverHandle* As Long

Entry Parameters

deviceName	Board name. Valid value: **DAS800** (for DAS-800 Series boards)
cfgFile	Driver configuration file. Valid values: **0** = Current configuration –1 = Default configuration ***filename*** = Any configuration file

Exit Parameters

driverHandle	Handle associated with the driver.

Notes

This function initializes the Function Call Driver for the board associated with *deviceName* according to the information in the configuration file specified by *cfgFile*, and stores the driver handle in *driverHandle*.

You can use this function to initialize the Function Call Driver associated with any DAS board. For DAS-800 Series boards, the string stored in *deviceName* must be DAS800. Refer to other Function Call Driver user's guides for the appropriate string to store in *deviceName* for other DAS boards.

The value stored in *driverHandle* 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 *driverHandle*.

You create a configuration file using the D800CFG.EXE utility. Refer to the *DAS-800 Series User's Guide* for more information.

If *cfgFile* = 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. If *cfgFile* = −1, **K_OpenDriver** initializes the driver to its default configuration; the default configuration is shown in Table 4-2 on page 4-7.

You cannot use this function in BASIC or Borland Turbo Pascal for DOS.

The Function Call Driver requires null terminated strings. To create null terminated strings in Pascal and Visual Basic for Windows, refer to the following examples. These examples assume that the board is a DAS-800 Series board and that the configuration file (*cfgFile*) is DAS800.CFG.

Pascal (Windows Only):

```
deviceName : = 'DAS800' + #0;
cfgFile : = 'DAS800.CFG' + #0;
```

Visual Basic for Windows:

```
deviceName = "DAS800" + CHR$(0)
cfgFile = "DAS800.CFG" + CHR$(0)
```

Example

After you set up your DAS-801 board, you created a configuration file to reflect the settings of the jumper and switches on the board. The configuration file is stored in the memory location pointed to by CONF801. You want to initialize the DAS-800 Series Function Call Driver according to this configuration file and associate the driver with a driver handle called Drv800.

C

```
DWORD 800Drv1;
err = K_OpenDriver (DAS800, CONF801, &Drv800);
```

Pascal (Windows Only)

```
err : = K_OpenDriver ('DAS800' + #0, CONF801[1], Drv800);
```

Visual Basic for Windows

```
errnum = K_OpenDriver ("DAS800" + CHR$(0), CONF801, Drv800)
```

K_RestoreChanGAry

Purpose

Restores a converted channel-gain list.

Syntax

Visual Basic For Windows
K_RestoreChanGAry (*chanGainArray*) As Integer
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

BASIC
KRestoreChanGAry% (*chanGainArray*)
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

Entry Parameters

chanGainArray(0) Channel-gain list starting address.

Notes

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

Refer to page 4-37 for more information about the **K_FormatChanGAry** function.

Example

You created a channel-gain list in BASIC, named it CGList, and then converted it to single-byte values using **K_FormatChanGAry**. You want to restore the channel-gain list to its original format.

Visual Basic For Windows

```
errnum = K_RestoreChanGAry (CGList(0))
```

BASIC

```
errnum = KRestoreChanGAry% (CGList(0))
```

K_SetADTrig

Purpose

Sets up an analog trigger.

Syntax

C

K_SetADTrig (*framehandle, trigOption, chan, level*);
FRAMEH *framehandle*;
short *trigOption*;
short *chan*;
long *level*;

Pascal

K_SetADTrig (*frameHandle, trigOption, chan, level*) : Word;
frameHandle : Longint;
trigOption : Word;
chan : Word;
level : Longint;

Visual Basic for Windows

K_SetADTrig (*frameHandle, trigOption, chan, level*) As Integer
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *level* As Long

BASIC

KSetADTrig% (*frameHandle, trigOption, chan, level*)
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *level* As Long

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

trigOption — Analog trigger polarity and sense.
Valid values: **0** = positive edge
2 = negative edge

chan — Analog channel used as trigger channel.
Valid values: **0** to **127**

level Level at which the trigger event occurs.
Valid values: **0** to **4095**

Notes

For the operation defined by *frameHandle*, this function specifies the channel used for an analog trigger in *chan*, the level used for the analog trigger in *level*, and the trigger polarity and trigger sense in *trigOption*.

The range of valid values for *chan* depends on the number of expansion boards you are using. Refer to page 2-6 for more information.

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

The values you specify set the following elements in the frame identified by *frameHandle*:

- *trigOption* sets the value of the Trigger Polarity and Trigger Sense elements.
- *chan* sets the value of the Trigger Channel element.
- *level* sets the value of the Trigger Level element.

Example

You want to use an analog trigger to trigger the analog input operation defined by the frame ADFrame1. The board is configured for a bipolar input range type. You want to trigger the operation when the signal connected to analog input channel 22 rises above +2 V (positive-edge trigger).

C

```
err = K_SetADTrig (ADFrame1, 0, 22, 2867);
```

Pascal

```
err : = K_SetADTrig (ADFrame1, 0, 22, 2867);
```

Visual Basic for Windows

```
errnum = K_SetADTrig (ADFrame1, 0, 22, 2867)
```

BASIC

```
errnum = KSetADTrig% (ADFrame1, 0, 22, 2867)
```

K_SetBuf

Purpose

Specifies the starting address of a previously allocated or dimensioned buffer.

Syntax

C

K_SetBuf (*frameHandle*, *acqBuf*, *samples*);
FRAMEH *frameHandle*;
void **acqBuf*;
long *samples*;

Pascal

K_SetBuf (*frameHandle*, *acqBuf*, *samples*) : Word;
frameHandle : Longint;
acqBuf : Pointer;
samples : Longint;

Visual Basic for Windows

K_SetBuf (*frameHandle*, *acqBuf*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *acqBuf* As Long
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
acqBuf	Starting address of buffer.
samples	Number of samples.

Notes

For the operation defined by *frameHandle*, this function specifies the starting address of a previously allocated buffer in *acqBuf* and the number of samples stored in the buffer in *samples*.

If you are specifying the starting address of a local memory buffer, make sure that you have dimensioned the buffer as an integer.

Do not use this function for BASIC; for the BASIC languages, use **K_SetBufI**. Refer to page 4-97 for more information.

For C and Pascal application programs, use this function whether you dimensioned your buffer locally or allocated your buffer dynamically using **K_IntAlloc**. 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 3-18 for more information.

For Visual Basic for Windows, use this function only for buffers allocated dynamically using **K_IntAlloc**. For locally dimensioned buffers, use **K_SetBufI**. Refer to page 4-97 for more information.

Do not use this function if you are using multiple buffers. Use **K_BufListAdd** to specify the starting addresses of multiple buffers; refer to page 4-22 for more information.

The syntax of this function in the DAS-800 Series Function Call Driver is slightly different from the syntax of this function in other DAS Function Call Drivers. Therefore, you may have to modify application programs written for other DAS boards before you use them with DAS-800 Series boards.

The values you specify set the following elements in the frame identified by *frameHandle*:

- *acqBuf* sets the value of the Buffer element.
- *samples* sets the value of the Number of Samples element.

Example

You allocated a 1000-sample buffer to store data for an analog input operation defined by the frame ADFrame1; the buffer starts at the memory location pointed to by Buffer. You want to add the starting address of the buffer and the number of samples to the definition of the frame.

C

```
err = K_SetBuf (ADFrame1, Buffer, 1000);
```

Pascal

Refer to page 3-18 for an example of using **K_SetBuf** in Pascal.

