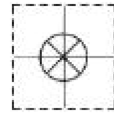


User's Guide

http://www.omega.com
e-mail: info@omega.com

OMB-DAQBOOK/DAQBOARD
Volume 2 - Programmer's Guide

Producing Custom Software
for Data Acquisition Systems

OMEGAnet℠ On-Line Service http://www.omega.com	Internet e-mail info@omega.com

Servicing North America:

USA: ISO 9001 Certified
One Omega Drive, Box 4047
Stamford, CT 06907-0047
Tel: (203) 359-1660 FAX: (203) 359-7700
e-mail: info@omega.com

Canada:
976 Berger
Laval (Quebec) H7L 5A1
Tel: (514) 856-6928 FAX: (514) 856-6886
e-mail: canada@omega.com

For immediate technical or application assistance:

USA and Canada:
Sales Service: 1-800-826-6342 / 1-800-TC-OMEGA℠
Customer Service: 1-800-622-2378 / 1-800-622-BEST℠
Engineering Service: 1-800-872-9436 / 1-800-USA-WHEN℠
TELEX: 996404 EASYLINK: 62968934 CABLE: OMEGA

Mexico and Latin America:
Tel: (95) 800-TC-OMEGA℠ FAX: (95) 203-359-7807
En Espanol: (95) 203-359-7803 e-mail: espanol@omega.com

Servicing Europe:

Benelux:
Postbus 8034, 1180 LA Amstelveen, The Netherlands
Tel: (31) 20 6418405 FAX: (31) 20 6434643
Toll Free in Benelux: 06 0993344
e-mail: nl@omega.com

Czech Republic:
ul. Rude armady 1868
733 01 Karvina-Hranice
Tel: 420 (69) 6311899 FAX: 420 (69) 6311114
e-mail:czech@omega.com

France:
9, rue Denis Papin, 78190 Trappes
Tel: (33) 130-621-400 FAX: (33) 130-699-120
Toll Free in France: 0800-4-06342
e-mail: france@omega.com

Germany/Austria:
Daimlerstrasse 26, D-75392 Deckenpfronn, Germany
Tel: 49 (07056) 3017 FAX: 49 (07056) 8540
Toll Free in Germany: 0130 11 21 66
e-mail: germany@omega.com

United Kingdom: ISO 9002 Certified
25 Swannington Road,
Broughton Astley, Leicestershire,
LE9 6TU, England
Tel: 44 (1455) 285520
FAX: 44 (1455) 283912

P.O. Box 7, Omega Drive,
Irlam, Manchester,
M44 5EX, England
Tel: 44 (161) 777-6611
FAX: 44 (161) 777-6622

Toll Free in England: 0800-488-488
e-mail: uk@omega.com

It is the policy of OMEGA to comply with all worldwide safety and EMC/EMI regulations that apply. OMEGA is constantly pursuing certification of its products to the European New Approach Directives. OMEGA will add the CE mark to every appropriate device upon certification.

The information contained in this document is believed to be correct but OMEGA Engineering, Inc. accepts no liability for any errors it contains, and reserves the right to alter specifications without notice.

WARNING: These products are not designed for use in, and should not be used for, patient connected applications.

How To Use This Programmer s Manual

Note: If you prefer to use DaqView, DaqViewXL, DASYLab, SnapMaster, or other out-of-the-box data acquisition software, you do not need to read this manual.

This manual explains how to program data acquisition systems using various APIs and programming languages. Besides the information in this manual, you must also read the user's manuals for your hardware. It may be helpful to read the DaqView chapter of the user's manual to appreciate how a user-friendly data acquisition system appears to the user. Also, you may need to consult documentation for your computer system and programming environment.

Everyone should read chapter 1 and then only the chapter(s) relevant to your programming environment.

After the table of contents, this manual is divided into a 6 chapters and an appendix as follows:

1. *Introduction* - The manual begins with an overview of issues related to programming a data acquisition system and what options are available to make this task as easy as possible. The various APIs and supported languages are introduced so you can determine which best fits your needs.
2. *Enhanced API Programming Models* describes the fundamental building blocks for data acquisition software. These programming blocks can then be arranged and filled with your parameters to make your system do as you please. Program excerpts illustrate the basic concepts and can often (with modifications) be used in your code.
3. *Daq* Command Reference (Enhanced API)* describes the commands and parameters of the "enhanced" API including useful reference tables.
4. *Standard API Programming Models* describes the fundamental building blocks for data acquisition software. These programming blocks can then be arranged and filled with your parameters to make your system do as you please. Program excerpts illustrate the basic concepts and can often (with modifications) be used in your code.
5. *Daq* Command Reference (Standard API)* describes the commands and parameters of the "standard" API including useful reference tables.
6. *Visual Basic VBX Support* explains the use of icon-based VBX programming tools.

Appendix. *Porting Applications* explains compatibility issues between APIs for Windows 3.1 and Windows 95/NT.

Table of Contents

1 Introduction

Overview ---------- 1-1
Driver Options ---------- 1-2
Standard API ---------- 1-2
Enhanced API ---------- 1-2
Language Support ---------- 1-2
16-Bit Standard API Languages ---------- 1-2
32-Bit Standard API Languages ---------- 1-3
Enhanced API Languages ---------- 1-3
Setup ---------- 1-4
Configuration ---------- 1-4

2 Enhanced API Programming Models

Overview ---------- 2-1
Data Acquisition Environment ---------- 2-1
Application Programming Interface (API) ---------- 2-1
Enhanced vs Standard API ---------- 2-1
Hardware Capabilities and Constraints ---------- 2-1
Signal Environment ---------- 2-2
Basic Models ---------- 2-2
Initialization and Error Handling ---------- 2-3
Foreground Acquisition with One-Step Commands ---------- 2-5
Counted Acquisitions Using Linear Buffers ---------- 2-6
Indefinite Acquisition, Direct-To-Disk Using Circular Buffers ---------- 2-8
Analog Output ---------- 2-11
Generating DAC FIFO Waveforms (DaqBoard Only) ---------- 2-12
Variable Rate, Variable Duty-Cycle Square-Wave Output ---------- 2-13
Digital I/O on P2 ---------- 2-15
Temperature Measurements Using Single TC Type on a Single DBK19 Card ---------- 2-17
Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards ---------- 2-25
Temperature Measurements Using Multiple RTDs on a Single DBK9 Card ---------- 2-29
Using DBK Card Calibration Files ---------- 2-31
Zero Compensation ---------- 2-34
Linear Conversion ---------- 2-36
Summary Guide of Selected Enhanced API Functions ---------- 2-38

3 Daq* Command Reference (Enhanced API)

Overview ---------- 3-1
Commands in Alphabetical Order ---------- 3-3
API Reference Tables ---------- 3-73
A/D Channel Descriptions ---------- 3-74
Daq Device Property Definitions ---------- 3-74
Digital I/O Port Connection ---------- 3-75
Event-Handling Definitions ---------- 3-76
Hardware Information Definitions ---------- 3-76
DBK Card Definitions ---------- 3-76
ADC Gain Definitions ---------- 3-77
ADC Trigger Source Definitions ---------- 3-78
ADC Miscellaneous Definitions ---------- 3-78
DAC Definitions ---------- 3-79
Data Conversion Definitions ---------- 3-79
WBK Card Definitions ---------- 3-80
General I/O Definitions ---------- 3-81
9513 Counter/Timer Definitions ---------- 3-82
daqTest Command Definitions ---------- 3-82
Calibration Input Signal Sources ---------- 3-82
API Error Codes ---------- 3-83

