DAS-TC
Function Call Driver

USER'S GUIDE

DAS-TC
Function Call Driver
User's Guide

Revision A – March 1996
Part Number: 84080

New Contact Information

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

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

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

Keithley MetraByte Division

Keithley Instruments, Inc.

440 Myles Standish Blvd. Taunton, MA 02780

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

Preface

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

- Microsoft® QuickBasic™ (Version 4.5)
- Microsoft Professional Basic (Version 7.0 and higher)
- Microsoft C/C++ (Version 7.0 and higher)
- Borland® C/C++ (Version 4.0 and higher)
- Borland Turbo Pascal® for DOS (Version 6.0 and higher)

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

- Microsoft C/C++ (Version 7.0 and higher)
- Borland C/C++ (Version 4.0 and higher)
- Microsoft Visual Basic for Windows® (Version 3.0 and higher)
- Microsoft Visual C++™ (Version 1.5)
- Borland Turbo Pascal for Windows (Version 1.0)

The manual is intended for application programmers using a DAS-TC or DAS-TC/B board in an IBM® PC AT® or compatible computer. Before using this manual, read the user's guide for your board to familiarize yourself with the board's features and complete the appropriate hardware installation and configuration. It is assumed that you are experienced in programming in your selected language and that you are familiar with data acquisition principles.

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

- Chapter 1 contains the information needed to get started using the DAS-TC Function Call Driver and to get help.
- Chapter 2 contains the background information needed to use the functions included in the DAS-TC Function Call Driver.
- Chapter 3 contains programming guidelines and language-specific information related to using the DAS-TC Function Call Driver.
- Chapter 4 contains detailed descriptions of the DAS-TC Function Call Driver functions, arranged in alphabetical order.
- Appendix A contains a list of the error codes returned by DAS-TC Function Call Driver functions.
- Appendix B contains information on the data formats used.

An index completes this manual.

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

- References to BASIC apply to all DOS-based BASIC languages (Microsoft QuickBasic and Microsoft Professional Basic). When a feature applies to a specific language, the complete language name is used. References to Visual Basic for Windows apply to Microsoft Visual Basic for Windows.
- Keyboard keys are represented in bold.

Table of Contents

4 Function Reference

List of Figures

List of Tables

1

Getting Started

The DAS-TC 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-TC standard software package** - This is the software package that is shipped with the DAS-TC and DAS-TC/B boards; it includes the following:
  - Libraries of FCD functions for Microsoft QuickBasic and Microsoft Professional Basic.
  - Support files, containing program elements, such as function prototypes and definitions of variable types, that are required by the FCD functions.
  - Utility programs (DOS-based only) that allow you to configure and test the features of DAS-TC and DAS-TC/B boards.
  - Language-specific example programs.
- **ASO-TC software package** - This is the advanced software option for the DAS-TC boards. It includes the following:
  - Libraries of FCD functions for Microsoft C/C++, Borland C/C++, and Borland Turbo Pascal.
  - Dynamic Link Libraries (DLLs) of FCD functions for Microsoft C/C++, Borland C/C++, Microsoft Visual Basic 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 (DOS-based and Windows-based) that allow you to configure and test the features of the DAS-TC and DAS-TC/B boards.
- Language-specific example programs.

Before you use the Function Call Driver, make sure that you have installed the software and your board using the procedures described in the user's guide for your board.

If you need help installing or using the DAS-TC Function Call Driver, call your local sales office or call the following number for technical support:

(508) 880-3000

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

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

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

DAS-TC/B board	Model	_____
	Serial #	_____
	Revision code	_____
	Base Address	_____
	Interrupt Level	_____
	Thermocouple type	_____
Computer	Manufacturer	_____
	CPU type	386 486 Pentium ____
	Clock speed (MHz)	_____
	Math coprocessor	Yes No
	Amount of RAM	_____
	Video system	EGA VGA SVGA
	BIOS type	_____
	Memory manager	_____
Operating system	DOS version	_____
	Windows version	3.0 3.1 95
Software package	Name	_____
	Serial #	_____
	Version	_____
	Invoice/Order #	_____
Compiler (if applicable)	Language	_____
	Manufacturer	_____
	Version	_____
Accessories	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____
	Type/Number	_____

2

Available Operations

This chapter contains the background information you need to use the FCD functions to perform operations on DAS-TC and DAS-TC/B boards. Table 2-1 lists the supported operations.

Table 2-1. Supported Operations

System Operations

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

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

Initializing the Driver

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

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

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

Note: If you are programming in BASIC or Turbo Pascal, **K_OpenDriver** and **K_CloseDriver** are not available. You must use the **DASTC_DevOpen** function instead. **DASTC_DevOpen** initializes the DAS-TC Function Call Driver according to the configuration file you specify. Refer to page 4-5 for more information. In BASIC and Turbo Pascal, closing the DAS-TC Function Call Driver is not required.

Initializing a Board

The DAS-TC Function Call Driver supports up to two DAS-TC or DAS-TC/B boards. You must use the **K_GetDevHandle** function to specify the boards you want to use. The driver returns a unique identifier for each board; this identifier is called the device handle.

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

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

Note: If you are programming in BASIC or Turbo Pascal, **K_GetDevHandle** and **K_FreeDevHandle** are not available. You must use the **DASTC_GetDevHandle** function instead. Refer to page 4-11 for more information. In BASIC or Turbo Pascal, freeing a device handle is not required.

Use **K_GetDevHandle** or **DASTC_GetDevHandle** the first time you initialize a DAS-TC or DAS-TC/B board only. Once you have a device handle, you can reinitialize a board as needed by using the **K_DASDevInit** function.

Retrieving Revision Levels

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

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

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

Handling Errors

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

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

C/C++	page 3-14
Turbo Pascal	page 3-24
Visual Basic for Windows	page 3-31
BASIC	page 3-38

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

Analog Input Operations

This section describes the following:

- Analog input operation modes available.
- How to allocate and manage memory for analog input operations.
- How to specify the following for an analog input operation:
  - Channels and input range
  - Buffering mode

Operation Modes

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

Single Mode

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

Use the **K_ADRead** function that is appropriate to your programming language to start an analog input operation in single mode. For each function you specify the board you want to use, the analog input channel to read, the gain for that channel (for voltage inputs only), and the variable in which to store the converted data.

Note: For thermocouple inputs, specify 0 for the gain; the gain is ignored for thermocouple inputs.

Depending on your configuration, the data is returned as a single voltage or temperature value in engineering units. Refer to Appendix B for more information on the format of the data returned.

If you wish, you can use a **K_ADRead** function with software looping to acquire more than one value from one or more channels. Typically, when acquiring more than one value you want more control over the data transfer than is possible with a single-mode function; in such cases, use either synchronous or interrupt mode, described in the next sections.

Note: To read the value of the CJC (cold junction compensation) channel, use the single-mode function **DASTC_GETCJC**. You can use the resulting value to correct a temperature reading when you want to perform your own linearization.

Synchronous Mode

In synchronous mode, the board acquires a single sample or multiple samples from one or more analog input channels. After transferring the specified number of samples to computer memory, the driver returns control to the application program. You cannot perform any other operation until a synchronous-mode operation is complete.

The DAS-TC and DAS-TC/B boards transfer data in blocks, where the block size equals the number of channels specified. Suppose, for example, you requested 43 samples using 10 channels. The Function Call Driver actually acquires 50 values in five blocks of 10 samples each. The first 40 values are transferred from the first four blocks that were acquired and the remaining three samples are transferred from the fifth acquired block of 10 samples.

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

Depending on your configuration, the data is returned as voltage or temperature values in engineering units. Refer to Appendix B for more information on the format of the data returned.

Interrupt Mode

In interrupt mode, the board acquires a single sample or multiple samples from one or more analog input channels. Once the analog input operation begins, control returns to your application program. The hardware transfers the data from the board to a user-defined buffer in computer memory using an interrupt service routine.

As in synchronous mode, in interrupt mode, the DAS-TC and DAS-TC/B boards transfer data in blocks, where the block size equals the number of channels specified.

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

Depending on your configuration, the data is returned as voltage or temperature values in engineering units. Refer to Appendix B for more information on the format of the data returned.

You can specify either single-cycle or continuous buffering mode for interrupt-mode operations. Refer to page 2-13 for more information on buffering modes. Use the **K_IntStop** function to stop a continuous-mode interrupt operation. Use the **K_IntStatus** function to determine the current status of an interrupt operation.

Memory Allocation and Management

Synchronous-mode and interrupt-mode analog input operations on the DAS-TC and DAS-TC/B boards require a single array or memory buffer in which to store acquired data. The ways you can allocate and manage memory are described in the following sections.

Dimensioning a Local Array

For the DAS-TC and DAS-TC/B boards, the simplest way to reserve a memory buffer is to dimension an array within your application program. The advantage of this method is that the array is directly accessible to your application program. The limitations of this method are as follows:

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

Make sure that the array you dimension matches the data type (long integer or floating point) specified in the configuration file. A single sample is four bytes long. Therefore, you should declare a local array as an array of four byte elements, the size of which is at least equal to the number of samples you are acquiring. For example, if you want to acquire 16,384 samples, you must dimension a 64K byte array.

Dynamically Allocating a Memory Buffer

If you wish, you can also reserve a memory buffer by allocating it dynamically outside of your application program's memory area. The advantages of this method are as follows:

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

The limitation of this method is that for Visual Basic for Windows and BASIC, the data in a dynamically allocated memory buffer is not directly accessible to your program. You must use the **K_MoveBufToArrayL** function (for long integer arrays) or the **K_MoveBufToArrayR** function (for floating-point arrays) to move the data from the dynamically allocated memory buffer to the program's local array. For Visual Basic for Windows, refer to page 3-26 for more information; for BASIC, refer to page 3-32 for more information.

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

Make sure that the pointers to the buffers allocated by **K_IntAlloc** are appropriate to the number type (long integer or floating point) specified in the configuration file.

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

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

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

Assigning the Starting Address

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

Language	Function	Refer to
C/C++	K_SetBuf	page 3-11
Turbo Pascal	K_SetBuf	page 3-20
Visual Basic for Windows	K_SetBufL[1] K_SetBufR	page 3-26
BASIC	K_SetBufL K_SetBufR	page 3-32

Notes
[1] Use **K_SetBufL** for long integer arrays or buffers; use **K_SetBufR** for floating-point arrays or buffers.

Channels, Gains, and Inputs

DAS-TC and DAS-TC/B boards are software-configurable for up to 16 differential analog input channels (numbered 0 through 15). You can mix and match thermocouple and voltage inputs. You configure the channels using the DASTCCFG.EXE configuration utility; refer to the user's guide for your board for more information. Table 2-2 lists the input types supported by the DAS-TC and DAS-TC/B boards.

Table 2-2. Input Types

Voltage Inputs	–2.5 V to 10 V
	–20 mV to 80 mV
	–15 mV to 60 mV
	–6.25 mV to 25 mV
Thermocouple Inputs	Type J
	Type K
	Type E
	Type T
	Type R
	Type S
	Type B

The input range is usually determined by the settings in the configuration file. However, for voltage inputs only, you can specify an input range using the gain and gain code, shown in Table 2-3. (The gain code is used by the FCD functions to represent the gain.) For thermocouple inputs, the gain is ignored; specify a gain code of 0 for thermocouple inputs.