Visual Basic for Windows

```
errnum = K_SetBuf (ADFrame1, Buffer, 1000)
```

K_SetBufI

Purpose

Specifies the starting address of a locally dimensioned integer buffer.

Syntax

Visual Basic for Windows

K_SetBufI (*frameHandle*, *acqBuf*, *samples*) As Integer
Dim *frameHandle* As Long
Dim *acqBuf* As Integer
Dim *samples* As Long

BASIC

KSetBufI% (*frameHandle*, *acqBuf*, *samples*)
Dim *frameHandle* As Long
Dim *acqBuf* As Integer
Dim *samples* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
acqBuf	Starting address of the user-dimensioned integer buffer.
samples	Number of samples.

Notes

For the operation defined by *frameHandle*, this function specifies the starting address of a locally dimensioned integer buffer in *acqBuf* and the number of samples stored in the buffer in *samples*.

Do not use this function for C and Pascal; for these languages, use **K_SetBuf**. Refer to page 4-95 for more information.

For Visual Basic for Windows, use this function only for locally dimensioned buffers. For buffers allocated dynamically using **K_IntAlloc**, use **K_SetBuf**. Refer to page 4-95 for more information.

Do not use this function if you are using multiple buffers. Instead, use **K_BufListAdd** to specify the starting addresses of multiple buffers; refer to page 4-22 for more information.

The values you specify set the following elements in the frame identified by *frameHandle*:

- *acqBuf* sets the value of the Buffer element.
- *samples* sets the value of the Number of Samples element.

Example

You dimensioned a 1000-sample local buffer called Buffer to store data for an analog input operation defined by the frame ADFrame1. You want to add the starting address of the buffer and the number of samples to the definition of the frame.

Visual Basic for Windows

```
errnum = K_SetBufI (ADFrame1, Buffer(0), 1000)
```

BASIC

```
errnum = KSetBufI% (ADFrame1, Buffer(0), 1000)
```

K_SetChn

Purpose

Specifies a single channel.

Syntax

C

K_SetChn (*frameHandle*, *chan*);
FRAMEH *frameHandle*;
short *chan*;

Pascal

K_SetChn (*frameHandle*, *chan*) : Word;
frameHandle : Longint;
chan : Word;

Visual Basic for Windows

K_SetChn (*frameHandle*, *chan*) As Integer
Dim *frameHandle* As Long
Dim *chan* As Integer

BASIC

KSetChn% (*frameHandle*, *chan*)
Dim *frameHandle* As Long
Dim *chan* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
chan	Channel on which to perform operation. Valid values: **0** to **127**

Notes

For the operation defined by *frameHandle*, this function specifies the single channel used in *chan*.

The value you specify in *chan* sets the Start Channel element in the frame identified by *frameHandle*.

The range of valid values for *chan* depends on the number of expansion boards you are using. Refer to page 2-6 for more information.

Example

You are defining an analog input operation in a frame called ADFrame1 and want to sample data from analog input channel 16.

C

```
err = K_SetChn (ADFrame1, 16);
```

Pascal

```
err : = K_SetChn (ADFrame1, 16);
```

Visual Basic for Windows

```
errnum = K_SetChn (ADFrame1, 16)
```

BASIC

```
errnum = KSetChn% (ADFrame1, 16)
```

K_SetChnGAry

Purpose

Specifies the starting address of a channel-gain list.

Syntax

C

K_SetChnGAry (*frameHandle*, *chanGainArray*);
FRAMEH *frameHandle*;
void **chanGainArray*;

Pascal

K_SetChnGAry (*frameHandle*, *chanGainArray*) : Word;
frameHandle : Longint;
chanGainArray : Integer;

Visual Basic for Windows

K_SetChnGAry (*frameHandle*, *chanGainArray*) As Integer
Dim *frameHandle* As Long
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

BASIC

KSetChnGAry% (*frameHandle*, *chanGainArray*)
Dim *frameHandle* As Long
Dim *chanGainArray(n)* As Integer
where *n* = (number of channels x 2) + 1

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
chanGainArray	Channel-gain list starting address.

Notes

For the operation defined by *frameHandle*, this function specifies the starting address of the channel-gain list in *chanGainArray*.

The value you specify in *chanGainArray* sets the Channel-Gain List element in the frame identified by *frameHandle*.

Refer to page 2-9 for information on setting up a channel-gain list.

If you created your channel-gain list in BASIC or Visual Basic for Windows, you must use **K_FormatChanGAry** to convert the channel-gain list before you specify the address with **K_SetChnGAry**.

Example

You are defining an analog input operation in a frame called ADFrame1 and want to sample data from the channels in a channel-gain list starting at the memory location pointed to by CGList.

C

```
err = K_SetChnGAry (ADFrame1, CGList);
```

Pascal

Refer to page 3-19 for an example of using **K_SetChnGAry** in Pascal.

Visual Basic for Windows

```
errnum = K_SetChnGAry (ADFrame1, CGList(0))
```

BASIC

```
errnum = KSetChnGAry% (ADFrame1, CGList(0))
```

K_SetClk

Purpose

Specifies the conversion clock source.

Syntax

C

K_SetClk (*frameHandle*, *clkSource*);
FRAMEH *frameHandle*;
short *clkSource*;

Pascal

K_SetClk (*frameHandle*, *clkSource*) : Word;
frameHandle : Longint;
clkSource : Word;

Visual Basic for Windows

K_SetClk (*frameHandle*, *clkSource*) As Integer
Dim *frameHandle* As Long
Dim *clkSource* As Integer

BASIC

KSetClk% (*frameHandle*, *clkSource*)
Dim *frameHandle* As Long
Dim *clkSource* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

clkSource — Conversion clock source.
Valid values: **0** = Internal
1 = External

Notes

For the operation defined by *frameHandle*, this function specifies the conversion clock source in *clkSource*.

The value you specify in *clkSource* sets the Conversion Clock Source element in the frame identified by *frameHandle*.

The internal clock source is the 1 MHz time base of the 8254 counter/timer circuitry; an external clock source is an external signal connected to the INT_IN / XCLK pin. Refer to page 2-13 for more information about conversion clock sources.

Example

You are defining an analog input operation in a frame called ADFrame1 and want to use an external clock to determine the time interval between conversions.

C

```
err = K_SetClk (ADFrame1, 1);
```

Pascal

```
err : = K_SetClk (ADFrame1, 1);
```

Visual Basic for Windows

```
errnum = K_SetClk (ADFrame1, 1)
```

BASIC

```
errnum = KSetClk% (ADFrame1, 1)
```

K_SetClkRate

Purpose

Specifies the clock rate (conversion frequency).

Syntax

C

K_SetClkRate (*frameHandle*, *clkTicks*);
FRAMEH *frameHandle*;
long *clkTicks*;

Pascal

K_SetClkRate (*frameHandle*, *clkTicks*) : Word;
frameHandle : Longint;
clkTicks : Longint;

Visual Basic for Windows

K_SetClkRate (*frameHandle*, *clkTicks*) As Integer
Dim *frameHandle* As Long
Dim *clkTicks* As Long

BASIC

KSetClkRate% (*frameHandle*, *clkTicks*)
Dim *frameHandle* As Long
Dim *clkTicks* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
clkTicks	Number of clock ticks between conversions. Valid values: **25** to **65535** (Normal) **25** to **4,294,967,295** (Cascaded)

Notes

For the operation defined by *frameHandle*, this function specifies the number of clock ticks between conversions in *clkTicks*.