4 Standard API Programming Models

Overview ---------- 4-1
Data Acquisition Environment ---------- 4-1

Application Programming Interface (API) ---------- 4-1
Standard vs Enhanced API ---------- 4-1
Hardware Capabilities and Constraints ---------- 4-2
Signal Environment ---------- 4-2
Basic Models ---------- 4-3
Initialization and Error Handling ---------- 4-4
Foreground Acquisition with User-Level Commands ---------- 4-5
Foreground Acquisition with Low-Level Commands ---------- 4-7
Foreground Acquisition, High-Speed Digital Input ---------- 4-8
Background Acquisition, Multi-Channel, Multi-Scan ---------- 4-9
Background Acquisition, Direct-To-Disk In Cycle Mode ---------- 4-11
Analog Output ---------- 4-13
Generating DAC FIFO Waveforms with User-Level Commands (DaqBoard Only) ---------- 4-14
Generating DAC FIFO Waveforms with Hardware-Level Commands (DaqBoard) ---------- 4-16
Background Counter Acquisition Using Interrupts ---------- 4-18
Variable Rate, Variable Duty-Cycle Square-Wave Output ---------- 4-20
Single Square-Wave Output ---------- 4-22
Digital I/O on P2 ---------- 4-23
Temperature Measurements Using Single TC Type on a Single DBK19 Card ---------- 4-24
Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards ---------- 4-32
Temperature Measurements Using Multiple RTDs on a Single DBK9 Card ---------- 4-35
Using DBK Card Calibration Files ---------- 4-37
Zero Compensation ---------- 4-40
Linear Conversion ---------- 4-42
Summary Guide of Selected API Functions ---------- 4-44

5 Daq* Command Reference (Standard API)
Overview ---------- 5-1
Commands in Alphabetical Order ---------- 5-2
A/D Channel Descriptions ---------- 5-65
Thermocouple Types ---------- 5-65
A/D Trigger Source Definitions ---------- 5-65
A/D Gain Definitions ---------- 5-66
Digital I/O Port Connection ---------- 5-67
API Error Codes ---------- 5-68

6 Visual Basic VBX Support
Overview ---------- 6-1
DBK VBX ---------- 6-1
Event Routines - DBK ---------- 6-2
DBK Properties ---------- 6-2
ADC VBX ---------- 6-5
Event Routines - ADC ---------- 6-5
ADC Properties ---------- 6-6
CTR VBX ---------- 6-13
Event Routines - CTR ---------- 6-15
CTR Properties ---------- 6-16
DAC VBX ---------- 6-27
Event Routines - DAC ---------- 6-27
DAC Properties ---------- 6-27
DIO VBX ---------- 6-28
Event Routines - DIO ---------- 6-28
DIO Properties ---------- 6-28
Programming Examples ---------- 6-31
Example Summary ---------- 6-31

Appendix: Porting Applications
Overview ---------- A-1
Porting Daq* Applications Written for Windows 3.1 ---------- A-2
Windows 3.1 Binary Compatibility (16-bit) ---------- A-2
Unsupported Windows 3.1 API Functions ---------- A-2
Porting Visual Basic Programs ---------- A-3
Porting C Programs ---------- A-4

Notes

Introduction 1

Overview

New operating systems and new hardware have led to a new Application Programming Interface (API) for Daq* and related products (the enhanced API). For users building on previously written programs, the standard API will continue to be supported.

- The ***User s Manual*** describes hardware installation and setup, theory of operation, troubleshooting, and ready-to-run software. If you plan to use the DaqView software shipped with the system (or other ready-to-run software such as DASYLab or SnapMaster), the user's manual is all you need.
- The ***Programmer s Manual*** describes the API for programmers who are creating custom software for their particular application. **Note**: to create effective programs, programmers must also be familiar with the hardware and operation as described in the user's manual.

This *Programmer s Manual* covers several APIs used with various products. Often, a product is shipped with several APIs to accommodate various hardware environments and programming preferences. Thus, not all of this manual will apply to your system. After reading this introduction, you can then read only the relevant chapters. **Note**: the *readme* files on the driver disks will keep you up-to-date as the APIs continue to evolve.

This manual currently covers several APIs and programming environments for the Daq* product line including all models of DaqBook, DaqBoard, Daq PCMCIA, and DBK Option Cards and Modules. **Note:** As of this writing, the Daq PCMCIA is not supported under Windows95/NT drivers (the enhanced API).

This manual serves both novice and experienced programmers.

- As a tutorial - The *Programming Models* chapters (enhanced and standard) explain how to combine commands to do useful work in a typical data acquisition environment. Program excerpts illustrate the concepts and can be modified as needed to use in your programs.
- As a reference - Much of this manual is a detailed description of the API commands. These details and the related tables of parameter values and definitions are important to ensure proper syntax and that your programs run as intended.

Note: This manual is not a tutorial on computer programming in general. You may need to consult other texts for such information.

Driver Options

This section is intended to help you decide which API and programming language to use in developing your application. The install disks include several “drivers” to accommodate various programming environments.

Standard API

The standard API was originally written for the Daq* Windows 3.1 driver. However, it can be used under Windows 95 in 16- or 32-bit mode or under Windows NT in 32-bit mode. The standard API is the only API option available for Windows 3.1 or DOS applications. Use the Standard API:

- When developing a new or existing DOS application
- When developing a new or existing Windows 3.1 application
- When a quick port of an existing 16-bit mode (Windows 3.1) application to 32-bit mode (Windows95/NT) is required

Enhanced API

The Enhanced API has several features that are not present in the standard API:

- Multi-device - can concurrently handle up to 4 devices of the Daq* family
- Larger buffer - can handle up to 2 billion samples at a time
- Enhanced acquisition and trigger modes
- Direct-to-disk capabilities
- Wait-on-event features
- Uses multi-tasking advantages of Windows 95/NT

Because of these new features and other improvements, it is recommended to use the Enhanced API, when feasible. Use the Enhanced API:

- When developing new or existing Windows 95 applications
- When developing new or existing Windows NT applications
- When porting an existing Standard API application to 32-bit mode to take advantage of the Enhanced API features

Language Support

The following table shows language support for the standard and enhanced API drivers.

Standard API (16-bit) Supported Languages	Enhanced API (or 32-bit Standard) Supported Languages
C/C++ Microsoft Visual C++ Borland C++ (v4.0 and greater)	C/C++ Microsoft Visual C++ Borland C++ (v4.0 and greater)
BASIC Microsoft Visual Basic (v4.0 and greater) QuickBASIC	BASIC Microsoft Visual Basic (v4.0 and greater)
Pascal Turbo Pascal	Delphi Borland Delphi (v2.0)

16-bit Standard API Languages

C (for Windows)

There is one library and one header file located in the DAQBOOK\WIN\C directory. The header file, DAQBOOK.H, must be included at the top of a C program using the `#include` command. This will allow the compiler to know what Daq* functions and constants are available.

The library, DAQBOOK.LIB, must be included in the applications makefile or project file so that the linker will find the Daq* functions. See the documentation for your specific C compiler for a description on using header files and libraries.

To use the example program located in the DAQBOOK\DOS\C directory, create a makefile or project file which consists of the DAQEX.C source file, DAQEX.RC resource file, DAQEX.DEF definition file, DAQEX.ICO icon file, and the DAQBOOK.LIB library. **Note**: The file DAQBOOK.DLL must be present in the WINDOWS directory. (If necessary, the file DAQBOOK.DLL can be copied from the DAQBOOK\WIN directory.)

QuickBASIC

Basic interface, library, and quick library files are located in the DAQBOOK\DOS\QB directory.

- The Basic interface file DAQBOOK.BI must be included at the top of a QuickBASIC program using the **'$INCLUDE** command (**'$INCLUDE: 'daqbook.bi'**). This will allow QuickBASIC to know what Daq* functions and constants are available.
- The library DAQBOOK.LIB must be included during the link process when creating a program from the DOS command line. The **/NOE** option of the linker may be necessary when linking the Daq* library.
- Alternatively, the quick library DAQBOOK.QLB can be used to access the Daq* from within the QuickBASIC environment. Use the **/L** option of QuickBASIC to load the appropriate Quick Library. See the QuickBASIC documentation for the various command line options.

To run an example program located in the DAQBOOK\DOS\QB directory, start QuickBASIC using the /L option, such as QB /LDAQBOOK.QLB. Then load and run the desired program. The example program could also be compiled using QuickBASIC's BC.EXE compiler to create an .OBJ file. This .OBJ could then be linked to the DAQBOOK.LIB file using QuickBASIC's LINK.EXE linker.