Table 2-3. Gain Codes for Voltage Inputs

Gain Code	Gain	Voltage Input Range
0	1	–2.5 V to 10 V
1	125	–20 mV to 80 mV
2	166.67	–15 mV to 60 mV
3	400	–6.25 mV to 25 mV

Depending on the settings in the configuration file, data is returned in volts, degrees Celsius, or degrees Fahrenheit, as appropriate for the input types configured. Refer to Appendix B for information on the data formats.

How you specify a channel and the input range differs depending on the operation mode and the sequence of channels you want to use, as described in the following sections.

Specifying a Single Channel or a Group of Consecutive Channels

For single-mode operations, you can acquire a single sample from a single channel. Use the **K_ADRead** function appropriate to your programming language to specify an analog input channel and the gain for the channel (for voltage inputs only).

Note: For thermocouple inputs, the gain is ignored; specify a gain code of 0 for channels configured as thermocouple inputs.

For synchronous-mode and interrupt-mode analog input operations, you can acquire samples from a single channel or a group of consecutive channels. Use the **K_SetStartStopChn** function to specify the first and last channels in the group; to read a single channel, specify the same channel as both the start and the stop channel. The input ranges of the channels are determined by the settings in the configuration file.

The channels are sampled in order from the first to the last. For example, if the start channel is 10 and the stop channel is 15, the channels are sampled in the following order: 10, 11, 12, 13, 14, 15. If the start channel is 10 and the stop channel is 3, the channels are sampled in the following order: 10, 11, 12, 13, 14, 15, 0, 1, 2, 3. The channels are repeatedly sampled in the specified order until the required number of samples is read.

Note: When you use the **K_SetStartStopChn** function, the Function Call Driver reads the configuration file to determine whether the signal connected to the specified channel is a voltage input or a thermocouple input and to determine the appropriate gain for that channel. If you want to change the gain without changing the configuration file, use a channel-gain queue, as described in the next section.

Specifying Channels in a Channel-Gain Queue

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

Note: For thermocouple inputs, the gain is ignored; specify a gain code of 0 for channels configured as thermocouple inputs.

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

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

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

C/C++	page 3-13
Turbo Pascal	page 3-23
Visual Basic for Windows	page 3-30
BASIC	page 3-37

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

Buffering Modes

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

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

  If you are using continuous buffering, as soon as the last block of samples is transferred, the following occur:

  – the transfer count and buffer pointer are reset to zero

  – **K_IntStatus** returns zero instead of the requested sample size in the *index* parameter

  – the driver begins to overwrite your buffer's data

Therefore, if your application requires consecutive blocks of data, you should begin processing the buffer before the buffer is full, using **K_IntStatus** to determine how many blocks have been transferred (this function's *index* parameter increments by the block size).

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

3

Programming with the Function Call Driver

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

How the Driver Works

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

You can program one or more boards in your application program. Up to two DAS-TC or DAS-TC/B boards are supported. You initialize each board using a unique device handle to identify each board. Each device handle is associated with a particular driver.

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

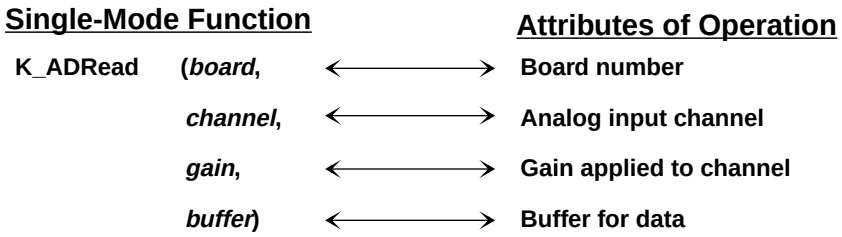

Figure 3-1. Single-Mode Function

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

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

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

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

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

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

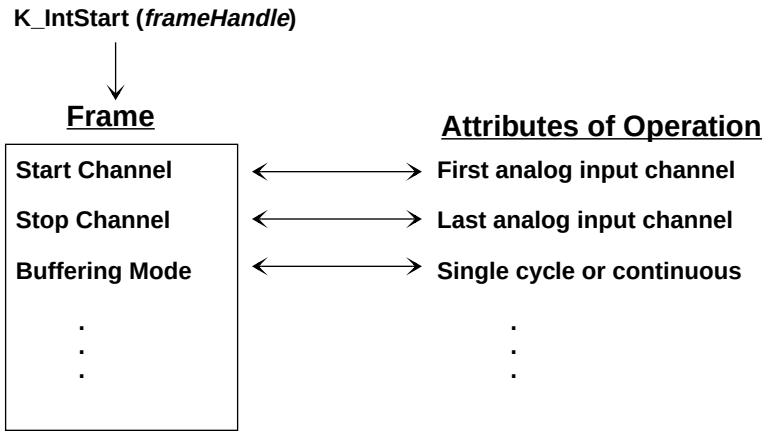

Figure 3-2. Interrupt-Mode Operation

For DAS-TC and DAS-TC/B boards, synchronous-mode and interrupt-mode analog input operations require frames, called A/D (analog-to-digital) frames. Use the **K_GetADFrame** function to access an available A/D frame.

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

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

Table 3-1 lists the elements of an A/D frame for DAS-TC and DAS-TC/B boards. This table also lists the default value of each element and the setup function used to define each element.

Table 3-1. A/D Frame Elements

Element	Default Value	Setup Function
Buffer[1]	0 (NULL)	K_SetBuf[2] K_SetBufL K_SetBufR
Number of Samples	0	K_SetBuf[2] K_SetBufL K_SetBufR
Buffering Mode	Single-cycle	K_SetContRun K_ClrContRun[3]
Start Channel	0	K_SetStartStopChn
Stop Channel	0	K_SetStartStopChn
Gain	0	Not applicable[4]
Channel-Gain Queue	0 (NULL)	K_SetChnGAry

Notes

[1] You must set this element.

[2] Use **K_SetBuf** for C/C++ and Turbo Pascal languages; use **K_SetBufL** (for long integer arrays) or **K_SetBufR** (for floating-point arrays) for Visual Basic and BASIC languages.

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

[4] The gain value is ignored; the driver reads the value from the configuration file.

Note: The DAS-TC 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 initialization functions, memory management functions, and miscellaneous functions.

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

Programming Overview

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

1. Define the application's requirements. Refer to Chapter 2 for a description of the operations supported by the Function Call Driver and the functions that you can use to define each operation.
2. Write your application program. Refer to the following for additional information:
   - Preliminary Tasks, the next section, which describes the programming tasks that are common to all application programs.
   - Analog Input Programming Tasks on page 3-6, which describes operation-specific programming tasks and the sequence in which these tasks must be performed.
   - Chapter 4, which contains detailed descriptions of the FCD functions.
   - The example programs in the DAS-TC standard software package and the ASO-TC software package. The FILES.TXT file in the installation directory lists and describes the example programs.
3. Compile and link the program. Refer to the following pages for information on compile and link statements and other language-specific considerations:

C/C++	page 3-11
Turbo Pascal	page 3-20
Visual Basic for Windows	page 3-26
BASIC	page 3-32

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-TC standard software package or the ASO-TC software package.
2. Declare and initialize program variables.
3. Use a driver initialization function (**K_OpenDriver** or **DASTC_DevOpen**) to initialize the driver.
4. Use a board initialization function (**K_GetDevHandle** or **DASTC_GetDevHandle**) to specify the DAS-TC or DAS-TC/B board you want to use and to initialize the board. If you are using two DAS-TC or DAS-TC/B boards, repeat this step.

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

Analog Input Programming Tasks

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

Single-Mode Operations

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

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

The following table lists the three **K_ADRead** functions and explains when to use each function. For details on each function, refer to Chapter 4.

Function	Use with
K_ADRead	C/C++, Turbo Pascal, and Turbo Pascal for Windows for any single-mode operation.
K_ADReadL	Visual Basic for Windows and BASIC, when you want to store the value read as a long integer.
K_ADReadR	Visual Basic for Windows and BASIC, when you want to store the value read as a floating-point number.

Synchronous-Mode Operations

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

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Dimension the array in which to store the acquired data. (Use the **K_IntAlloc** function if you want to allocate the buffer dynamically outside your program's memory area.)
3. *If you want to use a channel-gain queue to specify the channels acquiring data*, define and assign the appropriate values to the queue and note the starting address. Refer to page 2-12 for more information about channel-gain queues.
4. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-2.

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

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

Attribute	Setup Functions
Buffer	K_SetBuf[1] K_SetBufL K_SetBufR
Number of Samples	K_SetBuf[1] K_SetBufL K_SetBufR
Start Channel	K_SetStartStopChn
Stop Channel	K_SetStartStopChn
Channel-Gain Queue	K_SetChnGAry

Notes

[1] Use **K_SetBuf** for C/C++ and Turbo Pascal languages; use **K_SetBufL** (for long integer arrays) or **K_SetBufR** (for floating-point arrays) for Visual Basic and BASIC languages.

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

5. Use the **K_SyncStart** function to start the synchronous-mode operation.
6. *If you are programming in Visual Basic for Windows or BASIC and you used **K_IntAlloc** to allocate your buffer*, use the **K_MoveBufToArrayL** function (for long integer arrays) or the **K_MoveBufToArrayR** function (for floating-point arrays) to transfer the acquired data from the allocated buffer to the program's local array.
7. *If you used **K_IntAlloc** to allocate your buffer*, use the **K_IntFree** function to deallocate the buffer.
8. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

Interrupt-Mode Operations

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

1. Use the **K_GetADFrame** function to access an A/D frame.
2. Dimension the array in which to store the acquired data. (Use the **K_IntAlloc** function if you want to allocate a buffer dynamically outside your program's memory area.)
3. *If you want to use a channel-gain queue to specify the channels acquiring data*, define and assign the appropriate values to the queue and note the starting address. Refer to page 2-12 for more information about channel-gain queues.
4. Use the appropriate setup functions to specify the attributes of the operation. The setup functions are listed in Table 3-3.

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

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

Attribute	Setup Functions
Buffer	K_SetBuf[1] K_SetBufL K_SetBufR
Number of Samples	K_SetBuf[1] K_SetBufL K_SetBufR
Buffering Mode	K_SetContRun K_ClrContRun
Start Channel	K_SetStartStopChn

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

Attribute	Setup Functions
Stop Channel	K_SetStartStopChn
Channel-Gain Queue	K_SetChnGAry

Notes

[1] Use **K_SetBuf** for C/C++ and Turbo Pascal languages; use **K_SetBufL** (for long integer arrays) or **K_SetBufR** (for floating-point arrays) for Visual Basic and BASIC languages.

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

5. Use the **K_IntStart** function to start the interrupt-mode operation.
6. Use the **K_IntStatus** function to monitor the status of the interrupt-mode operation.
7. *If you specified continuous buffering mode*, use the **K_IntStop** function to stop the interrupt-mode operation when the appropriate number of samples has been acquired.
8. *If you are programming in Visual Basic for Windows or BASIC and you used **K_IntAlloc** to allocate your buffer*, use the **K_MoveBufToArrayL** function (for long integer arrays) or the **K_MoveBufToArrayR** function (for floating-point arrays) to transfer the acquired data from the allocated buffer to the program's local array.
9. *If you used **K_IntAlloc** to allocate your buffer*, use the **K_IntFree** function to deallocate the buffer.
10. Use the **K_FreeFrame** function to return the frame you accessed in step 1 to the pool of available frames.