The value you specify in *clkTicks* sets the Conversion Frequency element in the frame identified by *frameHandle*.

This function applies to an internal clock source only.

Example

You are defining an analog input operation in a frame called ADFrame1. C/T2 on your board is configured for normal mode and you are using the internal clock to determine the time interval between conversions. You want to specify a conversion frequency of 25 kHz (40 µs between conversions).

C

```
err = K_SetClkRate (ADFrame1, 40);
```

Pascal

```
err : = K_SetClkRate (ADFrame1, 40);
```

Visual Basic for Windows

```
errnum = K_SetClkRate (ADFrame1, 40)
```

BASIC

```
errnum = KSetClkRate% (ADFrame1, 40)
```

K_SetContRun

Purpose

Specifies continuous buffering mode.

Syntax

C
K_SetContRun (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_SetContRun (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_SetContRun (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KSetContRun% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters

frameHandle Handle to the frame that defines the A/D operation.

Notes

For the operation defined by *frameHandle*, this function sets the buffering mode to continuous mode and sets the Buffering Mode element in the frame accordingly.

Refer to page 2-16 for a description of buffering modes.

The Buffering Mode element is meaningful for interrupt operations only.

Example

You want to specify continuous buffering mode for the analog input operation defined by a frame called ADFrame1.

C

```
err = K_SetContRun (ADFrame1);
```

Pascal

```
err : = K_SetContRun (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_SetContRun (ADFrame1)
```

BASIC

```
errnum = KSetContRun% (ADFrame1)
```

K_SetDITrig

Purpose

Sets up a digital trigger.

Syntax

C

K_SetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*);
FRAMEH *frameHandle*;
short *trigOption*;
short *chan*;
long *pattern*;

Pascal

K_SetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*) : Word;
frameHandle : Longint;
trigOption : Word;
chan : Word;
pattern : Longint;

Visual Basic for Windows

K_SetDITrig (*frameHandle*, *trigOption*, *chan*, *pattern*) As Integer
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *pattern* As Long

BASIC

KSetDITrig% (*frameHandle*, *trigOption*, *chan*, *pattern*)
Dim *frameHandle* As Long
Dim *trigOption* As Integer
Dim *chan* As Integer
Dim *pattern* As Long

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
trigOption	Trigger polarity and sense. Valid value: **0** = Positive, edge-sensitive
chan	Digital input channel. Valid value: **0**

pattern Trigger pattern.

Notes

This function specifies the use of a digital trigger for the operation defined by *frameHandle*.

Since the DAS-800 Series Function Call Driver does not currently support digital pattern triggering, the value of *pattern* is meaningless; the *pattern* parameter is provided for future compatibility.

You cannot set up a digital trigger if the hardware gate is enabled.

The values you specify set the following elements in the frame identified by *frameHandle*:

- *trigOption* sets the value of the Trigger Polarity and Trigger Sense elements.
- *chan* sets the value of the Trigger Channel element.
- *pattern* sets the value of the Trigger Pattern element.

Example

You want to use a digital trigger to trigger the analog input operation defined by the frame ADFrame1.

C

```
err = K_SetDITrig (ADFrame1, 0, 0, 0);
```

Pascal

```
err : = K_SetDITrig (ADFrame1, 0, 0, 0);
```

Visual Basic for Windows

```
errnum = K_SetDITrig (ADFrame1, 0, 0, 0)
```

BASIC

```
errnum = KSetDITrig% (ADFrame1, 0, 0, 0)
```

Purpose

Sets the gain.

Syntax

C

K_SetG (*frameHandle*, *gainCode*);
FRAMEH *frameHandle*;
short *gainCode*;

Pascal

K_SetG (*frameHandle*, *gainCode*) : Word;
frameHandle : Longint;
gainCode : Word;

Visual Basic for Windows

K_SetG (*frameHandle*, *gainCode*) As Integer
Dim *frameHandle* As Long
Dim *gainCode* As Integer

BASIC

KSetG% (*frameHandle*, *gainCode*)
Dim *frameHandle* As Long
Dim *gainCode* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

gainCode — Gain code.
Valid values:

Gain Code	DAS-801 Gain	DAS-802 Gain
0	1	1
1	0.5	0.5
2	10	2
3	100	4
4	500	8

Notes

For the operation defined by *frameHandle*, this function specifies the gain code for a single channel or for a group of consecutive channels in *gainCode*.

A gain of 0.5 (*gainCode* = 1) is valid only for boards configured with a bipolar input range type. The DAS-800 board supports a gain of 1 only (*gainCode* must equal 0). Refer to Table 2-2 on page 2-6 for a list of the voltage ranges associated with each gain.

The value you specify in *gainCode* sets the Gain element in the frame identified by *frameHandle*.

Example

You are defining an analog input operation for a DAS-801 board in a frame called ADFrame1. You want to sample data from a group of consecutive channels and specify a gain of 10 for all channels in the group.

C

```
err = K_SetG (ADFrame1, 2);
```

Pascal

```
err : = K_SetG (ADFrame1, 2);
```

Visual Basic for Windows

```
errnum = K_SetG (ADFrame1, 2)
```

BASIC

```
errnum = KSetG% (ADFrame1, 2)
```

K_SetGate

Purpose

Specifies the status of the hardware gate.

Syntax

C

K_SetGate (*frameHandle*, *gateOpt*);
FRAMEH *frameHandle*;
short *gateOpt*;

Pascal

K_SetGate (*frameHandle*, *gateOpt*) : Word;
frameHandle : Longint;
gateOpt : Integer;

Visual Basic for Windows

K_SetGate (*frameHandle*, *gateOpt*) As Integer
Dim *frameHandle* As Long
Dim *gateOpt* As Integer

BASIC

KSetGate% (*frameHandle*, *gateOpt*)
Dim *frameHandle* As Long
Dim *gateOpt* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

gateOpt — Status of the hardware gate.
Valid values: **0** = Disabled
1 = Enabled

Notes

For the operation defined by *frameHandle*, this function specifies the status of the hardware gate in *gateOpt*.

DAS-800 Series boards support a positive gate only. If you enable the hardware gate, conversions occur while the gate signal is high and are inhibited while the gate signal is low.

You cannot enable the hardware gate if you are using an external digital trigger.

Example

You are defining an analog input interrupt operation in a frame called ADFrame1 and you want to enable the hardware gate.

C

```
err = K_SetGate (ADFrame1, 1);
```

Pascal

```
err : = K_SetGate (ADFrame1, 1);
```

Visual Basic for Windows

```
errnum = K_SetGate (ADFrame1, 1)
```

BASIC

```
errnum = KSetGate% (ADFrame1, 1)
```

K_SetStartStopChn

Purpose

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

Syntax

C

K_SetStartStopChn (*frameHandle*, *start*, *stop*);
FRAMEH *frameHandle*;
short *start*;
short *stop*;

Pascal

K_SetStartStopChn (*frameHandle*, *start*, *stop*) : Word;
frameHandle : Longint;
start : Word;
stop : Word;

Visual Basic for Windows

K_SetStartStopChn (*frameHandle*, *start*, *stop*) As Integer
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer

BASIC

KSetStartStopChn% (*frameHandle*, *start*, *stop*)
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
start	First channel in a group of consecutive channels. Valid values: **0** to **127**
stop	Last channel in a group of consecutive channels. Valid values: **0** to **127**

Notes

For the operation defined by *frameHandle*, this function specifies the first channel in a group of consecutive channels in *start* and the last channel in the group of consecutive channels in *stop*.