If you need to use more than one quick library with your application program, you will need to create a combined library. The first step is to extract the object modules from DAQBOOK.LIB using the LIB program provided with QuickBASIC:

```
C:\QB45 LIB daqbook *lowqb *highqb *highcqb *stubstb *tcqb
```

Next, you need to link the object modules along with your other libraries into the combined Quick Library using the LINK program provided with QuickBASIC. The following example creates a Quick Library called COMBINED.QLB from the Daq* object modules and USEROBJ.OBJ:

```
C:\QB45LINK
Object Modules [.OBJ]: lowqb+highbqb+hgihcqb+stubsqb+tcqb+userobj
Run File [LOWQB.EXE]: combined.qbl /q
List File [NUL.MAP]: /noe
Libraries [.LIB]: bqlb45
```

Turbo Pascal

To use the example programs located in the DAQBOOK\DOS\TP7 directory, make sure that your program specifies DAQBOOK.TPU unit in the uses clause. Also be sure that the DAQBOOK.TPU unit file is in the Turbo Pascal search path.

32-bit Standard API Languages

C/C++ For native Microsoft Visual C++, support is located in **<installationpath>\C\32Std**. For Borland C++ (v4.0 and greater) via dynamic linking, support is located in **<installationpath>\C\32Std\Dynamic**.

Visual Basic For Microsoft Visual Basic (v4.0 and greater), support is located in **<installationpath>\VB\32Std**.

Delphi For Borland Delphi (v2.0), support is located in **<installation path>\Delphi\32Std**.

Enhanced API Languages

C/C++ For native Microsoft Visual C++, find enhanced API support in **<installationpath>\C\32Enh**. For Borland C++ (v4.0 and greater) via dynamic linking, support is located in **<installationpath>\C\32Enh\Dynamic**.

Visual Basic For Microsoft Visual Basic (v4.0 and greater), support is located in **<installationpath>\VB\32Enh**.

Delphi For Borland Delphi (v2.0), support is located in **<installationpath>\Delphi\32Enh**.

Setup

Driver installation uses one of two disk sets:

- If installing on a DOS or Windows 3.1 system, use the DaqBook/DaqBoard Software disk set (#232-0601).
- If installing on a Windows 95 or Windows NT system, use the DaqBook/DaqBoard Software for Windows95/NT disk set (#443-0601).

In either case, the setup (***Setup.Exe)*** routine is located on Disk 1 of the respective disk set. When run, the setup routine will automatically detect on which operating system the installation is being performed and will install the appropriate driver. For more details on the setup process, see sections that discuss installation for the specific Daq* device (in the *User s Manual*: chapter 2 for DaqBook, chapter 3 for DaqBoard, chapter 5 for specific DBK cards and modules).

Configuration

For details on configuring specific Daq* devices, see related sections that discuss that specific Daq* device (in the *User s Manual*: chapter 2 for DaqBook, chapter 3 for DaqBoard, chapter 5 for specific DBK cards and modules).

Enhanced API Programming Models 2

Overview

By using the Application Programming Interface (API) with Daq* systems, you can create custom software to satisfy your data acquisition requirements. Chapter 3 explains the enhanced API functions in detail. This chapter shows how to combine API functions to perform typical tasks. When you understand how the API works with the hardware, you are ready to program for optimum data acquisition. To help you get this perspective, this chapter is divided into 3 parts:

- **Data Acquisition Environment** outlines related concepts and defines Daq* capabilities the programmer must work with (the API, hardware features, and signal management).
- **Programming Models** explains the sequence and type of operations necessary for data acquisition. These models provide the software building blocks to develop more complex and specialized programs. The description for each model has a flowchart and program excerpt to show how the API functions work.
- **Summary Guide of Selected API Functions** is an easy-to-read table that describes when to use the basic API functions.

Data Acquisition Environment

In order to write effective data acquisition software, programmers must understand:

- Software tools (the API documented in this manual and the programming language—you may need to consult documentation for your chosen language)
- Hardware capabilities and constraints
- General concepts of data acquisition and signal management

Application Programming Interface (API)

The API includes all the software functions needed for building a data acquisition system with the hardware described in this manual. Chapter 3 (*Daq* Command Reference   Enhanced API)* supplies the details about how each function is used (parameters, hardware applicability, etc). In addition, you may need to consult your language and computer documentation.

Enhanced vs Standard API

Major differences between the enhanced and standard APIs were described in the introductory chapter. Language support varies as follows:

- The **enhanced** API (32-bit only) accommodates C, Visual Basic, and Delphi.
- The **standard** API accommodates C (16- or 32-bit), QuickBASIC (16-bit only), Visual Basic (16- or 32-bit), and Turbo Pascal 7 (16-bit only).

Note: Coding for the enhanced and standard API cannot be used together; enhanced and standard models are slightly different (this chapter is for the enhanced API models; chapter 4 is for the standard API models).

Hardware Capabilities and Constraints

To program the system effectively, you must understand your Daq* and DBK hardware capabilities. Obviously you cannot program the hardware to perform beyond its design and specifications, but you also want to take full advantage of the system's power and features. In the *User s Manual*, you may need to refer to sections that describe your hardware's capability. In addition, you may need to consult your computer documentation. In some cases, you may need to verify the hardware setup, use of channels, and signal conditioning options (some hardware devices have jumpers and DIP switches that must match the programming, especially as the system evolves).

Signal Environment

Important data acquisition concepts for programmers are listed here and explained in the chapter *Signal Management and Troubleshooting Tips* in the *User s Manual*. You must apply these concepts as needed in your situation. Some of these concepts include:

- **Channel Identification**. Refer to *Signal Management* and the related reference table in chapter 3.
- **Scan Rates and Sequencing**. With multiple scans, the time between scans becomes a parameter. This time can be a constant or can be dependent upon a trigger. Refer to *Signal Management* .
- **Counter/Timer Operation**. Refer to *Signal Management* and `daq9513…` functions in chapter 3.
- **Triggering Options**. Triggering starts the A/D conversion. The trigger can be an external analog or TTL trigger, or a program controlled software trigger. Refer to *Signal Management* and the trigger functions in chapter 3.
- **Foreground/Background**. Foreground transfer routines require the entire transfer to occur before returning control to the application program. Background routines start the A/D acquisition and return control to the application program before the transfer occurs. Data is transferred while the application program is running. Data will be transferred to the user memory buffer during program execution in 1 sample or 256 sample blocks, depending on the configuration. The programmer must determine what tasks can proceed in the background while other tasks perform in the foreground and how often the status of the background operations should be checked.

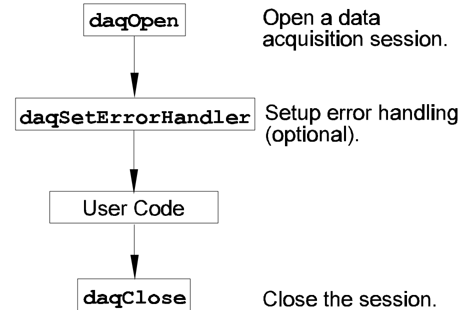
Parameters in the various A/D routines include: number of channels, number of scans, start of conversion triggering, timing between scans, and mode of data transfer. Up to 512 A/D channels can be sampled in a single scan. These channels can be consecutive or non-consecutive with the same or different gains. The scan sequence makes no distinction between local and expansion channels.

Basic Models

This section outlines basic programming steps commonly used for data acquisition. Consider the models as building blocks that can be put together in different ways or modified as needed. As a general tutorial, these examples use Visual Basic since most programmers know BASIC and can translate to other languages as needed. The enhanced API programming models discussed in this chapter include:

Model Type	Model Name	Page
Configuration	Initialization and Error Handling	2-3
Acquisition	Foreground Acquisition with One-Step Commands Counted Acquisition Using Linear Buffers Indefinite Acquisition, Direct-To-Disk Using Circular Buffers	2-5 2-6 2-8
Analog Output	Analog Output Generating DAC FIFO Waveforms (DaqBoard Only)	2-11 2-12
Use of P3 s Counter/Timer	Variable Rate, Variable Duty-Cycle Square-Wave Output	2-13
Use of 8255 Chip	Digital I/O on P2	2-15
Temperature Measurements	Temperature Measurements Using Single TC Type on Single DBK19 Card Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards Temperature Measurements Using Multiple RTDs on a Single DBK9 Card	2-17 2-25 2-29
Calibration	Using DBK Card Calibration Files	2-31
Zero Compensation	Zero Compensation	2-34
Conversion	Linear Conversion	2-36

Initialization and Error Handling



This section demonstrates how to initialize the Daq* and use various methods of error handling. Most of the example programs use similar coding as detailed here. Functions used include:

- **`VBdaqOpen&(daqName$)`**
- **`VBdaqSetErrorHandler&(errHandler&)`**
- **`VBdaqClose&(handle&)`**

All Visual Basic programs should include the DaqX.bas file into their project. The DaqX.bas file provides the necessary definitions and function prototyping for the DAQX driver DLL.

```
handle& = VBdaqOpen&("DaqBook0")
ret& = VBdaqClose&(handle&)
```

The Daq* device is opened and initialized with the **`daqOpen`** function. **`daqOpen`** takes one parameter—the name of the device to be opened. The device name information can be accessed or changed via the Daq* Configuration utility located in the operating system's Control Panel. The **`daqOpen`** call, if successful, will return a *handle* to the opened device. This handle may then be used by other functions to configure or perform other operations on the device. When operations with the device are complete, the device may then be closed using the **`daqClose`** function. If the device could not be found or opened, **`daqOpen`** will return **`-1`**.