C/C++ Programming Information

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

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

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

Dimensioning and Assigning a Local Array

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

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations. The following code fragment illustrates how to dimension an array of 10,000 samples for the frame defined by hFrame and how to use **K_SetBuf** to assign the starting address of the array.

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

Dynamically Allocating and Assigning a Memory Buffer

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

Note: If you are using a large memory buffer, you may be limited in the amount of memory you can allocate. It is recommended that you install the Keithley Memory Manager before you begin programming to ensure that you can allocate a large enough buffer. Refer to the user's guide for your board for more information on the Keithley Memory Manager.

Allocating a Memory Buffer

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

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

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

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

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

Accessing the Data

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

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

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

Creating a Channel-Gain Queue

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

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

The following example illustrates how to create a channel-gain queue called MyChanGainQueue for a DAS-TC or DAS-TC/B board by declaring and initializing a variable of type GainChanTable:

```
GainChanTable MyChanGainQueue =
   {8,          //Number of entries
   0, 0,        //Channel 0, gain is ignored for
                //thermocouples
   1, 3,        //Channel 1, gain of 400
   2, 2,        //Channel 2, gain of 166.67
   3, 1,        //Channel 3, gain of 125
   3, 0,        //Channel 3, gain of 1
   2, 3,        //Channel 2, gain of 400
   1, 3,        //Channel 1, gain of 400
   0, 0};       //Channel 0, gain is ignored
                //for thermocouples
```

Note: Gain for thermocouple inputs is ignored; specify a gain code of 0 for channels configured as thermocouple inputs.

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

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

When you start the next analog input operation (using **K_SyncStart** or **K_IntStart**), the channels are sampled in the following order: channel 0, 1, 2, 3, 3, 2, 1, 0.

Handling Errors

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

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

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

```
. . .
if ((wDasErr = K_SetStartStopChn (hAD, 2, 15) ! = 0)
  {
  Error = K_GetErrMsg (hDev, wDasErr, &pMessage);
  printf ("%s", pMessage);
  exit (1);
  }
```

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

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

File	Description
DASTC.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C
DTCDECL.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DASTC.HPP	Include file when compiling in C++
USEDASTC.OBJ	Linkable object

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

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

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

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

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DASTC.DLL	Dynamic Link Library
DASDECL.H	Include file when compiling in C
DTCDECL.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DASTC.HPP	Include file when compiling in C++
DASIMP.LIB	DAS Shell Imports
DASTCIMP.LIB	DAS-TC Imports

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

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

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

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

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

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

File	Description
DASTC.LIB	Linkable driver
DASRFACE.LIB	Linkable driver
DASDECL.H	Include file when compiling in C
DTCDECL.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DASTC.HPP	Include file when compiling in C++
USEDASTC.OBJ	Linkable object

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

Type of Compile	Compile and Link Statements[1]
C	BCC *filename*.c usetc.obj dastc.lib dasrface.lib
C++	BCC *filename*.cpp usetc.obj dastc.lib dasrface.lib

Notes

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

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

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

File	Description
DASSHELL.DLL	Dynamic Link Library
DASSUPRT.DLL	Dynamic Link Library
DASTC.DLL	Dynamic Link Library
DASDECL.H	Include file when compiling in C
DTCDECL.H	Include file when compiling in C
DASDECL.HPP	Include file when compiling in C++
DASTC.HPP	Include file when compiling in C++
DASTCIMP.LIB	DAS Shell Imports
DTCIMP.LIB	DAS-TC Imports

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

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

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

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

Turbo Pascal Programming Information

The following sections contain information you need to dimension an array or allocate a memory buffer, to create channel-gain queues, and to handle errors when programming in Turbo Pascal, as well as language-specific information for Borland Turbo Pascal (for DOS) and Borland Turbo Pascal for Windows.

Dimensioning and Assigning a Local Array

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

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

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

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

Dynamically Allocating and Assigning a Memory Buffer

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

Note: If you are using a large buffer and you are programming in Borland Turbo Pascal for Windows, you may be limited in the amount of memory you can allocate. It is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate a large enough buffer. Refer to the user's guide for your board for more information about the Keithley Memory Manager.

Reducing the Memory Heap

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

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

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

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

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

Allocating a Memory Buffer

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

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

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

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

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

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

Accessing the Data

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

```
. . .
for i := 0 to 9 do begin
   writeln ('Sample #', i,' =', AcqBuf^[i]);
End;
. . .
```

Creating a Channel-Gain Queue

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

```
Type
   GainChanTable = Record
   num_of_codes : Integer;
   queue : Array[0..255] of Byte;
   end;
. . .
Const
   MyChanGainQueue : GainChanTable = (
   num_of_codes : (8);     { Number of entries }
   queue :(0, 0,     { Channel 0, gain is ignored for thermocouples}
          1, 3,      { Channel 1, gain of 400}
          2, 2,      { Channel 2, gain of 166.67}
          3, 1,      { Channel 3, gain of 125}
          3, 0,      { Channel 3, gain of 1}
          2, 3,      { Channel 2, gain of 400}
          1, 3,      { Channel 1, gain of 400}
          0, 0)      { Channel 0, gain is ignored for thermocouples}
         );
         . . .
wDasErr := K_SetChnGAry (hFrame, MyChanGainQueue.num_of_codes);
```

Note: Gain for thermocouple inputs is ignored; specify a gain code of 0 for channels configured as thermocouple inputs.

When you start the next analog input operation (using **K_SyncStart** or **K_IntStart**), the channels are sampled in the following order: channel 0, 1, 2, 3, 3, 2, 1, 0.

Handling Errors

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

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

Programming in Borland Turbo Pascal (for DOS)

To program in Borland Turbo Pascal, you need the file DASTC.TPU, which is the Turbo Pascal unit for Version 6.0. This file is provided in the ASO-TC software package.

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.

Programming in Borland Turbo Pascal for Windows

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

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

Visual Basic for Windows Programming Information

The following sections contain information you need to dimension an array or allocate a memory buffer, to create channel-gain queues, and to handle errors in Microsoft Visual Basic for Windows, as well as other language-specific information for Microsoft Visual Basic for Windows.

Dimensioning and Assigning a Local Array

This section provides code fragments that describe how to dimension and assign a local array when programming in Microsoft Visual Basic for Windows. Note that the code fragments assume Option Base 0. Refer to the example programs on disk for more information.

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations. The following code fragment illustrates how to dimension a long integer array of 10,000 samples for the frame defined by hFrame and how to use **K_SetBufL** to assign the starting address of the long integer array.

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

Dynamically Allocating and Assigning a Memory Buffer

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

Note: If you are using a large buffer, you may be limited in the amount of memory you can allocate. It is recommended that you use the Keithley Memory Manager before you begin programming to ensure that you can allocate large enough buffers. Refer to the user's guide for your board for more information about the Keithley Memory Manager.

Allocating a Memory Buffer

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

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

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

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

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

Accessing the Data from Buffers with Fewer than 64K Bytes

In Microsoft Visual Basic for Windows, you cannot directly access analog input samples stored in a dynamically allocated memory buffer. You must use either the **K_MoveBufToArrayL** function (for long integer arrays) or the **K_MoveBufToArrayR** function (for floating-point arrays) to move a subset (up to 32,767 samples) of the data into a local array as required. The following code fragment illustrates how to move the first 100 samples of the buffer in the operation described in the previous section (AcqBuf) to a local array.

```
. . .
Dim Buffer(1000) As Long’ Declare local memory buffer
. . .
wDasErr = K_MoveBufToArrayL (Buffer(0), AcqBuf, 100)
. . .
```

Accessing the Data from Buffers with More than 64K Bytes

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

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

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

Table 3-4. Protected-Mode Memory Architecture

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

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

```
Dim AcqBuf As Long
Dim NumSamps As Long

Dim Array(999) As Long

NumSamps = 50000
wDasErr = K_IntAlloc (hFrame, NumSamps, AcqBuf, hMem)
. . .
'Acquisition routine
. . .
DesiredSamp = 40000
DesiredByte = DesiredSamp * 4       'Number of bytes into buffer
AddSelector = DesiredByte / &h10000 'Number of 64K pages into buffer
RemainingOffset = DesiredByte Mod &h10000    'Additional offset

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

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

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

```
...
Dim Data [2, 32768] As Long
...
wDasErr = K_MoveBufToArrayL (Data(0,0), pBuf, 32768)

'Add 8 to selector, offset = 0: add &h80000
wDasErr = K_MoveBufToArrayL (Data(1,0), pBuf + &h80000, 32768)

'Add 8 to selector, offset=0: add &h100000
wDasErr = K_MoveBufToArrayL (Data(2,0), pBuf + &h100000, 32768)
```

Creating a Channel-Gain Queue

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

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

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

```
. . .
Global MyChanGainQueue(9) As Integer '(4 channels x 2) + 1
. . .
MyChanGainQueue(0) = 4  ' Number of channel-gain pairs
MyChanGainQueue(1) = 0    ' Channel 0
MyChanGainQueue(2) = 0    ' Gain ignored for thermocouples
MyChanGainQueue(3) = 1    ' Channel 1
MyChanGainQueue(4) = 3    ' Gain of 400
MyChanGainQueue(5) = 2    ' Channel 2
MyChanGainQueue(6) = 2    ' Gain of 166.67
MyChanGainQueue(7) = 2    ' Channel 2
MyChanGainQueue(8) = 0    ' Gain of 1
. . .
wDasErr = K_FormatChnGAry (MyChanGainQueue(0))
wDasErr = K_SetChnGAry (hFrame, MyChanGainQueue(0))
. . .
```

Note: Gain for thermocouple inputs is ignored; specify a gain code of 0 for channels configured as thermocouple inputs.

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

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

When you start the next analog input operation (using **K_SyncStart** or **K_IntStart**), the channels are sampled in the following order: channel 0, 1, 2, 2.

Handling Errors

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

```
. . .
wDasErr = K_GetDevHandle (hDrv, BoardNum, hDev)
If (wDasErr <> 0) Then
   MsgBox "K_GetDevHandle Error: " + Hex$ (wDasErr),
      MB_ICONSTOP, "DAS-TC/B ERROR"
   End
End If
. . .
```

Programming in Microsoft Visual Basic for Windows

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

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

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

BASIC Programming Information

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

Dimensioning and Assigning a Local Array

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

You can use a single, local array for synchronous-mode and interrupt-mode analog input operations. The following code fragment illustrates how to dimension a long array of 10,000 samples for the frame defined by hFrame and how to use **KSetBufL** to assign the starting address of the long integer array.

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

Dynamically Allocating and Assigning a Memory Buffer

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

Reducing the Memory Heap

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

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

Allocating a Memory Buffer

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

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

```
. . .
Dim AcqBuf As Long   ' Declare pointer to buffer
Dim hMem As Integer  ' Declare memory handle
. . .
wDasErr = KIntAlloc% (hFrame, Samples, AcqBuf, hMem)
wDasErr = KSetBufL% (hFrame, AcqBuf, Samples)
. . .
```

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

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

Accessing the Data from Buffers with Fewer than 64K Bytes

In BASIC, you cannot directly access analog input samples stored in a dynamically allocated memory buffer. You must use either the **KMoveBufToArrayL** function (for long integer arrays) or the **KMoveBufToArrayR** function (for floating-point arrays) to move a subset of the data (up to 32,767 samples) into a local array. The following code fragment illustrates how to move the first 100 samples of the buffer in the operation described in the previous section (AcqBuf) into a local memory buffer.