The range of valid values for *start* and *stop* depends on the number of expansion boards you are using. Refer to page 2-6 for more information.

The values you specify set the following elements in the frame identified by *frameHandle*:

- *start* sets the value of the Start Channel element.
- *stop* sets the value of the Stop Channel element.

Example

You are defining an analog input operation in a frame called ADFrame1. You want to sample data from channels 2, 3, and 4 in order.

C

```
err = K_SetStartStopChn (ADFrame1, 2, 4);
```

Pascal

```
err : = K_SetStartStopChn (ADFrame1, 2, 4);
```

Visual Basic for Windows

```
errnum = K_SetStartStopChn (ADFrame1, 2, 4)
```

BASIC

```
errnum = KSetStartStopChn% (ADFrame1, 2, 4)
```

K_SetStartStopG

Purpose

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

Syntax

C

K_SetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*);
FRAMEH *frameHandle*;
short *start*;
short *stop*;
short *gainCode*;

Pascal

K_SetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*) : Word;
frameHandle : Longint;
start : Word;
stop : Word;
gainCode : Word;

Visual Basic for Windows

K_SetStartStopG (*frameHandle*, *start*, *stop*, *gainCode*) As Integer
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer
Dim *gainCode* As Integer

BASIC

KSetStartStopG% (*frameHandle*, *start*, *stop*, *gainCode*)
Dim *frameHandle* As Long
Dim *start* As Integer
Dim *stop* As Integer
Dim *gainCode* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
start	First channel in the group of consecutive channels. Valid values: **0** to **127**
stop	Last channel in the group of consecutive channels. Valid values: **0** to **127**

gainCode Gain code.
Valid values:

Gain Code	DAS-801 Gain	DAS-802 Gain
0	1	1
1	0.5	0.5
2	10	2
3	100	4
4	500	8

Notes

For the operation defined by *frameHandle*, this function specifies the first channel in a group of consecutive channels in *start*, the last channel in a group of consecutive channels in *stop*, and the gain code for all channels in the group in *gainCode*.

The range of valid values for *start* and *stop* depends on the number of expansion boards you are using. Refer to page 2-6 for more information.

A gain of 0.5 (*gainCode* = 1) is valid only for boards configured with a bipolar input range type. The DAS-800 board supports a gain of 1 only (*gainCode* must equal 0). Refer to Table 2-2 on page 2-6 for a list of the voltage ranges associated with each gain.

The values you specify set the following elements in the frame identified by *frameHandle*:

- *start* sets the value of the Start Channel element.
- *stop* sets the value of the Stop Channel element.
- *gainCode* sets the value of the Gain element.

Example

You are defining an analog input operation for a DAS-801 board in a frame called ADFrame1. You want to sample data from channels 5, 6, and 7 in order, at a gain of 100 for all channels.

C

```
err = K_SetStartStopG (ADFrame1, 5, 7, 3);
```

Pascal

```
err : = K_SetStartStopG (ADFrame1, 5, 7, 3);
```

Visual Basic for Windows

```
errnum = K_SetStartStopG (ADFrame1, 5, 7, 3)
```

BASIC

```
errnum = KSetStartStopG% (ADFrame1, 5, 7, 3)
```

K_SetTrig

Purpose

Specifies the trigger source.

Syntax

C

K_SetTrig (*frameHandle*, *trigSource*);
FRAMEH *frameHandle*;
short *trigSource*;

Pascal

K_SetTrig (*frameHandle*, *trigSource*) : Word;
frameHandle : Longint;
trigSource : Word;

Visual Basic for Windows

K_SetTrig (*frameHandle*, *trigSource*) As Integer
Dim *frameHandle* As Long
Dim *trigSource* As Integer

BASIC

KSetTrig% (*frameHandle*, *trigSource*)
Dim *frameHandle* As Long
Dim *trigSource* As Integer

Entry Parameters

frameHandle — Handle to the frame that defines the A/D operation.

trigSource — Trigger source.
Valid values: **0** = Internal trigger
1 = External trigger

Notes

For the operation defined by *frameHandle*, this function specifies the trigger source in *trigSource*.

An internal trigger is a software trigger; conversions begin when the operation is started. An external trigger is either an analog trigger or a digital trigger; conversions begin when the trigger event occurs. Refer to page 2-16 for more information about internal and external trigger sources.

If *trigSource* = 1, make sure that you use either **K_SetADTrig** or **K_SetDITrig** to specify whether the external trigger source is an analog trigger or a digital trigger.

Example

You are defining an analog input interrupt operation in a frame called ADFrame1. You want to specify an internal trigger; you want the operation to start as soon as **K_IntStart** is executed.

C

```
err = K_SetTrig (ADFrame1, 0);
```

Pascal

```
err : = K_SetTrig (ADFrame1, 0);
```

Visual Basic for Windows

```
errnum = K_SetTrig (ADFrame1, 0)
```

BASIC

```
errnum = KSetTrig% (ADFrame1, 0)
```

K_SetTrigHyst

Purpose

Specifies the hysteresis value.

Syntax

C

K_SetTrigHyst (*frameHandle*, *hyst*);
FRAMEH *frameHandle*;
short *hyst*;

Pascal

K_SetTrigHyst (*frameHandle*, *hyst*) : Word;
frameHandle : Longint;
hyst : Word;

Visual Basic for Windows

K_SetTrigHyst (*frameHandle*, *hyst*) As Integer
Dim *frameHandle* As Long
Dim *hyst* As Integer

BASIC

KSetTrigHyst% (*frameHandle*, *hyst*)
Dim *frameHandle* As Long
Dim *hyst* As Integer

Entry Parameters

frameHandle	Handle to the frame that defines the A/D operation.
hyst	Hysteresis value. Valid values: **0** to **4095**

Notes

For the operation defined by *frameHandle*, this function specifies the hysteresis value used for an analog trigger in *hyst*. You must specify the hysteresis value in raw counts. Refer to Appendix B for information on converting the hysteresis voltage to a raw count.

Refer to page 2-17 for more information about analog triggers.

The value you specify in *hyst* sets the Trigger Hysteresis element in the frame identified by *frameHandle*.

Example

You want to use an analog trigger to trigger the analog input operation defined by the frame ADFrame1. The board is configured for a unipolar input range type. You used **K_SetADTrig** to specify that you want to trigger the operation when the signal connected to analog input channel 0 rises above +4 V (positive-edge trigger). To prevent noise from causing the trigger event to occur, you want to specify a hysteresis value of 0.1 V to make sure that the analog signal falls below +3.9 V before it rises above +4 V.

C

```
err = K_SetTrigHyst (ADFrame1, 41);
```

Pascal

```
err : = K_SetTrigHyst (ADFrame1, 41);
```

Visual Basic for Windows

```
errnum = K_SetTrigHyst (ADFrame1, 41)
```

BASIC

```
errnum = KSetTrigHyst% (ADFrame1, 41)
```

K_SyncStart

Purpose Starts a synchronous operation.