The DAQX library has a default error handler defined upon loading. However; if it is desirable to change the error handler or to disable error handling, then the **`daqSetErrorHandler`** function may be used to setup an error handler for the driver. In the following example the error handler is set to **`0`** (no handler defined) which disables error handling.

```
ret& = VBdaqSetErrorHandler&(0&)
```

If there is a Daq* error, the program will continue. The function's return value (an error number or **`0`** if no error) can help you debug a program.

```
If (VBdaqOpen&("DaqBook0") <  0) Then
  "Cannot open DaqBook0"
```

Daq* functions return **`daqErrno&`**.

```
  Print "daqErrno& : "; HEX$(daqErrno&)
End If
```

The next statement defines an error handling routine that frees us from checking the return value of every Daq* function call. Although not necessary, this sample program transfers program control to a user-defined routine when an error is detected. Without a Daq* error handler, Visual Basic will receive and handle the error, post it on the screen and terminate the program. Visual Basic provides an integer variable (ERR) that contains the most recent error code. This variable can be used to detect the error source and take the appropriate action. The function **`daqSetErrorHandler`** tells Visual Basic to assign ERR to a specific value when a Daq*error is encountered. The following line tells Visual Basic to set ERR to 100 when a Daq*error is encountered. (Other languages work similarly; refer to specific language documentation as needed.)

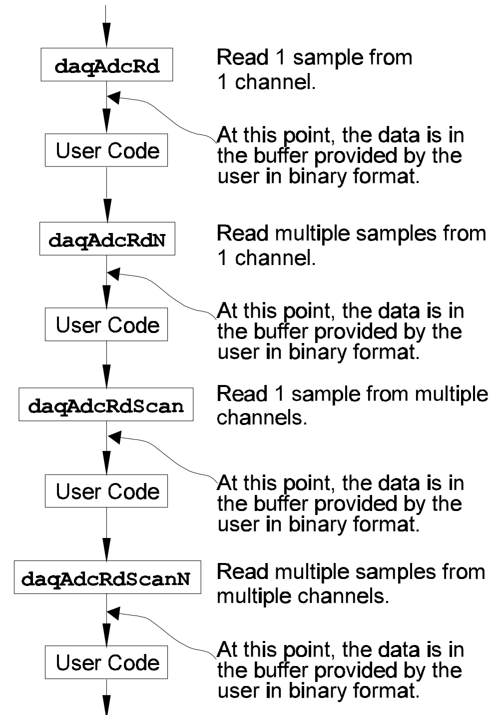
```
handle& = VBdaqOpen&("DaqBook0")
ret& = VBdaqSetErrorHandler&(handle&, 100)

   On Error GoTo ErrorHandler
```

The **`On Error GoTo`** command in Visual Basic allows a user-defined error handler to be provided, rather than the standard error handler that Visual Basic uses automatically. The program uses **`On Error GoTo`** to transfer program control to the label ErrorHandler if an error is encountered.

Daq* errors will send the program into the error handling routine. This is the error handler. Program control is sent here on error.

```
ErrorHandler:

   errorString$ = "ERROR in ADC1"
   errorString$ = errorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
   If Err = 100 Then errorString$ = errorString$ & Chr(10) & "DaqBook
 Error : " + Hex$(daqErrno&)

   MsgBox errorString$, , "Error!"

End Sub
```

Foreground Acquisition with One-Step Commands



This section shows the use of several one-step analog input routines. These commands are easier to use than low-level commands but less flexible in scan configuration. These commands provide a single function call to configure and acquire analog input data. This example demonstrates the use of the 4 Daq*'s one-step ADC functions. Functions used include:

- **`VBdaqAdcRd&(handle&,chan&, sample%, gain&)`**
- **`VBdaqAdcRdN&(handle&,chan&, Buf%(), count&, trigger%, level%, freq!, gain&,flags&)`**
- **`VBdaqAdcRdScan&(handle&,startChan&, endChan&, Buf%(), gain&, flags&)`**
- **`VBdaqAdcRdScanN&(handle&,startChan&, endChan&, Buf%(), count&, triggerSource&, level%, freq!, gain&, flags&)`**

This program will initialize the Daq* hardware, then take readings from the analog input channels in the base unit (not the expansion cards). For transporting data in and out of the Daq* driver, arrays are dimensioned.

```
Dim sample%(1), buf%(80), handle&,
  ret&, flags&, gain&
```

The following code assumes that the Daq* device has been successfully opened and the **`handle&`** value is a valid handle to the device. All the following one-step functions define the channel scan groups to be analog unipolar input channels. Specifying this configuration uses the **`DafAnalog`** and the **`DafUnipolar`** values in the **`flags`** parameter. The **`flags`** parameter is a bit-mask field in which each bit specifies the characteristics of the channel(s) specified. In this case, the **`DafAnalog`** and the **`DafUnipolar`** values are added together to form the appropriate bit mask for the specified **`flags`** parameter.

The next line requests 1 reading from 1 channel with a gain of ×1. The variable **`DgainX1&`** is actually a defined constant from DaqX.bas, included at the beginning of this program.

```
ret& = VBdaqAdcRd&(handle& 0, sample%(0), DgainX1&,
  DafAnalog&+DafUnipolar&)
Print Format$"&   ####"; "Result of AdcRd:"; sample%(0)
```

The next line requests 10 readings from channel 0 at a gain of ×1, using immediate triggering at 1 kHz.

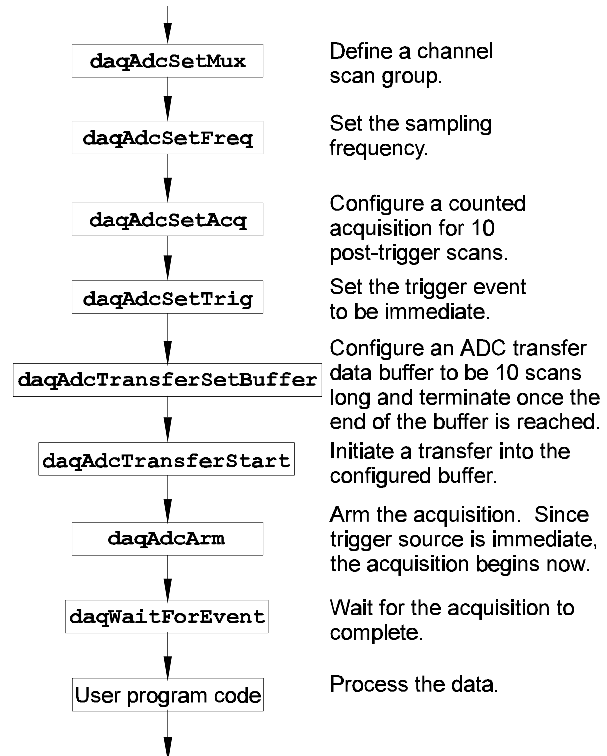
```
ret& = VBdaqAdcRdN&(handle&,0, buf%(), 10, DatsImmediate&, 0, 1000!,
  DgainX1&, DafAnalog&+DafUnipolar&)
Print "Results of AdcRdN: ";
For x& = 0 To 9
   Print Format$ "####  "; buf%(x&);
Next x&
```

The program will then collect one sample of channels 0 through 7 using the **`VBdaqAdcRdScan`** function.

```
ret& = VBdaqAdcRdScan&(handle&,0, 7, buf%(), DgainX1&,
  DafAnalog&+DafUnipolar&)

Print "Results of AdcRdscan:"
For x& = 0 To 7
  Print Format$"& #  &  ####"; "Channel:"; buf%(x); "Data:"; buf%(x)
Next x&: Print
```

Counted Acquisitions Using Linear Buffers

This section sets up an acquisition that collects post-trigger A/D scans. This particular example demonstrates the setting up and collection of a fixed-length A/D acquisition in a linear buffer.