```
. . .
Dim Buffer(99) As Long        ' Declare local memory buffer
. . .
wDasErr = KMoveBufToArrayL% (Buffer(0), AcqBuf, 100)
. . .
```

Accessing the Data from Buffers with More than 64K Bytes

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

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

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

Table 3-5. Real-Mode Memory Architecture

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

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

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

Dim Array(999) As Long

. . .                ’Initialize array with desired values

NumSamps = 50000
wDasErr = KIntAlloc% (hFrame, NumSamps, AcqBuf, hMem)

DesiredSamp = 40000
DesiredByte = DesiredSamp * 4       ’Number of bytes into buffer

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

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

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

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

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

Although it is recommended that you perform all calculations on the linear address and then convert the result to the segment:offset format (as shown in the previous code fragment), this example illustrates an alternative method of calculating the address by working on the segment:offset form of the address directly. You can use this method if you already know how deep you want to go into the buffer with each move and the offset of the starting address is 0, as is the case when the buffer is allocated with **KIntAlloc**.

In this method, you add &h10000000 to the buffer address for each 64K byte page and then add the remainder of the buffer.

```
...
Dim Data [2, 32768] As Long
...
wDasErr = KMoveBufToArrayL% (Data(0,0), pBuf, 32768)

’Add 8 to selector, offset = 0: add &h80000
wDasErr = KMoveBufToArrayL% (Data(1,0), pBuf + &h80000, 32768)

’Add 8 to selector, offset=0: add &h100000
wDasErr = KMoveBufToArrayL% (Data(2,0), pBuf + &h100000, 32768)
```

Creating a Channel-Gain Queue

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

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

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

```
. . .
Dim MyChanGainQueue(9) As Integer '(4 channels x 2) + 1
. . .
MyChanGainQueue(0) = 4     ' Number of channel-gain pairs
MyChanGainQueue(1) = 0     ' Channel 0
MyChanGainQueue(2) = 0     ' Gain ignored for thermocouples
MyChanGainQueue(3) = 1     ' Channel 1
MyChanGainQueue(4) = 3     ' Gain of 400
MyChanGainQueue(5) = 2     ' Channel 2
MyChanGainQueue(6) = 2     ' Gain of 166.67
MyChanGainQueue(7) = 2     ' Channel 2
MyChanGainQueue(8) = 0     ' Gain of 1
. . .
wDasErr = KFormatChnGAry% (MyChanGainQueue(0))
wDasErr = KSetChnGAry% (hFrame, MyChanGainQueue(0))
. . .
```

Note: Gain is ignored for thermocouple inputs; specify a gain code of 0 for channels configured as thermocouple inputs.

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

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

When you start the next analog input operation (using **KSyncStart** or **KIntStart**), the channels are sampled in the following order: channel 0, 1, 2, 2.

Handling Errors

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

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

Programming in Microsoft QuickBasic

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

File	Description
DTCQ45.LIB	Linkable driver for QuickBasic (Version 4.5) stand-alone, executable (.EXE) programs
DTCQ45.QLB	Command-line loadable driver for the QuickBasic (Version 4.5) integrated environment
QB4DECL.BI	Include file
DASDECL.BI	Include file
DASTC.BI	Include file

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

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

1. Enter the following to invoke the environment:

   ```
   QB /L DTCQ45 filename.bas
   ```

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

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

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

```
BC filename.bas /O
Link filename.obj,,,DTCQ45.lib+BCOM45.lib;
```

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

Programming in Microsoft Professional Basic

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

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

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

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

1. Enter the following to invoke the environment:

   ```
   QBX /L DTCQBX filename.bas
   ```

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

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

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

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

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

4

Function Reference

The FCD functions are organized into the following groups:

- Initialization functions
- Operation functions
- Frame management functions
- Memory management functions
- Buffer address functions
- Buffering mode functions
- Channel and gain functions
- Miscellaneous functions

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

Table 4-1. Functions

Function Type	Function Name	Page Number
Initialization	DASTC_DevOpen	page 4-5
	DASTC_GetDevHandle	page 4-11
	K_OpenDriver	page 4-65
	K_CloseDriver	page 4-24
	K_GetDevHandle	page 4-38
	K_FreeDevHandle	page 4-32
	K_DASDevInit	page 4-28
Operation	K_ADRead	page 4-13
	K_ADReadL	page 4-16
	K_ADReadR	page 4-19
	K_SyncStart	page 4-84
	K_IntStart	page 4-53
	K_IntStatus	page 4-55
	K_IntStop	page 4-58
Frame Management	K_GetADFrame	page 4-36
	K_FreeFrame	page 4-34
	K_ClearFrame	page 4-22
Memory Management	K_IntAlloc	page 4-48
	K_IntFree	page 4-51
	K_MoveBufToArrayL	page 4-61
	K_MoveBufToArrayR	page 4-63
Buffer Address	K_SetBuf	page 4-70
	K_SetBufL	page 4-72
	K_SetBufR	page 4-74

Table 4-1. Functions (cont.)

Function Type	Function Name	Page Number
Buffering Mode	K_SetContRun	page 4-79
	K_ClrContRun	page 4-26
Channel and Gain	K_SetStartStopChn	page 4-81
	K_SetChnGAry	page 4-76
	K_FormatChnGAry	page 4-30
	K_RestoreChnGAry	page 4-68
Miscellaneous	K_GetErrMsg	page 4-40
	K_GetVer	page 4-45
	K_GetShellVer	page 4-42
	DASTC_GETCJC	page 4-8

Keep the following conventions in mind throughout this chapter:

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

Table 4-2. Data Type Prefixes

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

DASTC_DevOpen

Purpose

Initializes the DAS-TC Function Call Driver.

Prototype

C/C++

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

Turbo Pascal

Function DASTC_DevOpen (Var *szCfgFile* : char;
Var *pBoards* : Integer) : Word; far; external 'DASTC';

Turbo Pascal for Windows

Function DASTC_DevOpen (Var *szCfgFile* : char;
Var *pBoards* : Integer) : Word; far; external 'DASTC';

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION DASTCDEVOPEN% ALIAS
"DASTC_DevOpen" (BYVAL *szCfgFile* AS LONG,
SEG *pBoards* AS INTEGER)

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the driver according to the information in the configuration file specified by *szCfgFile* and stores the number of DAS-TC or DAS-TC/B boards defined in *szCfgFile* in *pBoards*.

DASTC_DevOpen (cont.)

You create a configuration file using the DASTCCFG.EXE utility. If *szCfgFile* = 0, **DASTC_DevOpen** looks for the DASTC.CFG configuration file in the current directory. If *szCfgFile* = –1, **DASTC_DevOpen** uses the default configuration settings. Refer to the user's guide for your board for more information about configuration files and settings.

See Also

K_OpenDriver

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
szCfgName : String;
nBoards : Integer;
...
szCfgName := 'DASTC.CFG' + #0;
wDasErr := DASTC_DevOpen (szCfgName[1], nBoards);
```

Turbo Pascal for Windows

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

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

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

BASIC

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

DASTC_GETCJC

Purpose

Returns the value of the CJC on the DAS-TC or DAS-TC/B board in degrees Celsius; this value is used to correct temperature input values.

Prototype

C/C++

DASErr far pascal DASTC_GETCJC (int *nBrdNum*,
float far **pCJCtemp*);

Turbo Pascal

Function DASTC_GETCJC (*nBrdNum* : Integer;
Var *pCJCtemp* : Real) : Word; far; external 'DASTC';

Turbo Pascal for Windows

Function DASTC_GETCJC *nBrdNum* : Integer;
Var *pCJCtemp* : Single) : Word; far; external 'DASTC';

Visual Basic for Windows

Declare Function DASTC_GETCJC Lib "DASTC.DLL"
(ByVal *nBrdNum* As Integer, *pCJCtemp* As Single) As Integer

BASIC

DECLARE FUNCTION DASTCGETCJC% ALIAS "DASTC_GETCJC"
(BYVAL *nBrdNum* AS INTEGER, BYVAL *pCJCtemp* AS SINGLE)

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
pCJCtemp	CJC sensor temperatures in degrees Celsius.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DAS-TC or DAS-TC/B board specified by *nBrdNum*, this function reads the cold junction compensation temperature at the STA-TC, STC-TC, STA-TC/B, or STC-TC/B terminals connected to the board and stores the value in *pCJCtemp*.

The board number specified in *nBrdNum* refers to the board number specified in the configuration file.

The value stored in *pCJCtemp* is floating point regardless of the format specified in the configuration file.

In order to obtain a temperature reading from a thermocouple type not recognized by the Function Call Driver, you need to perform your own linearization by calling **DASTC_GETCJC** and using the resulting value to correct the linearization.

Depending on the volatility of the ambient temperature where the CJC resides, use **DASTC_GETCJC** more often as you take more samples.

Usage

C/C++

```
#include "DTCDECL.H"   // Use DASTC.HPP for C++
...
float hTemp;
...
wDasErr = DASTC_GETCJC (0, &hTemp);
```

Turbo Pascal

```
uses DTCTPU;
...
hTemp : Real;   {CJC Temperature}
...
wDasErr := DASTC_GETCJC (0, hTemp);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hTemp : Single;   { CJC Temperature }
...
wDasErr := DASTC_GETCJC (0, hTemp);
```

Visual Basic for Windows

(Add DTCDECL.BAS to your project)

```
...
Global hTemp As Single ' CJC Temperature
...
wDasErr = DASTC_GETCJC (0, hTemp)
```

DASTC_GETCJC (cont.)

BASIC

```
' $INCLUDE: 'DASTC.BI'
...
DIM hTemp AS Single' CJC Temperature
...
wDasErr = DASTCGETDEVHANDLE% (0, hTemp)
```

Purpose

Initializes a DAS-TC or DAS-TC/B board.

Prototype

C/C++

DASErr far pascal DASTC_GetDevHandle (WORD *nBrdNum*,
void far * far **phDev*);

Turbo Pascal

Function DASTC_GetDevHandle (*nBrdNum* : Word;
Var *phDev* : Longint) : Word; far; external 'DASTC';

Turbo Pascal for Windows

Function DASTC_GetDevHandle (*nBrdNum* : Word;
Var *phDev* : Longint) : Word; far; external 'DASTC';

Visual Basic for Windows

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

BASIC

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

Parameters

nBrdNum	Board number. Valid values: **0** or **1**
phDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DAS-TC or DAS-TC/B board specified by *nBrdNum* and stores the device handle of the specified board in *phDev.*

The board number specified in *nBrdNum* refers to the board number specified in the configuration file.

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

DASTC_GetDevHandle (cont.)

See Also K_GetDevHandle

Usage

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DTCDECL.BAS to your project)

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

BASIC

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

K_ADRead

Purpose

For use with the C/C++, Turbo Pascal, and Turbo Pascal for Windows languages only, reads a single analog input value.

Prototype

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

Not supported. Use **K_ADReadL** or **K_ADReadR** instead.

BASIC

Not supported. Use **K_ADReadL** or **K_ADReadR** instead.