Syntax

C
K_SyncStart (*frameHandle*);
FRAMEH *frameHandle*;

Pascal
K_SyncStart (*frameHandle*) : Word;
frameHandle : Longint;

Visual Basic for Windows
K_SyncStart (*frameHandle*) As Integer
Dim *frameHandle* As Long

BASIC
KSyncStart% (*frameHandle*)
Dim *frameHandle* As Long

Entry Parameters *frameHandle* Handle to the frame that defines the A/D operation.

Notes This function starts the synchronous operation defined by *frameHandle*.

Refer to page 3-7 for a discussion of the programming tasks associated with synchronous operations.

Example

You defined an analog input operation in a frame called ADFrame1 and want to start the operation in synchronous mode.

C

```
err = K_SyncStart (ADFrame1);
```

Pascal

```
err : = K_SyncStart (ADFrame1);
```

Visual Basic for Windows

```
errnum = K_SyncStart (ADFrame1)
```

BASIC

```
errnum = KSyncStart% (ADFrame1)
```

A

Error/Status Codes

Table A-1 lists the error/status codes that are returned by the DAS-800 Function Call Driver functions, possible causes for error conditions, and possible solutions for resolving error conditions. The error/status codes are returned in hexadecimal format.

If you cannot resolve an error condition, contact the factory.

Table A-1. Error/Status Codes

Error Code	Cause	Solution
0	No error has been detected.	Status only; no action is necessary.
6000	**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 D800CFG.EXE utility to create and modify configuration files.
6001	**Illegal Base Address in Configuration File:** The base address specified in the configuration file is invalid.	Use the D800CFG.EXE utility to change the base address in the configuration file.
6005	**Illegal Channel Number:** The specified channel is out of range.	Specify a legal channel number: Analog input: 0 to 127 Digital input: 0 Digital output: 0
6006	**Illegal Gain:** The gain code specified for an analog input operation is out of range.	Specify a legal gain code: 0 to 5 Refer to Table 2-2 on page 2-6 for more information about the meaning of the gain codes.

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

Error Code	Cause	Solution
6008	**Bad Number in Configuration File**: The configuration file contains a numeric value that is not in the correct format.	Check all numeric entries in the configuration file; make sure that &H precedes hexadecimal numbers. Use the D800CFG.BXB utility to modify the configuration file.
600A	**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.
600C	**Error in Returning Interrupt Buffer**: The memory handle specified in **K_IntFree** is invalid.	Check the memory handle stored by **K_IntAlloc** and make sure that it was not modified.
600D	**Bad 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	**No More Frame Handles**: No frames are left in the pool of available frames.	Use **K_FreeFrame** to free a frame that the application is no longer using.
600F	**Requested Interrupt Buffer Too Large**: The number of samples specified in **K_IntAlloc** is too large.	Specify a smaller number of samples; remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
6010	**Cannot Allocate Interrupt Buffer**: For Windows-based languages only, not enough DOS memory (less than 1 MByte) is available.	Remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
6012	**Interrupt Buffer Deallocation Error**: For Windows-based languages only, an error occurred when **K_IntFree** attempted to free a memory handle.	Remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
602B	**No Room in Heap**: The number of samples you requested in the Keithley Memory Manager is greater than the largest contiguous block available in the reserved heap.	Specify a smaller number of samples; free a previously allocated buffer; use the KMMSETUP utility to expand the reserved heap.

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

Error Code	Cause	Solution
602C	**Number of Samples Too Large**: The number of samples you requested in the Keithley Memory Manager is greater than 65,536.	Specify a value between 0 and 65,536 in the KMMSETUP utility.
6035	**Driver in Use**: The Function Call Driver you are trying to initialize has already been initialized using the specified configuration file.	Wait and try to initialize the driver at a later time. Use **K_OpenDriver** with the *cfgFile* parameter set to 0 to open the driver using the current configuration. Use a different configuration file to initialize the driver.
6036	**Bad 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	**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**.
7000	**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: DAS800, DAS801, DAS802
7001	**Bad Board Name**: The board name in the specified configuration file is illegal.	Specify a legal board name in the configuration file: DAS800, DAS801, DAS802
7002	**Bad Board Number**: The driver initialization function found an illegal board number in the specified configuration file.	Specify a legal board number: 0 to 3
7003	**Bad Base Address**: The driver initialization function found an illegal base address in the specified configuration file.	Specify a base address in the inclusive range 200H (512) to 3F0H (1008) in increments of 8H (8). Make sure that &H precedes hexadecimal numbers.

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

Error Code	Cause	Solution
7004	**Bad Interrupt Level**: The driver initialization function found an illegal interrupt level in the specified configuration file.	Specify a legal interrupt level: 2 to 7, X (disabled)
7005	**Bad Counter Configuration**: The driver initialization function found an illegal counter/timer configuration in the specified configuration file.	Specify a legal counter/timer configuration: cascaded, normal
7006	**Bad A/D Gain Mode**: The driver initialization function found an illegal input range type in the specified configuration file.	Specify a legal A/D mode: bipolar, unipolar
7007	**Bad A/D Channel Configuration**: The driver initialization function found an illegal input configuration in the specified configuration file.	Specify a legal input configuration: single-ended, differential
7008	**Bad Number of EXP-16 Expansion Boards**: The driver initialization function found an illegal number of EXP-16 or EXP-16/A expansion boards in the specified configuration file.	Specify a legal number of EXP-16 or EXP-16/A expansion boards: 1 to 8
7009	**Bad EXP-16 Expansion Board Number**: The driver initialization function found an illegal number assigned to one of the EXP-16 or EXP-16/A expansion boards in the specified configuration file.	Specify a legal number for each EXP-16 or EXP-16/A expansion board: 0 to 7
700A	**Bad EXP-16 Expansion Board Gain**: The driver initialization function found an illegal gain assigned to one of the EXP-16 or EXP-16/A expansion boards in the specified configuration file.	Specify a legal gain value for each EXP-16 or EXP-16/A expansion board: 0.5 to 2000

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

Error Code	Cause	Solution
700B	**Bad Number of EXP-GP Expansion Boards**: The driver initialization function found an illegal number of EXP-GP expansion boards in the specified configuration file.	Specify a legal number of EXP-GP expansion boards: 1 to 8
700C	**Bad EXP-GP Expansion Board Number**: The driver initialization function found an illegal number assigned to one of the EXP-GP expansion boards in the specified configuration file.	Specify a legal number for each EXP-GP expansion board: 0 to 7
700D	**Bad EXP-GP Expansion Board Gain**: The driver initialization function found an illegal gain series assigned to one of the EXP-GP expansion boards in the specified configuration file.	Specify a legal gain series for each EXP-GP expansion board: 1.0 series (1, 10, 100, 1000), 2.5 series (2.5, 25, 250, 2500)
700E	**Bad EXP-GP Expansion Board Channel**: The driver initialization function found an illegal gain assigned to one of the channels on one of the EXP-GP expansion boards in the specified configuration file.	Specify a legal gain for each EXP-GP expansion board channel: 1, 10, 100, 1000 (1.0 series) or 2.5, 25, 250, 2500 (2.5 series)
700F	**Bad 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 value: 1 to 7, -1 (unused)
7800	**Bad Revision Number**: The revision of the driver you are using does not match the revision of the Keithley DAS Driver Specification.	Make sure that you are using the appropriate driver.
7801	**Resource Busy**: You started an operation while an operation of the same type was still in progress.	Wait and try the operation again later.