First, the acquisition is configured by setting up the channel scan group configuration, the acquisition frequency, the acquisition trigger and the acquisition mode. When configured, the acquisition is then armed by calling the **`daqAdcArm`** function.

At this point, the Daq* device trigger is armed and A/D acquisition will begin upon trigger detection. If the trigger source has been configured to be **`DatsImmediate&`**, A/D data collection will begin immediately.

This example will retrieve 10 samples from channels 0 through 7, triggered immediately with a 1000 Hz sampling frequency and unity gain. Functions used include:

- **`VBdaqAdcSetMux&(handle&, startChan&, endChan&, gain&, flags&)`**
- **`VBdaqAdcSetFreq&(handle&,freq!)`**
- **`VBdaqAdcSetTrig&(handle&, triggerSource&, rising&, level%, hysteresis%,channel&)`**
- **`VBdaqAdcSetAcq&(handle&,mode&,preTrigCount&,postTrigCount&)`**
- **`VBdaqAdcTransferSetBuffer&(handle&,buf%(), scanCount&, transferMask&)`**
- **`VBdaqAdcTransferStart&(handle&)`**
- **`VBdaqAdcWaitForEvent&(handle&,daqEvent&)`**

This program will initialize the Daq* hardware, then take readings from the analog input channels in the base unit (not the expansion cards). The functions used in this program are of a lower level than those used in the previous section and provide more flexibility.

```
Dim buf%(80), handle&, ret&, flags&
```

The following function defines the channel scan group. The function specifies a channel scan group from channel 0 through 7 with all channels being analog unipolar input channels with a gain of ×1. Specifying this configuration uses **`DgainX1`** in the gain parameter and the **`DafAnalog`** and the **`DafUnipolar`** values in the **`flags`** parameter. The **`flags`** parameter is a bit-mask field in which each bit specifies the characteristics of the specified channel(s). In this case, the **`DafAnalog`** and the **`DafUnipolar`** values are added together to form the appropriate bit mask for the specified **`flags`** parameter.

```
ret& = VBdaqAdcSetMux&(handle&,0, 7, DgainX1&, DafAnalog&+DafUnipolar&)
```

Next, set the internal sample rate to 1 kHz.

```
ret& = VBdaqAdcSetFreq&(handle&,1000!)
```

The acquisition mode needs to be configured to be fixed length acquisition with no pre-trigger scan data and 10 scans of post-trigger scan data. The mode is set to **`DaamNShot&`**, which will configure the acquisition as a fixed-length acquisition that will terminate automatically upon the satisfaction of the post-trigger count of 10.

```
ret& = VBdaqAdcSetAcq&(handle&,DaamNShot&, 0, 10)
```

The acquisition begins upon detection of the trigger event. The trigger event is configured with **daqAdcSetTrig**. The next line defines the trigger event to be the immediate trigger source which will start the acquisition immediately. The variable **DatsImmediate&** is a constant defined in DaqX.bas. Since the trigger source is configured as immediate, the other trigger parameters are not needed.

```
ret& = VBdaqAdcSetTrig&(handle&,DatsImmediate&, 0, 0, 0, 0)
```

A buffer now is configured to hold the A/D data to be acquired. Since this is to be a fixed length transfer to a linear buffer, the buffer cycle mode should be turned off with **DatmCycleOff&**. For efficiency, block update mode is specified with **DatmUpdateBlock&**. The buffer size is set to 10 scans. **Note**: the user-defined buffer must have been allocated with sufficient storage to hold the entire transfer prior to invoking the following line.

```
ret& = VBdaqAdcTransferSetBuffer&(handle&,buf%(), 10,
 DatmUpDateBlock&+DatmCycleOff&)
```

With all acquisition parameters being configured, the acquisition can now be armed. Once armed, the acquisition will begin immediately upon detection of the trigger event. As in the case of the immediate trigger, the acquisition will begin immediately upon execution of the **daqAdcArm** function.

```
ret& = VBdaqAdcArm&(handle&)
```

After setting up and arming the acquisition, the data is immediately ready to be collected. Had the trigger source been anything other than immediate, the data would only be ready after the trigger had been satisfied. The following line initiates an A/D transfer from the Daq* device to the defined user buffer.

```
ret& = VBdaqAdcTransferStart&(handle&)
```

Wait for the transfer to complete in its entirety, then proceed with normal application processing. This can be accomplished with the **daqWaitForEvent** command. The **daqWaitForEvent** allows the application processing to become blocked until the specified event has occurred. **DteAdcDone**, indicates that the event to wait for is the completion of the transfer.

```
ret& = VBdaqWaitForEvent(handle&,DteAdcDone&)
```

At this point, the transfer is complete; all data from the acquisition is available for further processing.

```
Print “Results of Transfer”
For i& = 0 To 10
    Print "Scan "; Format$(Str$(i& + 1), "00"); " -->";
    For k& = k& To k& + 7
        Print Format$(IntToUint&(buf%(k&)), "00000"); "  ";
    Next k&
    Print
Next i&
```

Indefinite Acquisition, Direct-To-Disk Using Circular Buffers

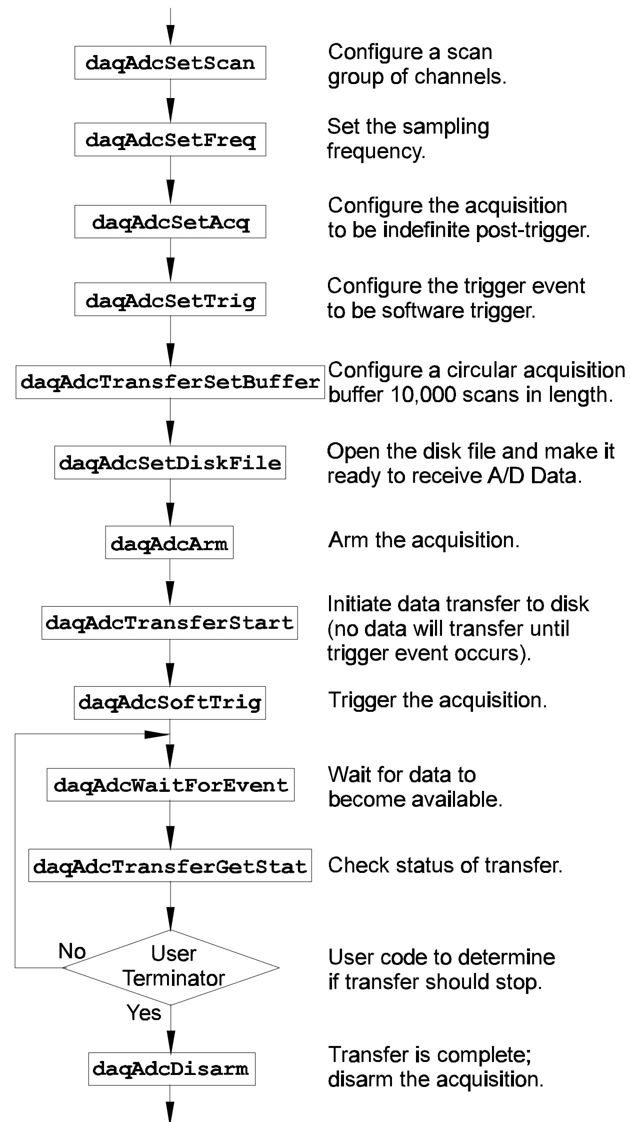
This program demonstrates the use of circular buffers in cycle mode to collect analog input data directly to disk. In cycle mode, this data transfer can continue indefinitely. When the transfer reaches the end of the physical data array, it will reset its array pointer back to the beginning of the array and continue writing data to it. Thus, the allocated buffer can be used repeatedly like a FIFO buffer.