Parameters

hDev	Handle associated with the board.
nChan	Analog input channel. Valid values: **0** to **15**

nGain — Gain code.
Valid values are listed in the table below. For thermocouple inputs, gain is ignored; specify a gain code of 0 for thermocouple inputs.

Gain Code	Gain	Voltage Input Range
0	1	–2.5 V to 10 V
1	125	–20 mV to 80 mV
2	166.67	–15 mV to 60 mV
3	400	–6.25 mV to 25 mV

pData — Acquired analog input value.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function reads the analog input channel *nChan* on the DAS-TC or DAS-TC/B board specified by *hDev* and stores the value in *pData*.

Depending on the input type specified in the configuration file, the value stored in *pData* is in microvolts or in hundredths of degrees for integer types and is not scaled for floating point. Refer to Appendix B for more information on the format of acquired data.

See Also

K_IntStart, K_SyncStart

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
dwADValue : Long;
...
wDasErr := K_ADRead (hDev, 0, 0, @dwADValue);
```

Turbo Pascal for Windows

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

K_ADReadL

Purpose

For use with the Visual Basic for Windows and BASIC languages only, reads a single analog input value. Use **K_ADReadL** when you want to store the value as a long integer.

Prototype

C/C++
Not supported. Use **K_ADRead** instead.

Turbo Pascal
Not supported. Use **K_ADRead** instead.

Turbo Pascal for Windows
Not supported. Use **K_ADRead** instead.

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

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

Parameters

hDev	Handle associated with the board.
nChan	Analog input channel. Valid values: **0** to **15**

nGain Gain code.
Valid values are listed in the table below. For thermocouple inputs, gain is ignored; specify a gain code of 0 for thermocouple inputs.

Gain Code	Gain	Voltage Input Range
0	1	–2.5 V to 10 V
1	125	–20 mV to 80 mV
2	166.67	–15 mV to 60 mV
3	400	–6.25 mV to 25 mV

pData Acquired analog input value.

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

Remarks This function reads the analog input channel *nChan* on the DAS-TC or DAS-TC/B board specified by *hDev* and stores the value in *pData*.

Depending on the input type specified in the configuration file, the value stored in *pData* is in microvolts or in hundredths of degrees for long integer types. Refer to Appendix B for more information on the format of acquired data.

See Also K_ADReadR, K_IntStart, K_SyncStart

Usage

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

```
...
Global dwADValue As Long
...
wDasErr = K_ADReadL (hDev, 0, 0, dwADValue)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwADValue AS LONG
...
wDasErr = KADREADL% (hDev, 0, 0, dwADValue)
```

Purpose

For use with the Visual Basic for Windows and BASIC languages only, reads a single analog input value. Use **K_ADReadR** when you want to store the value using a floating-point (real) data type.

Prototype

C/C++
Not supported. Use **K_ADRead** instead.

Turbo Pascal
Not supported. Use **K_ADRead** instead.

Turbo Pascal for Windows
Not supported. Use **K_ADRead** instead.

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

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

Parameters

hDev	Handle associated with the board.
nChan	Analog input channel. Valid values: **0** to **15**

nGain — Gain code.
Valid values are listed in the table below. For thermocouple inputs, gain is ignored; specify a gain code of 0 for thermocouple inputs.

Gain Code	Gain	Voltage Input Range
0	1	–2.5 V to 10 V
1	125	–20 mV to 80 mV
2	166.67	–15 mV to 60 mV
3	400	–6.25 mV to 25 mV

pData — Acquired analog input value.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function reads the analog input channel *nChan* on the DAS-TC or DAS-TC/B board specified by *hDev* and stores the value in *pData*.

The value stored in *pData* is not scaled for floating point. Refer to Appendix B for more information on the format of acquired data.

See Also

K_ADReadL, K_IntStart, K_SyncStart

Usage

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

```
...
Global dwADValue As Single
...
wDasErr = K_ADReadR (hDev, 0, 0, dwADValue)
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM dwADValue AS SINGLE
...
wDasErr = KADREADR% (hDev, 0, 0, dwADValue)
```

K_ClearFrame

Purpose

Sets the elements of a frame to their default values.

Prototype

C/C++

DASErr far pascal K_ClearFrame (DWORD *hFrame*);

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

Refer to Table 3-1 on page 3-4 for the default values of an A/D frame.

See Also

K_GetADFrame

Usage

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_CloseDriver

Purpose

Closes a previously initialized Keithley DAS Function Call Driver.

Prototype

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

Turbo Pascal
Not supported

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

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

BASIC
Not supported

Parameters

hDrv	Driver handle you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

See Also

K_FreeDevHandle

Usage

C/C++

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

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

K_ClrContRun

Purpose

Specifies single-cycle buffering mode.

Prototype

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

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

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

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

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

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

Refer to page 2-13 for more information on buffering modes.

See Also

K_SetContRun

Usage

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_DASDevInit

Purpose

Reinitializes a Keithley MetraByte DAS board.

Prototype

C/C++

DASErr far pascal K_DASDevInit (DWORD *hDev*);

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

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

BASIC

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

Parameters

hDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

Use **K_GetDevHandle** or **DASTC_GetDevHandle** the first time you initialize the board only. Once you have a device handle, use this function to reinitialize the board.

Usage

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_FormatChnGAry

Purpose

Converts the format of a channel-gain queue.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

After you use this function, your program can no longer read the converted channel-gain queue. You must use the **K_RestoreChnGAry** function to return the queue to its original format. Refer to page 3-30 for more information on creating channel-gain queues in Visual Basic; refer to page 3-37 for more information on creating channel-gain queues in BASIC.

See Also

K_SetChnGAry, K_RestoreChnGAry

Usage

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_FreeDevHandle

Purpose

Frees a previously specified device handle.

Prototype

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

Turbo Pascal
Not supported

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

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

BASIC
Not supported

Parameters

phDev	Device handle you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_GetDevHandle

Usage

C/C++

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

Turbo Pascal for Windows

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

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

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

K_FreeFrame

Purpose

Frees a frame.

Prototype

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

Turbo Pascal
Function K_FreeFrame (*hFrame* : Longint) : Word;

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

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

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

Parameters

hFrame	Handle to frame you want to free.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_GetADFrame

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_FreeFrame (hAD);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_GetADFrame

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_GetADFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word;

Turbo Pascal for Windows

Function K_GetADFrame (*hDev* : Longint;
Var *phFrame* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Remarks

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

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

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

See Also

K_ClearFrame, K_FreeFrame

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
hAD : Longint;
...
wDasErr := K_GetADFrame (hDev, hAD);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hAD : Longint;
...
wDasErr := K_GetADFrame (hDev, hAD);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_GetDevHandle

Purpose

Initializes any Keithley MetraByte DAS board.

Prototype

C/C++

DASErr far pascal K_GetDevHandle (DWORD *hDrv*,
WORD *nBoardNum*, DWORD far * *phDev*);

Turbo Pascal

Not supported

Turbo Pascal for Windows

Function K_GetDevHandle (*hDrv* : Longint; *nBoardNum* : Integer;
Var *phDev* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

Not supported

Parameters

hDrv	Driver handle of the associated Function Call Driver.
nBoardNum	Board number. Valid values: **0** or **1**
phDev	Handle associated with the board.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DAS-TC or DAS-TC/B board associated with *hDrv* and specified by *nBoardNum*, and stores the device handle of the specified board in *phDev*.

The board number specified in *nBoardNum* refers to the board number specified in the configuration file.

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

See Also

K_FreeDevHandle

Usage

C/C++

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

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
hDev : Longint;
...
wDasErr := K_GetDevHandle (hDrv, 0, hDev);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

K_GetErrMsg

Purpose

Gets the address of an error message string.

Prototype

C/C++

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

Turbo Pascal

Not supported

Turbo Pascal for Windows

Not supported

Visual Basic for Windows

Not supported

BASIC

Not supported

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DAS-TC or DAS-TC/B board specified by *hDev*, this function stores the address of the string corresponding to the error message number *nDASErr* in *pszErrMsg*.

Refer to page 2-4 and to page 3-14 for more information about error handling. Refer to Appendix A for a list of error codes and their meanings.

Usage

C/C++

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

K_GetShellVer

Purpose

Gets the current DAS shell version.

Prototype

C/C++

DASErr far pascal K_GetShellVer (WORD far **pVersion*);

Turbo Pascal

Function K_GetShellVer (Var *pVersion* : Word) : Word;

Turbo Pascal for Windows

Function K_GetShellVer (Var *pVersion* : Word) : Word; far;
external 'DASSHELL';

Visual Basic for Windows

Declare Function K_GetShellVer Lib "DASSHELL.DLL"
(*pVersion* As Integer) As Integer

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wShellVer : Word;
...
wDasErr := K_GetShellVer (wShellVer);
FormatStr(VerStr, ' %4x ',nShellVer / 256, '.', nShellVer And $ff);
writeln (' Shell Ver ', VerStr);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wShellVer : Word;
...
wDasErr := K_GetShellVer (wShellVer);
FormatStr(VerStr,' %4x ', nShellVer / 256, '.', nShellVer And $ff);
writeln (' Shell Ver ', VerStr);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

K_GetShellVer (cont.)

BASIC

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

Purpose

Gets revision numbers.

Prototype

C/C++

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

Turbo Pascal

Function K_GetVer (*hDev* : Longint; Var *pSpecVer* : Word; Var *pDrvVer* : Word) : Word;

Turbo Pascal for Windows

Function K_GetVer (*hDev* : Longint; Var *pSpecVer* : Word; Var *pDrvVer* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the DAS-TC or DAS-TC/B board specified by *hDev*, this function stores the revision number of the Function Call Driver in *pDrvVer* and the revision number of the driver specification in *pSpecVer*.

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

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

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

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
nSpecVer : Word;
nDrvVer : Word;
...
wDasErr := K_GetVer (hDev, nSpecVer, nDrvVer);
FormatStr (VerStr, ' %4x ', nDrvVer / 256, '.', nDrvVer And $ff);
writeln (' Driver Ver ', VerStr);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
nSpecVer : Word;
nDrvVer : Word;
...
wDasErr := K_GetVer (hDev, nSpecVer, nDrvVer);
FormatStr(VerStr, ' %4x ', nDrvVer / 256, '.', nDrvVer And $ff);
writeln (' Driver Ver ', VerStr);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_IntAlloc

Purpose

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

Prototype

C/C++

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

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
dwSamples	Number of samples. Valid values: **1** to **65535**
pBuf	Starting address of the allocated buffer.
phMem	Handle associated with the allocated buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

BASIC and Turbo Pascal (for DOS) require that you redistribute available memory before you dynamically allocate a buffer. Refer to page 3-32 (BASIC) or page 3-21 (Turbo Pascal) for additional information.

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

See Also

K_IntFree, K_SetBuf, K_SetBufL, K_SetBufR

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
TYPE
BufType = Array [0..1] of Longint;
VAR
pBuf : ^BufType;   { buffer pointer }
hMem : Word;   { Handle to buffer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr(pBuf), hMem);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
TYPE
BufType = Array [0..1] of Longint;
VAR
pBuf : ^BufType;   { buffer pointer }
hMem : Word;   { Handle to buffer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr(pBuf), hMem);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

Purpose

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

Prototype

C/C++

DASErr far pascal K_IntFree (WORD *hMem*);

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

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

BASIC

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

Parameters

hMem Handle to buffer.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_IntAlloc

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_IntFree (hMem);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

Purpose

Starts an interrupt-mode operation.

Prototype

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

Turbo Pascal
Function K_IntStart (*hFrame* : Longint) : Word;

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

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

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

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

The acquired values are stored at the location identified by the Buffer Address element of the frame identified by *hFrame*.

Depending on the settings in the configuration file, the values are stored in microvolts or in hundredths of degrees for integer types and are not scaled for floating point.

Refer to page 3-9 for a discussion of the programming tasks associated with interrupt-mode analog input operations.

See Also

K_IntStatus, K_IntStop

K_IntStart (cont.)

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_IntStart (hAD);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_IntStart (hAD)
```

BASIC

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

Purpose

Gets the status of an interrupt-mode operation.