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

Error Code	Cause	Solution
7803	**Bad Counter Number**: You specified an illegal counter/timer in **DAS800_Set8254.**	Specify a legal counter/timer: 0, 1, 2
7804	**Bad Counter Mode**: You specified an illegal counter/timer mode in **DAS800_Set8254.**	Specify a legal counter/timer mode: 0 to 5 Refer to the *DAS-800 Series User's Guide* for information about the counter/timer modes.
7805	**Bad Counter Count**: You specified an illegal count value in **DAS800_Set8254.**	Specify a legal count value: 2 to 65535
7806	**Illegal Hysteresis**: You specified an illegal hysteresis value.	Make sure that when the hysteresis value is added to or subtracted from the analog trigger level, the resulting value is between 0 and 4095.
7807	**Illegal Board**: You are attempting to program a board that is not a DAS-800 Series board.	Make sure that you are using the appropriate software for the appropriate board.
7808	**Interrupts Not Enabled**: You started an interrupt operation, but interrupts were disabled in the configuration file.	Specify an interrupt level in the configuration file: 2, 3, 4, 5, 6, or 7
7809	**Illegal Digital Trigger**: An illegal trigger polarity and sense value is specified in **K_SetDITrig.**	The trigger polarity and sense value must be 0; only a positive-edge trigger can be used.
780A	**Illegal Gain Mode**: An illegal input range type was specified in **DAS800_SetADGainMode.**	Specify a legal input range type: 0 (unipolar), 1 (bipolar)
780B	**Conversion Underflow**: You attempted to read data, but there was no data to read.	Check your application program.
780C	**Illegal Gate**: You enabled the hardware gate while a digital trigger was enabled.	Disable the digital trigger by selecting an analog trigger (using **K_SetADTrig**) or by specifying an internal trigger in **K_SetTrig**. Do not use the hardware gate at this time.

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

Error Code	Cause	Solution
8001	**Function Not Supported**: You have attempted to use a function not supported by the DAS-800 Series Function Call Driver.	Contact the factory.
8003	**Non Valid Board Number**: An illegal board number was specified in the board initialization function.	Specify a legal board number: 0 to 3
8004	**Non Valid Error Number**: The error message number specified in **K_GetErrMsg** is invalid.	Check the error message number and try again.
8005	**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.
8009	**Digital Output Not Initialized**: You may have expansion boards configured that are using the digital output lines to determine the channel to read.	Disconnect the expansion boards and make the appropriate changes to the configuration file. Do not attempt to use the digital output lines.
800B	**Conversion Overrun**: Data was overwritten before it was transferred to the computer's memory.	Adjust the clock source to slow down the rate at which the board acquires data. Remove other application programs that are running and using computer resources.
801A	**Interrupts Already Active**: You have attempted to start an operation whose interrupt level is being used by another system resource.	Wait and try the operation later.
FFFF	**User Aborted Operation**	You pressed [Ctrl] [Break] during a synchronous-mode operation or while waiting for an analog trigger event to occur.

B

Data Formats

When the DAS-800 Series Function Call driver reads data, it stores the data in the upper 12 bits of a 16-bit integer. Before displaying, printing, or converting the data, you may want to shift the upper 12 bits right by four bits so that the data is right-justified. After shifting, you can AND out the upper four bits to set them to zero. Use one of the following programming lines, where *data* is the value stored by the DAS-800 Series Function Call Driver:

For C: `data = (data>>4) & 0x0FFF`

For Pascal: `data = (data shr 4) and $0FFF`

For BASIC: `data = (data / 16) And &H0FFF`

Note: When you pass analog data to the Function Call Driver, the driver always assumes that the data is a 12-bit, right-justified value. No shifting is required.

The DAS-800 Series Function Call Driver can read and write raw counts only. When reading a value (as in **K_ADRead**), you may want to convert the raw count to a more meaningful voltage value; when writing a value (as in **K_SetTrigHyst**), you must convert the voltage value to a raw count.

The remainder of this appendix contains instructions for converting raw counts to voltage and for converting voltage to raw counts.

Converting Raw Counts to Voltage

You may want to convert raw counts to voltage when reading an analog input value or when reading the analog trigger level or hysteresis value.

To convert a raw count value to voltage, you must first shift the data, as described previously. Then, use one of the following equations, where *count* is the shifted count value, 10 V is the span of the analog input range, 4096 is the number of counts available in 12 bits, *gain* is the gain of the analog input channel, and 2048 is the offset value:

DAS-800

Always bipolar input range type:

$$\text{Voltage} = (\text{count} - 2048) \times \frac{10}{4096}$$

DAS-801 / DAS-802

For unipolar input range type:

$$\text{Voltage} = \left(\text{count} \times \frac{10}{4096}\right) + \text{gain}$$

For bipolar input range type:

$$\text{Voltage} = \left((\text{count} - 2048) \times \frac{10}{4096}\right) + \text{gain}$$

Note: When converting raw counts to voltage to read an analog trigger level or hysteresis value, always use a gain of 1 in your equation, no matter what the gain of the channel is.

For example, assume that you want to read analog input data from a channel on a DAS-801 board configured for a unipolar input range type; the channel collects the data at a gain of 10. The count value after shifting is 3072. The voltage is determined as follows:

$$\left(3072 \times \frac{10}{4096}\right) + 10 = 0.75 \text{ V}$$

As another example, assume that you want to read analog input data from a channel on a DAS-802 board configured for a bipolar input range type; the channel collects the data at a gain of 2. The count value after shifting is 1024. The voltage is determined as follows:

$$\left((1024 - 2048) \times \frac{10}{4096}\right) + 2 = -1.25 \text{ V}$$

Converting Voltage to Raw Counts

You must convert voltage to raw counts when specifying an analog trigger level or hysteresis value. You must specify the voltage value as a 12-bit, right-justified raw count (0 to 4095).

Specifying an Analog Trigger Level

To convert a voltage value to a raw count when specifying an analog trigger level, use one of the following equations, where *voltage* is the desired voltage in volts, 10 V is the span of the analog input range, 4096 is the number of counts available in 12 bits, and *gain* is the gain (always 1 in this case):

DAS-800

Always bipolar input range type:

$$\text{Count} = \frac{\text{voltage} \times 4096}{10} + 2048$$

DAS-801 / DAS-802

For unipolar input range type:

$$\text{Count} = \left(\frac{\text{voltage} \times 4096}{10}\right) \times \text{gain}$$

For bipolar input range type:

$$\text{Count} = \left(\left(\frac{\text{voltage} \times 4096}{10}\right) \times \text{gain}\right) + 2048$$

Note: The driver always interprets the count value you specify for an analog trigger level as based on a gain of 1 (for unipolar input range type, a count of 0 is interpreted as 0 V and a count of 4095 is interpreted as +9.9976 V; for bipolar input range type, a count of 0 is interpreted as –5 V and a count of 4095 is interpreted as +4.9976 V).

For example, assume that you want to specify an analog trigger level of –1.25 V for a channel on a DAS-802 board configured for a bipolar input range type and a gain of 2. The raw count is determined as follows:

$$\left(\left(\frac{-1.25 \times 4096}{10}\right) \times 1\right) + 2048 = 1536$$

Note: No matter what the gain of the channel is, always use a gain of 1 in your equation.

Specifying a Hysteresis Value

To convert a voltage value to a raw count when specifying a hysteresis value, use the following equation, where *voltage* is the desired voltage in volts, 10 V is the span of the analog input range, 4096 is the number of counts available in 12 bits, and *gain* is the gain (always 1 in this case):

$$\text{Count} = \left(\frac{\text{voltage} \times 4096}{10}\right) \times \text{gain}$$

Note: The driver always interprets the count value you specify for a hysteresis value as based on a gain of 1 (the span is 10 V).