Unlike the Standard API, the Enhanced API has built-in direct-to-disk functionality. Therefore, very little needs to be done by the application to configure direct-to-disk operations.

First, the acquisition is configured by setting up the channel scan group configuration, the acquisition frequency, the acquisition trigger and the acquisition mode. Once configured, the transfer to disk is set up and the acquisition is then armed by calling the **daqAdcArm** function.

At this point, the Daq* device trigger is armed and A/D acquisition to disk will begin immediately upon trigger detection.

This example will retrieve an indefinite amount of scans for channels 0 through 7, triggered via software with a 3000 Hz sampling frequency and unity gain. Functions used include:



- **VBdaqAdcSetScan&(handle&, startChan&, endChan&, gain&, flags&)**
- **VBdaqAdcSetFreq&(handle&,freq!)**
- **VBdaqAdcSetTrig&(handle&, triggerSource&, rising&, level%, hysteresis%,channel&)**
- **VBdaqAdcSetAcq&(handle&,mode&,preTrigCount&,postTrigCount&)**
- **VBdaqAdcTransferSetBuffer&(handle&,buf%(), scanCount&, transferMask&)**
- **VBdaqAdcTransferStart&(handle&)**
- **VBdaqAdcTransferGetStat&(handle&,status&,retCount&)**
- **VBdaqAdcWaitForEvent&(handle&,daqEvent&)**
- **VBdaqAdcSetDiskFile&(handle&,filename$,openMode&,preWrite&)**

This program will initialize the Daq* hardware, then take readings from the analog input channels in the base unit (not the expansion cards) and store them to disk automatically. The following lines demonstrate channel scan group configuration using the **daqAdcSetScan** command. **Note**: flags may be channel-specific.

```
Dim handle&, ret&, channels&(8), gains&(8) flags&(8)
Dim buf%(80000), active&, count&
Dim bufsize& = 10000            ' In scans
```

```
' Define arrays of channels and gains : 0-7 , unity gain
For x& = 0 To 7
  channels&(x&) = x&
  gains&(x&) = DgainX1&
  flags&(x&) = DafAnalog& + DafSingleEnded& + DafUnipolar&
Next x&

' Load scan sequence FIFO
ret& = VBdaqAdcSetScan&(handle&,channels&(), gains&(), flags&(), 8)
```

Next, set the internal sample rate to 3 kHz.

```
ret& = VBdaqAdcSetFreq&(handle&,3000!)
```

The acquisition mode needs to be configured to be fixed-length acquisition with no pre-trigger scan data and 10 scans of post-trigger scan data. The mode is set to **DaamInfinitePost&**, which will configure the acquisition as having indefinite length and, as such, will be terminated by the application. In this mode, the pre- and post-trigger count values are ignored.

```
ret& = VBdaqAdcSetAcq&(handle&,DaamInfinitePost&, 0, 0)
```

The acquisition begins upon detection of the trigger event. The trigger event is configured with **daqAdcSetTrig**. The next line defines the trigger event to be the immediate trigger source which will start the acquisition immediately. The variable **DatsSoftware&** is a constant defined in DaqX.bas. Since the trigger source is configured as immediate, the other trigger parameters are not needed.

```
ret& = VBdaqAdcSetTrig&(handle&,DatsSoftware&, 0, 0, 0, 0)
```

A buffer now is configured to hold the A/D data to be acquired. This buffer is necessary to hold incoming A/D data while it is being prepared for disk I/O. Since this is to be an indefinite-length transfer to a circular buffer, the buffer cycle mode should be turned on with **DatmCycleOn&**. For efficiency, block update mode is specified with **DatmUpdateBlock&**. The buffer size is set to 10,000 scans. The buffer size indicates only the size of the circular buffer, not the total number of scans to be taken.

```
ret& = VBdaqAdcTransferSetBuffer&(handle&,buf%(), bufsize&,
  DatmUpDateBlock&+DatmCycleOn&)
```

Now the destination disk file is configured and opened. For this example, the disk file is a new file to be created by the driver. After the following line has been executed, the specified file will be opened and ready to accept data.

```
ret& = VBdaqAdcSetDiskFile&(handle&,"c:dasqdata.bin", DaomCreateFile&, 0)
```

With all acquisition parameters being configured and the acquisition transfer to disk configured, the acquisition can now be armed. Once armed, the acquisition will begin immediately upon detection of the trigger event. As in the case of the immediate trigger, the acquisition will begin immediately upon execution of the **daqAdcArm** function.

```
ret& = VBdaqAdcArm&(handle&)
```

After setting up and arming the acquisition, data collection will begin upon satisfaction of the trigger event. Since the trigger source is software, the trigger event will not take place until the application issues the software trigger event. To prepare for the trigger event, the following line initiates an A/D transfer from the Daq* device to the defined user buffer and, subsequently, to the specified disk file. No data is transferred at this point, however.

```
ret& = VBdaqAdcTransferStart&(handle&)
```

The transfer has been initiated, but no data will be transferred until the trigger event occurs. The following line will signal the software trigger event to the driver; then A/D input data will be transferred to the specified disk file as it is being collected.

```
ret& = VBdaqAdcSoftTrig&(handle&)
```

Both the acquisition and the transfer are now currently active. The transfer to disk will continue indefinitely until terminated by the application. The application can monitor the transfer process with the following lines of code:

```
acqTermination& = 0
Do
    ' Wait here for new data to arrive
    ret& = VBdaqWaitForEvent(handle&,DteAdcData&)

    ' New data has been transferred - Check status
    ret& = VBdaqAdcTransferGetStat&(handle&,active&,retCount&)

    ' Code may be placed here which will process the buffered data or
    ' perform other application activities.
    '
    ' At some point the application needs to determine the event on which
    ' the direct-to-disk acquisition is to be halted and set the
    ' acqTermination flag.

Loop While acqTermination& = 0
```

At this point the application is ready to terminate the acquisition to disk. The following line will terminate the acquisition to disk and will close the disk file.

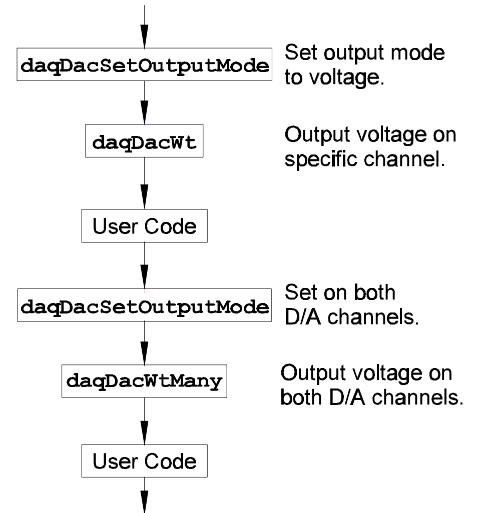
```
ret& = VBdaqAdcDisarm&(handle&)
```

The acquisition as well as the data transfer has been stopped. We should check status one more time to get the total number of scans actually transferred to disk.

```
ret& = VBdaqAdcTransferGetStat(handle&,active&,retCount&)
```

The specified disk file is now available. The **retCount&** parameter will indicate the total number of scans transferred to disk.