Prototype

C/C++

DASErr far pascal K_IntStatus (DWORD *hFrame*, short far **pStatus*, DWORD far **pIndex*);

Turbo Pascal

Function K_IntStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pIndex* : Longint) : Word;

Turbo Pascal for Windows

Function K_IntStatus (*hFrame* : Longint; Var *pStatus* : Word; Var *pIndex* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt-mode operation. Valid values: **0** = Interrupt operation idle **1** = Interrupt operation active
pIndex	Buffer array index.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the interrupt-mode operation defined by *hFrame*, this function stores the status in *pStatus* and the index of the next element in the buffer or array to be written to in *pIndex*.

In continuous buffering operations, *pIndex* is reset to zero when the last block transfer has been completed and another acquisition cycle is initiated.

See Also K_IntStart, K_IntStop

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wStatus : Word;
dwIndex : Longint;
...
wDasErr := K_IntStatus (hAD, wStatus, dwIndex);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

K_IntStop

Purpose

Stops an interrupt-mode operation.

Prototype

C/C++

DASErr far pascal K_IntStop (DWORD *hFrame*, short far **pStatus*, DWORD far **pIndex*);

Turbo Pascal

Function K_IntStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pIndex* : Longint) : Word;

Turbo Pascal for Windows

Function K_IntStop (*hFrame* : Longint; Var *pStatus* : Word; Var *pIndex* : Longint) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

hFrame	Handle to the frame that defines the operation.
pStatus	Status of interrupt-mode operation. Valid values: **0** = Interrupt operation idle **1** = Interrupt operation active
pIndex	Buffer array index.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function stops the interrupt-mode operation defined by *hFrame*, stores the status of the interrupt-mode operation in *pStatus*, and stores the index of the next element in the buffer or array to be written to in *pIndex*.

In continuous buffering operations, *pIndex* is reset to zero when the last block transfer has been completed and another acquisition cycle is initiated.

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

See Also

K_IntStart, K_IntStatus

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wStatus : Word;
dwIndex : Longint;
...
wDasErr := K_IntStop (hAD, wStatus, dwIndex);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

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

Purpose

For use with the Visual Basic for Windows and BASIC languages only, transfers data from a buffer allocated through **K_IntAlloc** to the program's local array. Use **K_MoveBufToArrayL** when you want to store the data in a long-integer array.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

K_MoveBufToArrayL (cont.)

Remarks

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

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

See Also

K_IntAlloc, K_MoveBufToArrayR

Usage

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

```
...
Dim ADArray (1000) As Long
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
...
wDasErr = K_MoveBufToArrayL (ADArray(0), pBuf, 1000)
```

BASIC

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

Purpose

For use with the Visual Basic for Windows and BASIC languages only, transfers data from a buffer allocated through **K_IntAlloc** to the program's local array. Use **K_MoveBufToArrayR** when you want to store the data in a floating-point (real) array.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

K_MoveBufToArrayR (cont.)

Remarks

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

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

See Also

K_IntAlloc, K_MoveBufToArrayL

Usage

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

```
...
Dim ADArray (1000) As Single
...
wDasErr = K_IntAlloc (hAD, 1000, pBuf, hMem)
...
wDasErr = K_MoveBufToArrayR (ADArray(0), pBuf, 1000)
```

BASIC

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

Purpose

Initializes any Keithley DAS Function Call Driver.

Prototype

C/C++

DASErr far pascal K_OpenDriver (char far * *szDrvName*,
char far * *szCfgName*, DWORD far * *phDrv*);

Turbo Pascal

Not supported

Turbo Pascal for Windows

Function K_OpenDriver (Var *szDrvName* : char; Var *szCfgName* : char;
Var *phDrv* : LongInt) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_OpenDriver Lib "DASSHELL.DLL"
(ByVal *szDrvName* As String, ByVal *szCfgName* As String,
phDrv As Long) As Integer

BASIC

Not supported

Parameters

szDrvName	Driver name. Valid value: **"DASTC"** (for DAS-TC or DAS-TC/B boards)
szCfgName	Driver configuration file. Valid value: The name of a configuration file **0** if driver has already been opened.
phDrv	Handle associated with the driver.

Return Value

Error/status code. Refer to Appendix A.

Remarks

This function initializes the DAS-TC Function Call Driver according to the information in the configuration file specified by *szCfgName*, and stores the driver handle in *phDrv*.

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

For DAS-TC or DAS-TC/B boards, the string stored in *szDrvName* must be DASTC.

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

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

See Also DASTC_DevOpen

Usage

C/C++

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

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
szDrvName : String;
szCfgName : String;
hDrv : Longint;
...
szDrvName := 'DASTC' + #0;
szCfgName := 'DASTC.CFG' + #0;
wDasErr := K_OpenDriver (szDrvName[1], szCfgName[1], hDrv);
```

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

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

K_RestoreChnGAry

Purpose

Restores a converted channel-gain queue.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

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

Parameters

pArray	Channel-gain queue starting address.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

See Also

K_FormatChnGAry, K_SetChnGAry

Usage

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

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

BASIC

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

K_SetBuf

Purpose

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

Prototype

C/C++

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

Turbo Pascal

Function K_SetBuf (*hFrame* : Longint; *pBuf* : Pointer; *dwSamples* : Longint) : Word;

Turbo Pascal for Windows

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

Visual Basic for Windows

Not supported

BASIC

Not supported

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of buffer or array.
dwSamples	Number of samples. Valid values: **1** to **65535**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

For Visual Basic for Windows and BASIC, use **K_SetBufL** for long integer arrays or buffers or **K_SetBufR** for floating point arrays or buffers.

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

See Also

K_IntAlloc, K_SetBufL, K_SetBufR

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
TYPE
BufType = Array [0..1] of Longint;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr(pBuf), hMem);
wDasErr := K_SetBuf (hAD, pBuf, 1000);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
TYPE
BufType = Array [0..1] of Longint;
VAR
pBuf : ^BufType;   { buffer pointer }
...
wDasErr := K_IntAlloc (hAD, 1000, Addr(pBuf), hMem);
wDasErr := K_SetBuf (hAD, pBuf, 1000);
```

K_SetBufL

Purpose

Specifies the starting address of a long integer array or buffer and the number of samples to acquire.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

BASIC
DECLARE FUNCTION KSETBUFL% Alias "K_SetBuf"
(BYVAL *hFrame* AS LONG, SEG *pBuf* AS LONG,
BYVAL *dwSamples* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of the long integer array or buffer.
dwSamples	Number of samples. Valid values: **1** to **65535**

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function specifies the starting address of the long integer array or buffer in *pBuf* and the number of samples to acquire in *dwSamples*.

For C/C++ and Turbo Pascal application programs, use **K_SetBuf**.

For Visual Basic for Windows and BASIC, use this function only for long integer arrays or buffers; for floating point arrays or buffers, use **K_SetBufR**.

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

See Also

K_IntAlloc, K_SetBuf, K_SetBufR

Usage

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

```
...
Dim ADData(2000) As Long
...
wDasErr = K_SetBufL (hAD, ADData(0), 2000)
```

BASIC

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

K_SetBufR

Purpose

Specifies the starting address of a floating point array or buffer and the number of samples to acquire.

Prototype

C/C++
Not supported

Turbo Pascal
Not supported

Turbo Pascal for Windows
Not supported

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

BASIC
DECLARE FUNCTION KSETBUFR% Alias "K_SetBuf"
(BYVAL *hFrame* AS LONG, SEG *pBuf* AS Single,
BYVAL *dwSamples* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.
pBuf	Starting address of the floating point array or buffer.
dwSamples	Number of samples. Valid values: **1** to **65535**

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

For C/C++ and Turbo Pascal application programs, use **K_SetBuf**.

For Visual Basic for Windows and BASIC, use this function only for floating point arrays and buffers; for long integer arrays, use **K_SetBufL**.

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

See Also

K_IntAlloc, K_SetBuf, K_SetBufL

Usage

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

```
...
Dim ADData(2000) As Single
...
wDasErr = K_SetBufR (hAD, ADData(0), 2000)
```

BASIC

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

K_SetChnGAry

Purpose

Specifies the starting address of a channel-gain queue.

Prototype

C/C++

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

Turbo Pascal

Function K_SetChnGAry (*hFrame* : Longint;
Var *pArray* : Integer) : Word;

Turbo Pascal for Windows

Function K_SetChnGAry (*hFrame* : Longint;
Var *pArray* : Integer) : Word; far; external 'DASSHELL';

Visual Basic for Windows

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

BASIC

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

Parameters

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

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

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

Refer to page 2-12 for information on setting up a channel-gain queue.

If you created your channel-gain queue in BASIC or Visual Basic for Windows, you must use **K_FormatChnGAry** to convert the channel-gain queue before you specify the address with **K_SetChnGAry**.

See Also

K_FormatChnGAry, K_RestoreChnGAry

Usage

C/C++

```
#include "DASDECL.H"   // Use DASDECL.HPP for C++
...
// DECLARE AND INITIALIZE CHAN/GAIN PAIRS
// (GainChanTable-TYPE IS DEFINED IN dasdecl.h)
GainChanTable ChanGainArray= {2,   // # of entries
   0, 3,    // chan 0, gain is 400
   1, 0};   // chan 1, gain is ignored for thermocouples
...
wDasErr = K_SetChnGAry (hAD, &ChanGainArray);
```

Turbo Pascal

```
uses DTCTPU;
...
{ Define Gain/Channel array type }
TYPE GainChanTable = Record
       num_of_codes : Integer;
       queue : Array[0..15] of Byte;
       END;
CONST ChanGainArray : GainChanTable = (
       num_of_codes : (8); { # of chan/gain pairs }
       queue : (0,2, 1,1)
);
...
wDasErr := K_SetChnGAry (hAD, ChanGainArray.num_of_codes);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
{ Define Gain/Channel array type }
TYPE GainChanTable = Record
       num_of_codes : Integer;
       queue : Array[0..15] of Byte;
       END;
CONST ChanGainArray : GainChanTable = (
       num_of_codes : (8);   { # of chan/gain pairs }
       queue : (0,2, 1,3)
);
...
wDasErr := K_SetChnGAry (hAD, ChanGainArray.num_of_codes);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
Global ChanGainArray(5) As Integer
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 3
ChanGainArray(3) = 1: ChanGainArray(4) = 2
wDasErr = K_FormatChnGAry (ChanGainArray(0))
wDasErr = K_SetChnGAry (hAD, ChanGainArray(0))
```