For example, assume that you want to specify an analog trigger hysteresis value of 0.05 V for a channel on a DAS-801 board configured for a unipolar input range type and a gain of 10. The raw count is determined as follows:

$$\left(\frac{0.05 \times 4096}{10}\right) \times 1 = 20$$

Note: No matter what the gain of the channel is, always use a gain of 1 in your equation.

Index

A

Accessory boards: *see* Expansion boards
ADC: *see* Analog-to-digital converter
Allocating memory 2-3
    Windows 2-4
Analog input operations 2-1
    programming tasks 3-6
Analog-to-digital converter 2-15
Analog triggers 2-17
    time delays 2-19
Applications Engineering Department 1-6
ASO-800 software package 1-1
    installing from DOS 1-3
    installing from Windows 1-4

B

BASIC
    setting up a channel-gain list 2-11
    specifying the buffer address 2-4, 3-26
    see also Professional Basic, QuickBASIC (Version 4.0), QuickBasic (Version 4.5), Visual Basic for DOS
Bipolar configuration: *see* Input range type
Board handle 2-29
Board initialization 2-29
Borland C/C++
    programming information 3-14
    see also C languages
Borland Turbo Pascal: *see* Turbo Pascal
Borland Turbo Pascal for Windows: *see* Turbo Pascal for Windows
Buffer address 2-4, 3-18, 3-25
Buffer address functions 4-3
Buffering mode functions 4-3
Buffering modes 2-16
Buffers 2-3
    multiple 2-4

C

C languages
    setting up a channel-gain list 2-10
    specifying the buffer address 2-5
    see also Borland C/C++, Microsoft C/C++, QuickC for Windows, Visual C++
Cascaded mode 2-13
Channel and gain functions 4-3
Channel-gain list 2-9, 3-19
Channels
    multiple using a channel-gain list 2-9
    multiple using a group of consecutive channels 2-9
    number supported 2-6
    single 2-8
Clocks: *see* Conversion clocks, External clock source, Internal clock source
Commands: *see* Functions
Common tasks 3-6
Compile and link statements
    Borland C/C++ 3-14
    Microsoft C/C++ 3-13
    Professional Basic 3-22
    QuickBASIC (Version 4.0) 3-20
    QuickBasic (Version 4.5) 3-21
    Turbo Pascal 3-17
Configuration file default values 4-7
Continuous mode 2-16
Conventions 4-5
Conversion clock functions 4-3
Conversion clocks 2-13
Conversion frequency 2-15
Conversion rate: *see* Conversion frequency

Converting
raw counts to voltage B-2
voltage to raw counts B-3
Counter/timer functions 4-4
Counter/timer I/O operations 2-26
Counter/timer modes 2-27
Counter/timers: *see* 8254 counter/timer circuitry
Creating an executable file
Borland C/C++ 3-14
Microsoft C/C++ 3-13
Professional Basic 3-22
QuickBASIC (Version 4.0) 3-20
QuickBasic (Version 4.5) 3-21
QuickC for Windows 3-15
Turbo Pascal 3-17
Turbo Pascal for Windows 3-17
Visual Basic for DOS 3-23
Visual Basic for Windows 3-24

D

DAS800_DevOpen 2-28, 4-6
DAS800_GetADGainMode 2-5, 4-9
DAS800_GetDevHandle 2-29, 4-11
DAS800_Get8254 2-27, 4-13
DAS-800 Series Function Call Driver: *see* Function Call Driver
DAS-800 Series standard software package 1-1
installing 1-2
DAS800_SetADGainMode 2-5, 4-15
DAS800_Set8254 2-27, 4-17
Data formats B-1
Data transfer modes: *see* Operation modes
Default values
configuration file 4-7
frame elements 3-3
Digital I/O operations
input operations 2-24
output operations 2-25
programming tasks 3-12
Digital triggers 2-20
Dimensioning memory 2-3
Driver: *see* Function Call Driver
Driver handle 2-28

E

8254 counter/timer circuitry 2-26
used as internal clock source 2-13
Elements of frame 3-2
Error codes A-1
Error handling 2-30
Executable file: *see* Creating an executable file
Expansion boards 2-6
External clock source 2-14
used in triggered operation 2-19, 2-21
used with hardware gate 2-23
External trigger 2-16

F

Files required
Borland C/C++ 3-14
Microsoft C/C++ 3-13
Professional Basic 3-22, 3-23
QuickBASIC (Version 4.0) 3-20
QuickBasic (Version 4.5) 3-21
QuickC for Windows 3-15
Turbo Pascal 3-16
Turbo Pascal for Windows 3-17
Visual Basic for Windows 3-24
Visual C++ 3-16
Frame management functions 4-2

Frames 3-1
frame elements 3-2
frame handle 3-2
frame types 3-2
Function Call Driver
initialization 2-28
structure 3-1
Functions
buffer address 4-3
buffering mode 4-3
channel and gain 4-3
conversion clock 4-3
counter/timer 4-4
DAS800_DevOpen 2-28, 4-6
DAS800_GetADGainMode 2-5, 4-9
DAS800_GetDevHandle 2-29, 4-11
DAS800_Get8254 2-27, 4-13
DAS800_SetADGainMode 2-5, 4-15
DAS800_Set8254 2-27, 4-17
frame management 4-2
gate 4-4
initialization 4-2
K_ADRead 2-2, 2-8, 4-19
K_BufListAdd 2-4, 2-5, 4-22
K_BufListReset 2-4, 4-24
K_ClearFrame 3-4, 4-26
K_CloseDriver 2-28, 4-28
K_ClrContRun 2-16, 4-30
K_DASDevInit 2-29, 4-32
K_DIRead 2-24, 4-33
K_DOWrite 2-25, 4-35
K_FormatChanGAry 2-12, 4-37
K_FreeDevHandle 2-29, 4-38
K_FreeFrame 3-2, 4-39
K_GetADFrame 3-2, 4-40
K_GetADTrig 4-42
K_GetBuf 4-44
K_GetChn 4-46
K_GetChnGAry 4-48
K_GetClk 4-50
K_GetClkRate 4-52
K_GetContRun 4-54
K_GetDevHandle 2-29, 4-56
K_GetDITrig 4-58
K_GetErrMsg 2-30, 4-60
K_GetG 4-61
K_GetGate 4-63
K_GetStartStopChn 4-65
K_GetStartStopG 4-67
K_GetTrig 4-70
K_GetTrigHyst 4-72
K_GetVer 2-30, 4-74
K_InitFrame 2-3, 4-76
K_IntAlloc 2-3, 4-78
K_IntFree 2-4, 4-80
K_IntStart 2-2, 4-81
K_IntStatus 2-3, 4-83
K_IntStop 2-3, 4-86
K_MoveBufToArray 2-5, 4-88
K_OpenDriver 2-28, 4-89
K_RestoreChanGAry 2-12, 4-92
K_SetADTrig 2-17, 4-93
K_SetBuf 2-4, 2-5, 4-95
K_SetBufI 2-4, 4-97
K_SetChn 2-8, 4-99
K_SetChnGAry 2-11, 2-12, 4-101
K_SetClk 2-13, 4-103
K_SetClkRate 2-13, 4-105
K_SetContRun 2-16, 4-107
K_SetDITrig 2-20, 4-109
K_SetG 2-8, 2-9, 4-111
K_SetGate 2-22, 4-113
K_SetStartStopChn 2-9, 4-115
K_SetStartStopG 2-9, 4-117
K_SetTrig 2-16, 4-120
K_SetTrigHyst 2-18, 4-122
K_SyncStart 2-2, 4-124
memory management 4-2
miscellaneous 4-4
operation 4-2
readback 3-2, 3-3
setup 3-2, 3-3
trigger 4-4