Analog Output



The program DACEX1.BAS shows how to output analog voltages on analog output channels 0 and 1. These commands only have to be issued one time unless a related parameter is explicity changed. The output voltages will be sustained. This example demonstrates the use of the two digital-to-analog converters (values used assume bipolar mode). Functions used include:

- **VBdaqDacSetOutputMode&(handle&, DddtLocal&, 0, DdomVoltage&)**
- **VBdaqDacWt&(handle&, deviceType&, chan&, dataVal%)**
- **VBdaqDacWtMany&(handle, deviceTypes&(),chans&(), dataVals%())**

Assuming the voltage reference is connected to the internal default of 5 V, the next function will set channel 0 to an output voltage of 5 V. The values are set for a digital-to-analog converter with 16 bit resolution; 65535 represents full-scale. Channel 1 is equal to 0.

```
ret& = VBdaqDacSetOutputMode&(handle&, DddtLocal&, 0, DdomVoltage&)
ret& = VBdaqDacWt&(handle&, DddtLocal, 0, 65535)
```

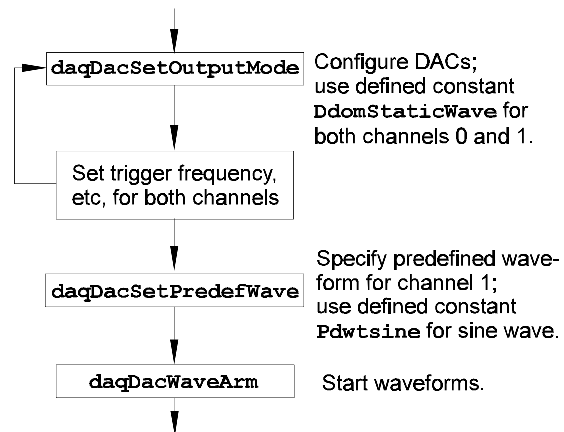
The **daqDacWtMany** writes to both analog outputs simultaneously. The following lines sets channel 0 to 5 V and channel 1 to 2.5 V. At full-scale, a digital value of 65535 corresponds to 5 V; a digital value of 49152 corresponds to ½ of 5 V.

```
Dim deviceTypes&(1)
Dim chans&(1)
Dim dataVals%(1)
The VBdaqSetOutputMode puts the channel in a voltage mode.
ret& = VBdaqSetOutputMode&(handle&, DddtLocal&, 0, DdomVoltage&)
ret& = VBdaqSetOutputMode&(handle&, DddtLocal&, 1, DdomVoltage&)
deviceTypes&(0) = DddtLocal&
deviceTypes&(1) = DddtLocal&
chans&(0) = 0
chans&(1) = 1
dataVals&(0) = 65535
dataVals&(1) = 49152
ret& = VBdaqDacWtMany&(handle&, deviceTypes&(), chans&(), dataVals%(),2)
```

The following sets the outputs to 0 V.

```
Dim deviceTypes&(1)
Dim chans&(1)
Dim dataVals%(1)
deviceTypes&(0) = DddtLocal&
deviceTypes&(1) = DddtLocal&
chans&(0) = 0
chans&(1) = 1
dataVals&(0) = 32768
dataVals&(1) = 32768
ret& = VBdaqSetOutputMode&(handle&, DddtLocal&, 0, DdomVoltage&)
ret& = VBdaqSetOutputMode&(handle&, DddtLocal&, 1, DdomVoltage&)
ret& = VBdaqDacWtMany&(handle&, deviceTypes&(), chans&(), dataVals%(),2)
```

Generating DAC FIFO Waveforms (DaqBoard Only)



This program demonstrates the use of the DAC FIFO to generate waveforms. The DAC is configured for output on both channels, and the user waveform is constructed. Output begins after the waveform is assigned to a channel. At this point, the program continues while the waveforms are generated.

The following example shows how to generate a pre-defined waveform using these functions:

- **`VBdaqDacWaveSetTrig&(handle&, deviceType&, chan&, triggerSource&, rising%)`**
- **`VBdaqDacWaveSetClockSource&(handle&, deviceType&, chan&, clockSource&)`**
- **`VBdaqDacWaveSetFreq&(handle&, deviceType&, chan&, freq!)`**
- **`VBdaqDacWaveSetMode&(handle&, deviceType&, chan&, mode&, updateCount&)`**
- **`VBdaqDacWaveSetBuffer&(handle&, deviceType&, chan&, buf%(), scanCount&, transferMask&)`**
- **`VBdaqDacWaveSetPredefWave&(handle&, deviceType&, chan&, waveType&, amplitude&, offset&, dutyCycle&, phaseShift&)`**
- **`VBdaqDacWaveArm&(ByVal handle&, ByVal deviceType&)`**

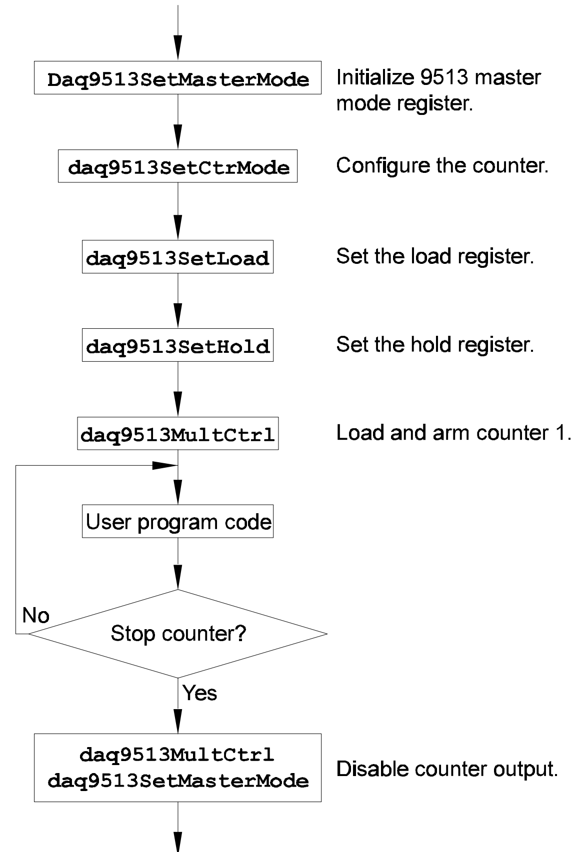
When using the pre-defined waveform generation, program the waveform parameters common to both channels. The double star (**) indicates the value must be the same on both channels of a DaqBoard.

```
For chan = 0 To 1 Step 1
  ' set the output mode to static waveform
     ret& = VBdaqDacSetOutputMode&(handle&, DddtLocal&, chan&,
    DdomStaticWave&)
  ' The trigger source must be set to immediate for static waveform.**
  err& = VBdaqDacWaveSetTrig&(handle&, DddtLocal&, chan&, DdtsImmediate&,
  1)
  ' set the internal dac clock
  ret& = VBdaqDacWaveSetClockSource&(handle&, DddtLocal&, chan&,
  DdcsDacClock&)
  ' the frequency of the internal clock. **
  ret& = VBdaqDacWaveSetFreq&(handle&, DddtLocal&, chan&, 10!)
  ' must be infinite for static mode
  ret& = VBdaqDacWaveSetMode&(handle&, DddtLocal&, chan&, DdwmInfinite&,
  0)
Next chan
' buffer cylce on, retransmit mode. **
' update count is the buffer length. **
ret& = VBdaqDacWaveSetBuffer&(handle&, DddtLocal&, chan&, buf0%(),
  updateCount&, DdtmCycleOn&)
' set the buffer for channel 1
ret& = VBdaqDacWaveSetBuffer&(handle&, DddtLocal&, chan&, buf1%(),
  updateCount&, DdtmCycleOn&)
' program the waveform parameters specific to dac channel 0
ret& = VBdaqDacWaveSetPredefWave&(handle&, DddtLocal&, 0, DdwtTriangle&,
  32768, 32768, 90, 0)
' program the waveform parameters specific to dac channel 1
ret& = VBdaqDacWaveSetPredefWave&(handle&, DddtLocal&, 1, DdwtSquare&,
  32768, 32768, 40, 0)
' buffer must be configured before the arm command is called. All
  channels
' will be armed.
ret& = VBdaqDacWaveArm(handle&, DddtLocal&)
```

Variable Rate, Variable Duty-Cycle Square-Wave Output

This section demonstrates the use of the counter/timer section of a DaqBook/100/200 or DaqBoard/100A/200A with the P3 port. After configuring the counter and setting the load and hold registers, the counter is armed. At this point, program execution continues while the counter outputs the signal. This example generates a variable rate, variable duty-cycle square wave. Functions used include:

- **`Vbdaq9513SetMasterMode&(handle&,deviceType&,whichDevice&,foutDiv&, cntSource&, comp1&, comp2&, tod&)`**
- **`Vbdaq9513SetCtrMode&(handle&,deviceType&,whichDevice&, ctrNum&,gayeCtrl&, cntEdge&, cntSource&, specGate&, reload&, cntRepeat&, cntType&, cntDir&, outputCtl&)`**
- **`Vbdaq9513SetHold&(handle&, deviceType&,whichDevice&, ctrNum&, ctrVal%)`**
- **`Vbdaq9513SetLoad&(handle&,deviceType&,whichDevice&, ctrNum&, ctrVal%)`**
- **`Vbdaq9513MultCtrl&(handle&,deviceType&,whichDevice&, ctrCmd&, ctr1&, ctr2&, ctr3&, ctr4&, ctr5&)`**