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
DIM ChanGainArray(5) AS INTEGER
...
' Create the array of channel/gain pairs
ChanGainArray(0) = 2   ' # of chan/gain pairs
ChanGainArray(1) = 0: ChanGainArray(2) = 1
ChanGainArray(3) = 1: ChanGainArray(4) = 3
wDasErr = KFORMATCHNGARY% (ChanGainArray(0))
wDasErr = KSETCHNGARY% (hAD, ChanGainArray(0))
```

Purpose

Specifies continuous buffering mode.

Prototype

C/C++

DASErr far pascal K_SetContRun (DWORD *hFrame*);

Turbo Pascal

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

Turbo Pascal for Windows

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

Visual Basic for Windows

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

BASIC

DECLARE FUNCTION KSETCONTRUN% ALIAS "K_SetContRun"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

For the operation defined by *hFrame*, this function sets the buffering mode to continuous mode and sets the Buffering Mode element in the frame accordingly.

K_GetADFrame and **K_ClearFrame** specify single-cycle as the default buffering mode.

Refer to page 2-13 for a description of buffering modes.

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_SetContRun (hAD);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SetContRun (hAD)
```

BASIC

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

Purpose

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

Prototype

C/C++

DASErr far pascal K_SetStartStopChn (DWORD *hFrame*, short *nStart*, short *nStop*);

Turbo Pascal

Function K_SetStartStopChn (*hFrame* : Longint; *nStart* : Word; *nStop* : Word) : Word;

Turbo Pascal for Windows

Function K_SetStartStopChn (*hFrame* : Longint; *nStart* : Word; *nStop* : Word) : Word; far; external 'DASSHELL';

Visual Basic for Windows

Declare Function K_SetStartStopChn Lib "DASSHELL.DLL" (ByVal *hFrame* As Long, ByVal *nStart* As Integer, ByVal *nStop* As Integer) As Integer

BASIC

DECLARE FUNCTION KSETSTARTSTOPCHN% ALIAS "K_SetStartStopChn" (BYVAL *hFrame* AS LONG, BYVAL *nStart* AS INTEGER, BYVAL *nStop* AS INTEGER)

Parameters

hFrame	Handle to the frame that defines the operation.
nStart	First channel in a group of consecutive channels. Valid values: **0** to **15**
nStop	Last channel in a group of consecutive channels. Valid values: **0** to **15**

Return Value

Error/status code. Refer to Appendix A.

K_SetStartStopChn (cont.)

Remarks

For the operation defined by *hFrame*, this function specifies the first channel in a group of consecutive channels in *nStart* and the last channel in the group of consecutive channels in *nStop*. To specify a single channel, enter the same channel number in *nStart* and *nStop*.

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

When you use the **K_SetStartStopChn** function, the Function Call Driver reads the configuration file to determine whether the signal connected to the specified channel is configured for voltage input or thermocouple input and to determine the appropriate gain for that channel. If you want to change the gain without changing the configuration file, use a channel-gain queue.

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

See Also

K_SetChnGAry

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_SetStartStopChn (hAD, 0, 7);
```

Turbo Pascal for Windows

```
{$I DASDECL.INC}
...
wDasErr := K_SetStartStopChn (hAD, 0, 7);
```

Visual Basic for Windows

(Add DASDECL.BAS to your project)

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

BASIC

```
' $INCLUDE: 'DASDECL.BI'
...
wDasErr = KSETSTARTSTOPCHN% (hAD, 0, 7)
```

K_SyncStart

Purpose

Starts a synchronous-mode operation.

Prototype

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

Turbo Pascal
Function K_SyncStart (*hFrame* : Longint) : Word;

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

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

BASIC
DECLARE FUNCTION KSYNCSTART% ALIAS "K_SyncStart"
(BYVAL *hFrame* AS LONG)

Parameters

hFrame	Handle to the frame that defines the operation.

Return Value

Error/status code. Refer to Appendix A.

Remarks

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

Refer to page 3-7 for information on the programming tasks associated with synchronous-mode analog input operations.

See Also

K_IntStart

Usage

C/C++

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

Turbo Pascal

```
uses DTCTPU;
...
wDasErr := K_SyncStart (hAD);
```

Turbo Pascal for Windows

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

Visual Basic for Windows

(Add DASDECL.BAS to your project)