G

Gain codes 2-6
Gains 2-6
    see also analog input ranges 2-5
Gains: *see* Analog input ranges
Gate functions 4-4
Gates 2-22
Group of consecutive channels 2-9

H

Hardware gates: *see* Gates
Help 1-6
Hysteresis 2-18

I

Initialization functions 4-2
Initializing a board 2-29
Initializing the driver 2-28
Input range type 2-5
Installing the software 1-2
Internal clock source 2-13
    used in triggered operation 2-19, 2-21
    used with hardware gate 2-22
Internal trigger 2-16
Interrupt mode 2-2
    programming tasks 3-9
Interrupt status 2-3

K

K_ADRead 2-2, 2-8, 4-19
K_BufListAdd 2-4, 2-5, 4-22
K_BufListReset 2-4, 4-24
K_ClearFrame 3-4, 4-26
K_CloseDriver 2-28, 4-28
K_ClrContRun 2-16, 4-30
K_DASDevInit 2-29, 4-32
K_DIRead 2-24, 4-33
K_DOWrite 2-25, 4-35
K_FormatChanGAry 2-12, 4-37
K_FreeDevHandle 2-29, 4-38
K_FreeFrame 3-2, 4-39
K_GetADFrame 3-2, 4-40
K_GetADTrig 4-42
K_GetBuf 4-44
K_GetChn 4-46
K_GetChnGAry 4-48
K_GetClk 4-50
K_GetClkRate 4-52
K_GetContRun 4-54
K_GetDevHandle 2-29, 4-56
K_GetDITrig 4-58
K_GetErrMsg 2-30, 4-60
K_GetG 4-61
K_GetGate 4-63
K_GetStartStopChn 4-65
K_GetStartStopG 4-67
K_GetTrig 4-70
K_GetTrigHyst 4-72
K_GetVer 2-30, 4-74
K_InitFrame 2-3, 4-76
K_IntAlloc 2-3, 4-78
K_IntFree 2-4, 4-80
K_IntStart 2-2, 4-81
K_IntStatus 2-3, 4-83
K_IntStop 2-3, 4-86
K_MoveBufToArray 2-5, 4-88
K_OpenDriver 2-28, 4-89
K_RestoreChanGAry 2-12, 4-92
K_SetADTrig 2-17, 4-93
K_SetBuf 2-4, 2-5, 4-95
K_SetBufI 2-4, 4-97
K_SetChn 2-8, 4-99
K_SetChnGAry 2-11, 2-12, 4-101
K_SetClk 2-13, 4-103
K_SetClkRate 2-13, 4-105
K_SetContRun 2-16, 4-107

K_SetDITrig 2-20, 4-109
K_SetG 2-8, 2-9, 4-111
K_SetGate 2-22, 4-113
K_SetStartStopChn 2-9, 4-115
K_SetStartStopG 2-9, 4-117
K_SetTrig 2-16, 4-120
K_SetTrigHyst 2-18, 4-122
K_SyncStart 2-2, 4-124

M

Maintenance operations: *see* System operations
Managing memory 2-3
Memory allocation 2-3
Memory handle 2-3
Memory management 2-3
Memory management functions 4-2
Microsoft C/C++
  programming information 3-13
  see also C languages
Microsoft Professional Basic: *see* Professional Basic
Microsoft QuickBASIC (Version 4.0): *see* QuickBASIC (Version 4.0)
Microsoft QuickBasic (Version 4.5): *see* QuickBasic (Version 4.5)
Microsoft QuickC for Windows: *see* QuickC for Windows
Microsoft Visual Basic for DOS: *see* Visual Basic for DOS
Microsoft Visual Basic for Windows: *see* Visual Basic for Windows
Microsoft Visual C++: *see* Visual C++
Miscellaneous functions 4-4
Miscellaneous operations: *see* System operations
Multiple buffers 2-4

N

Normal mode 2-13
Null terminated strings
  DAS800_DevOpen 4-8
  K_OpenDriver 4-90

O

Operation functions 4-2
Operation modes 2-2, 3-1
Operations
  analog input 2-1
  counter/timer I/O 2-26
  digital I/O 2-24
  system 2-27
Operation-specific programming tasks 3-6

P

Pacer clocks: *see* Conversion clocks
Pascal
  setting up a channel-gain list 2-10
  specifying the buffer address 2-5, 3-18
  see also Turbo Pascal, Turbo Pascal for Windows
Preliminary tasks 3-6
Professional Basic
  programming information 3-22
  see also BASIC

Programming information
Borland C/C++ 3-14
Microsoft C/C++ 3-13
Professional Basic 3-22
QuickBASIC (Version 4.0) 3-20
QuickBasic (Version 4.5) 3-21
QuickC for Windows 3-15
Turbo Pascal 3-16
Turbo Pascal for Windows 3-17
Visual Basic for DOS 3-23
Visual Basic for Windows 3-24
Visual C++ 3-16
Programming overview 3-5
Programming tasks
analog input operations 3-6
common 3-6
digital I/O operations 3-12
interrupt-mode analog input operations 3-9
preliminary 3-6
single-mode analog input operations 3-7
synchronous-mode analog input operations 3-7

Q

QuickBASIC (Version 4.0)
programming information 3-20
see also BASIC
QuickBasic (Version 4.5)
programming information 3-21
see also BASIC
QuickC for Windows
programming information 3-15
see also C languages

R

Readback functions 3-2, 3-3
Retrieving data from buffer 3-19
Return values 2-30
Revision levels 2-30
Routines: *see* Functions

S

Sampling rate 2-15
Setting up boards 1-5
Setup functions 3-2, 3-3
interrupt mode 3-10
synchronous mode 3-7
Signal range: *see* Input range type
Single channel 2-8
Single-cycle mode 2-16
Single mode 2-2
programming tasks 3-7
Software
installing 1-2
packages 1-1
see also ASO-800 software package, DAS-800 standard software package
Standard software package 1-1
installing 1-2
Starting an analog input operation 2-2
Status codes 2-30, A-1
Status Word for interrupt-mode operations 4-84
Storing data: *see* Buffering modes
Synchronous mode 2-2
programming tasks 3-7
System operations 2-27

T

Technical support 1-6
Time base 2-13
Trigger functions 4-4
Trigger sources 2-16
Triggers 2-16
Triggers, analog: *see* Analog triggers
Triggers, digital: *see* Digital triggers
Turbo Pascal
    programming information 3-16
    specifying the channel-gain list starting address 3-19
    see also Pascal
Turbo Pascal for Windows
    programming information 3-17
    see also Pascal

U

Unipolar configuration: *see* Input range type

V

Visual Basic for DOS
    programming information 3-23
    see also BASIC
Visual Basic for Windows
    programming information 3-24
    setting up a channel-gain list 2-11
    specifying the buffer address 2-4, 3-25, 3-26
Visual C++
    programming information 3-16
    see also C languages

W

Windows
    allocating memory 2-4