```
...
wDasErr = K_SyncStart (hAD)
```

BASIC

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

A

Error/Status Codes

Table A-1 lists the error/status codes that are returned by the DAS-TC Function Call Driver, possible causes for error conditions, and possible solutions for resolving error conditions.

If you cannot resolve an error condition, contact Keithley MetraByte for technical support.

Table A-1. Error/Status Codes

Error Code		Cause	Solution
Hex	**Decimal**		
0	0	No error has been detected.	Status only; no action is necessary.
6000	24576	**Error in configuration file:** The configuration file you specified in the driver initialization function is corrupt, does not exist, or contains one or more undefined keywords.	Check that the file exists at the specified path. Check for illegal keywords in file; you can avoid illegal keywords by using the configuration utility to create and modify configuration files.
6001	24577	**Illegal base address in configuration file:** The base I/O address of the card/board in the configuration file is illegal and/or does not match the base address switches on the card/board.	Use the configuration utility to change the base I/O address of the card/board to one that matches the base address switches on the card/board, if applicable.
6002	24578	**Illegal IRQ level in configuration file:** The interrupt level in the configuration file is illegal.	Use the configuration utility to change the interrupt level to a legal one for your card/board. Refer to the user's guide for legal interrupt levels.

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

Error Code		Cause	Solution
Hex	Decimal		
6003	24579	**Illegal DMA channel in configuration file:** The DMA channel in the configuration file is illegal.	Use the configuration utility to change the DMA channel to a legal one for your card/board. Refer to the user's guide for legal DMA channels.
6005	24581	**Illegal channel number:** The specified channel number is illegal for the card/board and/or for the range type (unipolar or bipolar).	Specify a legal channel number. Refer to the user's guide or to the description of **K_SetStartStopChn** in Chapter 4 for legal channel numbers.
6006	24582	**Illegal gain code:** The specified analog I/O channel gain code is illegal for this card/board.	Specify a legal gain code. Refer to the user's guide or to the description of **K_SetG** in Chapter 4 for a list of legal gain codes.
6007	24583	**Illegal DMA address:** An FCD function specified a buffer address that is not suitable for a DMA operation for the number of samples required.	Use the **K_DMAAlloc** function to allocate dynamic buffers for DMA operations. In Windows, make sure that the Keithley Memory Manager is installed; refer to the user's guide for information.
6008	24584	**Illegal number in configuration file:** The configuration file contains one or more numeric values that are illegal.	Use the configuration utility to check and then change the configuration file.
600A	24586	**Configuration file not found:** The driver cannot find the configuration file specified as an argument to the driver initialization function.	Check that the file exists at the specified path. Check that the file name is spelled correctly in the driver initialization function parameter list.
600B	24587	**Error returning DMA buffer:** DOS returned an error in INT 21H function 49H during the execution of **K_DMAFree**.	Check that the memory handle passed as an argument to **K_DMAFree** was previously obtained using **K_DMAAlloc**.
600C	24588	**Error returning interrupt buffer:** The memory handle specified in **K_IntFree** is invalid.	Check the memory handle stored by **K_IntAlloc** and make sure that it was not modified.

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

Error Code		Cause	Solution
Hex	Decimal		
600D	24589	**Illegal frame handle:** The specified frame handle is not valid for this operation.	Check that the frame handle exists. Check that you are using the appropriate frame handle.
600E	24590	**No more frame handles:** No frames are left in the pool of available frames.	Use **K_FreeFrame** to free a frame that the application is no longer using.
600F	24591	**Requested buffer size too large:** The requested buffer cannot be dynamically allocated because of its size.	Specify a smaller buffer size; refer to the description of **K_IntAlloc** in Chapter 4 for the legal range. If in Windows Enhanced mode with the Keithley Memory Manager installed, use KMMSETUP.EXE to increase the reserved buffer heap size.
6010	24592	**Cannot allocate interrupt buffer:** (Windows-based languages only) **K_IntAlloc** failed because there was not enough available DOS memory.	Remove some Terminate and Stay Resident programs (TSRs) that are no longer needed.
6012	24594	**Interrupt buffer deallocation error:** (Windows-based languages only) An error occurred when **K_IntFree** attempted to free a memory handle.	Make sure that the memory handle passed as an argument to **K_IntFree** was previously obtained using **K_IntAlloc**.
6015	24597	**DMA Buffer too large:** The number of samples specified in **K_DMAAlloc** is too large.	Refer to the description of **K_DMAAlloc** in Chapter 4 for the buffer size range.
6016	24598	**VDS - Region not contiguous:** An error occurred while using Windows Virtual DMA Services. You tried to use **K_DMAAlloc** in Windows Enhanced mode and the Keithley Memory Manager was not installed.	Refer to the user's guide for information on how to install and set up the Keithley Memory Manager.
6017	24599	**VDS - DMA wraparound:** See error 6016.	See error 6016.

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

Error Code		Cause	Solution
Hex	**Decimal**		
6018	24600	**VDS - Unable to lock region:** See error 6016.	See error 6016.
6019	24601	**VDS - No buffer available:** See error 6016.	See error 6016.
601A	24602	**VDS - Region too large:** See error 6016.	See error 6016.
601B	24603	**VDS - Buffer in use:** See error 6016.	See error 6016.
601C	24604	**VDS - Illegal region:** See error 6016.	See error 6016.
601D	24605	**VDS - Region not locked:** See error 6016.	See error 6016.
601E	24606	**VDS - Illegal page:** See error 6016.	See error 6016.
601F	24607	**VDS - Illegal buffer:** See error 6016.	See error 6016.
6020	24608	**VDS - Copy out of range:** See error 6016.	See error 6016.
6021	24609	**VDS - Illegal DMA channel:** See error 6016.	See error 6016.
6022	24610	**VDS - Count overflow:** See error 6016.	See error 6016.
6023	24611	**VDS - Count underflow:** See error 6016.	See error 6016.
6024	24612	**VDS - Function not supported:** See error 6016.	See error 6016.
6025	24613	**Illegal OBM mode:** The mode number specified in **K_SetOBMMode** is illegal.	Specify a legal mode value; refer to the description of **K_SetOBMMode**.

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

Error Code		Cause	Solution
Hex	Decimal		
6026	24614	**Illegal DMA structure:** An error occurred during the execution of **K_DMAFree**.	Try using **K_DMAFree** again. If the error continues, contact Keithley MetraByte for technical support.
6027	24615	**DMA allocation error:** See error 6026.	See error 6026.
6028	24616	**NULL DMA handle:** See error 6026.	See error 6026.
6029	24617	**DMA unlock error:** See error 6026.	See error 6026.
602A	24618	**DMA free error:** See error 6026.	See error 6026.
602B	24619	**Not enough memory to accommodate request:** The number of samples you requested in the Keithley Memory Manager is greater than the largest contiguous block available in the reserved heap.	Specify a smaller number of samples. Free a previously allocated buffer. Use the KMMSETUP utility to expand the reserved heap.
602C	24620	**Requested buffer size exceeds maximum:** The number of samples you requested from the Keithley Memory Manager is greater than the allowed maximum.	Specify a value within the legal range when calling **K_DMAAlloc** or **K_IntAlloc** in Windows Enhanced mode. Refer to Chapter 4 for legal values.
602D	24621	**Illegal device handle:** A bad device handle was passed to a function such as **K_GetADFrame**. The handle used was not initialized through a call to the card/board initialization function (such as **K_GetDevHandle**) or it was corrupted by your program.	Check the device handle value.
602E	24622	**Illegal setup option:** An illegal option was specified to a function that accepts a user option, such as **K_SetDITrig**.	Check the option value passed to the function where the error occurred.

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

Error Code		Cause	Solution
Hex	**Decimal**		
6030	24624	**DMA word-page wrap:** During **K_DMAAlloc**, a DMA word-page wrap condition occurred and the allocation attempt failed since there is not enough free memory to accommodate the allocation request.	Reduce the number of samples and retry. If in Windows Enhanced mode, install and configure the Keithley Memory Manager.
6031	24625	**Illegal memory block handle:** A bad memory handle was passed to **K_IntFree** or **K_DMAFree**. The handle used was not initialized through a call to **K_IntAlloc** or **K_DMAAlloc**, or it was corrupted by your program.	Restart your program and monitor the memory handle values.
6032	24626	**Out of memory handles:** An attempt to allocate a memory block using **K_IntAlloc** or **K_DMAAlloc** failed because the maximum number of handles has already been assigned.	Use **K_IntFree** or **K_DMAFree** to free previously allocated memory blocks before allocating again.
6034	24628	**Memory corrupted:** Int 21H function 48H, used to allocate a memory block from the DOS far heap, returned the DOS error 7; this means that memory is corrupted. It is likely that you stored data (through a DMA-mode or interrupt-mode operation) into an illegal area of DOS memory.	Recheck the parameters set by **K_DMAAlloc** and **K_SetDMABuf**. If a fatal system error, restart your computer.
6035	24629	**Driver in use:** You attempted to initialize a driver that was already initialized by a call to **K_OpenDriver**. (This can occur since, under Windows, it is possible to open the same driver from multiple programs that are running simultaneously.)	To continue using the driver with the same configuration, pass a null string as the second argument to **K_OpenDriver**. To use the driver with a different configuration, close any application programs currently accessing the driver, and then open the driver again (using **K_OpenDriver**).

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

Error Code		Cause	Solution
Hex	Decimal		
6036	24630	**Illegal driver handle:** The specified driver handle is not valid.	Someone may have closed the driver; if so, use **K_OpenDriver** to reopen the driver with the desired driver handle. Try again using another driver handle.
6037	24631	**Driver not found:** The specified driver cannot be found.	Check your link statement to make sure the specified driver is included. Make sure that the device name string is entered correctly in **K_OpenDriver**.
6038	24632	**Invalid source pointer:** (Windows-based languages only) The pointer to the source buffer that you passed as an argument to **K_MoveBufToArrayL** or **K_MoveBufToArrayR** is invalid for the specified count. (The source pointer, when added to the number of samples, exceeds the programmed addressing range of that pointer.)	Check the pointer to the source buffer and the number of samples to transfer that you specified in **K_MoveBufToArrayL** or **K_MoveBufToArrayR**.
6039	24633	**Invalid destination pointer:** (Windows-based languages only) The pointer to the destination buffer (local array) that you passed as an argument to **K_MoveBufToArrayL** or **K_MoveBufToArrayR** is invalid for the specified count. (The destination pointer, when added to the number of samples, exceeds the dimension of the local array.)	Check the dimension of the local array and the number of samples to transfer that you specified in **K_MoveBufToArrayL** or **K_MoveBufToArrayR**.
603A	24634	**Illegal setup value:** An illegal value was passed to the function in which the error occurred.	Check the legal ranges of all parameters passed to this function.

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

Error Code		Cause	Solution
Hex	Decimal		
603B	24635	**Error freeing buffer selector: K_DMAFree** or **K_IntFree** failed because one or more of the selectors that reference the memory buffer could not be freed.	Check that the memory buffer being freed was previously obtained through **K_DMAAlloc** or **K_IntAlloc.**
603C	24636	**Error allocating buffer selector: K_DMAAlloc** or **K_IntAlloc** failed because a selector could not be allocated from Window's Local Descriptor Table.	Close all applications and restart Windows. If the error continues, contact Keithley MetraByte for technical support.
603D	24637	**Error allocating memory buffer: K_DMAAlloc** or **K_IntAlloc** failed because a necessary internal buffer could not be allocated to complete the operation.	Close all applications and restart Windows. If the error continues, contact Keithley MetraByte for technical support.
7000	28672	**No board name**: The driver initialization function did not find a board name in the specified configuration file.	Specify a legal board name in the configuration file.
7001	28673	**Bad board name**: The board name in the specified configuration file is illegal.	Specify a legal board name in the configuration file.
7002	28674	**Bad board numbe**r: The driver initialization function found an illegal board number in the specified configuration file.	Specify a legal board number: 0 or 1
7003	28675	**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 &H200 (512) to &H3F0 (1008) in increments of 10H (16). Make sure that &H precedes hexadecimal numbers.
7004	28676	**Bad interrupt level**: The driver initialization function found an illegal interrupt level in the specified configuration file.	Specify a legal interrupt level: 3, 5, 7, 10, 11, 12, or 15 for DAS-TC/B boards; 2, 3, 4, 5, 6, or 7 for DAS-TC boards

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

Error Code		Cause	Solution
Hex	Decimal		
7005	28677	**Bad Normal Mode Rejection Frequency**: You attempted to use a Normal Mode Rejection Frequency that is not supported.	Specify a legal normal mode rejection frequency in the configuration file (50 Hz, 60 Hz, or 400 Hz).
7006	28678	**Bad Number Type:** You attempted to use a number type that is not supported.	Specify a legal number type in the configuration file: integer or floating point
7007	28679	**Bad channel configuration:** The driver initialization function found a channel number out of range or an illegal channel argument.	Specify a legal channel configuration. See **K_SetStartStopChn**.
7008	28680	**Checksum Error**: The checksum is illegal resulting in communication failure.	Reinitialize the board using **K_DASDevInit**.
7009	28681	**Board Not Initialized:** A function was called before the board was initialized, initialization failed, or the wrong base address was specified.	Verify the base address and reinitialize the board using **K_DASDevInit**.
700A	28682	**Initialization Failure**: Initialization of the board failed.	Reinitialize the board using **K_DASDevInit**. If the problem persists, contact Keithley MetraByte for technical support.
700B	28683	**Protocol Communication Error:** Communication between the board and the computer failed.	Reinitialize the board using **K_DASDevInit**. If the problem persists, contact Keithley MetraByte for technical support.
700C	28684	**Bad Voltage to Temperature Calculation Error:** An error occurred when converting to engineering units.	Reinitialize the board using **K_DASDevInit**. If the problem persists, contact Keithley MetraByte for technical support.
8001	32769	**Function not supported:** You have attempted to use a function not supported by the Function Call Driver.	Contact Keithley MetraByte for technical support.

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

Error Code		Cause	Solution
Hex	**Decimal**		
8003	32771	**Illegal board number:** An illegal card/board number was specified in the card/board initialization function.	Refer to the description of the card/board initialization function (such as **K_GetDevHandle**) in Chapter 4 for legal card/board numbers.
8004	32772	**Illegal error number:** The error message number specified in **K_GetErrMsg** is invalid.	The error number must be one the error numbers listed in this appendix.
8005	32773	**Board not found at configured address:** The card/board initialization function does not detect the presence of a card/board.	If applicable, make sure that the base address setting of the switches on the card/board matches the base address setting in the configuration file.
8006	32774	**A/D not initialized:** You attempted to start a frame-based analog input operation without the A/D frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8007	32775	**D/A not initialized:** You attempted to start a frame-based analog output operation without the D/A frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8008	32776	**Digital input not initialized:** You attempted to start a frame-based digital input operation without the DI frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.
8009	32777	**Digital output not initialized:** You attempted to start a frame-based digital output operation without the DO frame being properly initialized.	Always call **K_ClearFrame** before setting up a new frame-based operation.

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

Error Code		Cause	Solution
Hex	**Decimal**		
800B	32779	**Conversion overrun:** The conversion rate is too fast or the time required to service an interrupt is too long.	Adjust the clock source to slow down the rate at which the card/board acquires data. Remove other application programs that are running and using computer resources. Try performing the operation in synchronous mode instead of interrupt mode.
8016	32790	**Interrupt overrun**: The card/board communicated a hardware event to the software by generating a hardware interrupt, but the software was still servicing a previous interrupt. This is usually caused by a pacer clock rate that is too fast.	Check the maximum throughput rate for your computer's programming environment and use **K_SetClkRate** to specify an appropriate rate.
801A	32794	**Interrupts already active:** You have attempted to start an operation whose interrupt level is being used by another system resource.	Use **K_IntStop** to stop the first operation before starting the second operation.
801B	32795	**DMA already active:** You attempted to start a DMA-mode operation using a DMA channel that is currently used by another active operation.	Use **K_DMAStop** to stop the first operation before starting the second operation.
8020	32800	**FIFO Overflow event detected:** During data acquisition, the temporary oncard/onboard data storage (FIFO) overflowed.	The conversion rate is too fast for your computer's programming environment; use **K_SetClkRate** to reduce the conversion rate. If you are using DMA-mode and your card/board supports dual-DMA, use the configuration utility to reconfigure your card/board to use dual-DMA.

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

Error Code		Cause	Solution
Hex	**Decimal**		
8021	32801	**Illegal clock sync mode:** The two operations you are trying to synchronize cannot be synchronized on your card/board.	Check the synchronizing operation that you specified in **K_SetSync**. Make sure that your card/board supports the synchronization of the two operations.
FFFF	65535	**User aborted operation:** You pressed **Ctrl+Break** during a synchronous-mode operation or while waiting for an analog trigger event to occur.	Start the operation again, if desired.

B

Data Formats

The DAS-TC Function Call Driver returns data in engineering units. The number type (integer or floating-point) specified in the configuration file determines what the data means, as described in the following sections.

Note: Ensure that the array or buffer you dimension matches the number type selected.

Integer Number Types

When the number type specified in the configuration file is integer, a twos complement 32-bit long integer is returned. If a channel is configured for thermocouple input, the value returned is in hundredths of degrees. If a channel is configured for voltage input, the value returned is in microvolts.

To convert hundredths of degrees to degrees, divide the value by 100. To convert microvolts to volts, divide the value by 1,000,000.

If the input is under or over the range setting of a particular channel, the DAS-TC or DAS-TC/B board returns the values shown in Table B-1.

Table B-1. Integer Input Error Conditions

Input Type	Condition	Value Returned
Voltage	Over the range setting	–971,227,136 mV
	Under the range setting	+1,176,256,512 mV
Thermocouple	Over the range setting	–971,227,136°C or F
	Under the range setting	+1,176,256,512°C or F

Floating-Point Number Types

When the number type specified in the configuration file is floating-point, an IEEE 32-bit real number is returned. If a particular channel is configured for thermocouple input, the value returned is in degrees. If a particular channel is configured for voltage input, the value returned is in volts.

If the input is under or over the range setting of a particular channel, the DAS-TC or DAS-TC/B returns the values shown in Table B-2.

Table B-2. Floating-Point Input Error Conditions

Input Type	Condition	Value Returned
Voltage	Over the range setting	–10,000.00 V
	Under the range setting	+10,000.00 V
Thermocouple	Over the range setting	–10,000.00°C or F
	Under the range setting	+10,000.00°C or F

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V