Initialize the 9513 master mode register fout divider: 10, fout source: DcsF2 (100 kHz), compare1: no, compare 2: no, time of day disabled. This will place a 10 kHz pulse on the oscillator output. The **`daq9513SetMasterMode`** function will initialize the counter/timer section and configure several of its parameters. This is a system-wide function which affects all 5 counter timers. **Note**: for a complete understanding of counter/timer operation, read the data book on the 9513 chip supplied by AMD. Aside from initializing the counter/timer section, this application does not use most of the capabilities of the **`daq9513SetMasterMode`** function. The first two arguments in this function select a clock source for the fout signal found on connector P3, then select a divider for that signal. F2 in this application is a fixed, internal frequency source of 100 kHz. Our example divides this fixed frequency by 10 yielding a signal on fout of 10 kHz.

```
ret& = VBdaq9513SetMasterMode&(handle&, DiodtLocal9513&, 0, 10, DcsF2&,
  0, 0,  DtodDisabled&)
```

The **`daq9513SetCtrMode`** function configures an individual counter in the 9513. The first argument specifies the counter to be configured; the second argument specifies the internal operation of the gate control. Our application does not use the gate, so it is disabled. The fixed 100 kHz internal clock (F1) is used as the source. By setting the reload parameter to 1, the counter will use the 'load' register and the 'hold' register to generate the pulse train. When the counter is armed, the 'load' register value is loaded then decremented on every edge of the F1 clock. The output signal will be high during this phase. When the terminal count is reached, the 'hold' register is loaded then decremented on every edge of the F1 clock. The output signal is low during this phase. If the reload argument is set to 0, only the 'load' register is used, always yielding a 50% duty-cycle pulse train. The **`cntRepeat`** argument specifies whether the pulse train should execute once or repeat continuously. The counter interprets the load and load register as either binary or BCD, depending on the value of the **`cntType`** argument. The **`cntDir`** specifies whether the internal counter should count up or down to reach the terminal count. A value of 5 counted down has the same effect as a value of 65,530 counted up.

```
ret& = VBdaq9513SetCtrMode&(handle&, DiodtLocal9513&, 0, 1, DgcNoGating&,
  1, DcsF1&, 0, 1, 1, 0, 0, DocTCToggled&)
```

Set the load register to 75 and the hold register to 25. This produces a high duty-cycle of 75% and (with 100 total counts to count down) a frequency of 10 kHz.

```
        ' Load the load register: 75 low counts & hold register with 25
  counts
ret& = VBdaq9513SetLoad&(handle&, DiodtLocal9513, 0, 1, 75)
ret& = VBdaq9513SetHold&(handle&, DiodtLocal9513, 0, 1, 25)
```

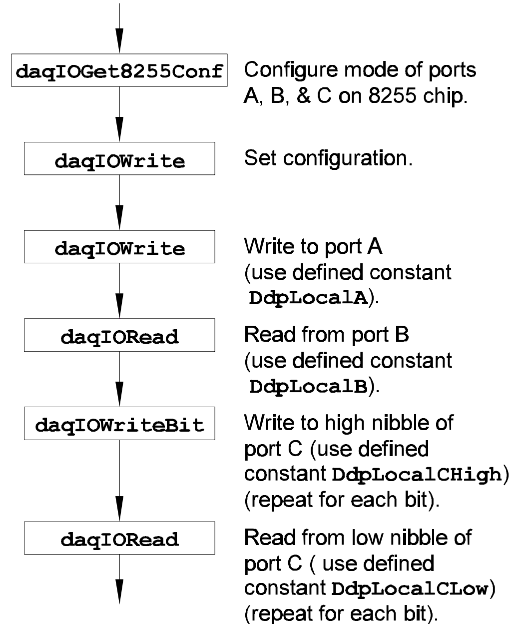
The **daq9513MultCtrl** function will arm counter 1.

```
ret& = VBdaq9513MultCtrl&(handle&, DiodtLocal9513&, 0, DmccLoadArm&, 1,
  0, 0, 0, 0)
```

Continue the pulse train until user terminates it.

```
Print "A 10Khz 25% duty cycle square wave is on the counter 1 output.":
  Print
MsgBox "Click to halt counter 1 output.", , "Counter 1"
    ' Halt all output
ret& = VBdaq9513MultCtrl&(handle&, DiodtLocal9513&, 0, DmccDisarm&, 1, 0,
  0, 0, 0)
ret& = VBdaq9513SetMasterMode&(handle&, DiodtLocal9513&, 0, 0, DcsF2&, 0,
  0, DtodDisabled&)
Print "Outputs disabled."
```

Digital I/O on P2



This program demonstrates the functions controlling digital I/O on connector P2 of the DaqBook/100/200 and DaqBoard/100A/200A. First, the 3 digital ports on the 8255 are configured as input, output, or both in the case of port C; then, appropriate I/O commands are issued. Functions used include:

- **`VBdaqIOReadBit&(handle&, devType&, devPort&, whichDevice&, whichExpPort&, bitNum&, bitValue&)`**
- **`VBdaqIORead&(handle&, devType&, devPort&, whichDevice&, whichExpPort&, value&)`**
- **`VBdaqIOWriteBit&(handle&, devType&, devPort&, whichDevice&, whichExpPort&, bitNum&, bitValue&)`**
- **`VBdaqIOWrite&(handle&, devType&, devPort&, whichDevice&, whichExpPort&, value&)`**
- **`VBdaqIOGet8255Conf&(handle&, portA&, portB&, portCHigh&, portCLow&, config&)`**

```
Dim config&, byteVal&, bitVal&, x%
Dim buf(10) As Byte, active&, retCount&
handle& = VBdaqOpen&("DaqBook0")
ret& = VBdaqSetErrorHandler&(handle&, 100)
On Error GoTo ErrorHandlerDIG1
ret& = VBdaqIOGet8255Conf&(handle&, 0, 1, 0, 1, config&)
```

The function **`daqIOGet8255Conf`** returns the appropriate configuration value to use in **`daqIOWrite`**. As shown above, the handle of the opened Daq* device is the first parameter passed. The second, third, fourth, and fifth parameters respectively indicate: the 8255 port A value, the port B value, the high-nibble value of port C, and the low-nibble value of port C. The values for the parameters passed in the call shown above will return the configuration value (port A = OUTPUT, port B = INPUT, port C / high nibble = output, port C / low nibble = INPUT) in the **`config&`** parameter, which matches the current configuration of the 8255.

The **`daqIOWrite`** function writes the obtained configuration value to the selected port.

```
ret& = VBdaqIOWrite&(handle&, DiodtLocal8255&, Diodp8255IR&, 0, 0,_
 config&)
```

Write hex 55 to port A on the Daq*'s base unit.

```
ret& = VBdaqIOWrite&(handle&, DiodtLocal8255&, Diodp8255A&, 0,  0,_
&H55)
```

Read port B and put the value into the variable byteVal%.

```
ret& = VBdaqIORead&(handle&, DiodtLocal8255&, Diodp8255B&, 0, 0,_
 byteVal&)
Print "The value on digital port B : &H"; Hex$(byteVal&): Print
```

The following lines write to the high nibble of port C.

```
ret& = VBdaqIOWriteBit&(handle&, DiodtLocal8255&, Diodp8255CHigh&, 0,_
0, 0, 1)
ret& = VBdaqIOWriteBit&(handle&, DiodtLocal8255&, Diodp8255CHigh&, 0,_
0, 1, 0)
ret& = VBdaqIOWriteBit&(handle&, DiodtLocal8255&, Diodp8255CHigh&, 0,_
0, 2, 1)
```

```
ret& = VBdaqIOWriteBit&(handle&, DiodtLocal8255&, Diodp8255CHigh&, 0,_
0, 3, 0)

Print "The high nibble of digital port C set to : 0101 (&H5)"  Print
```

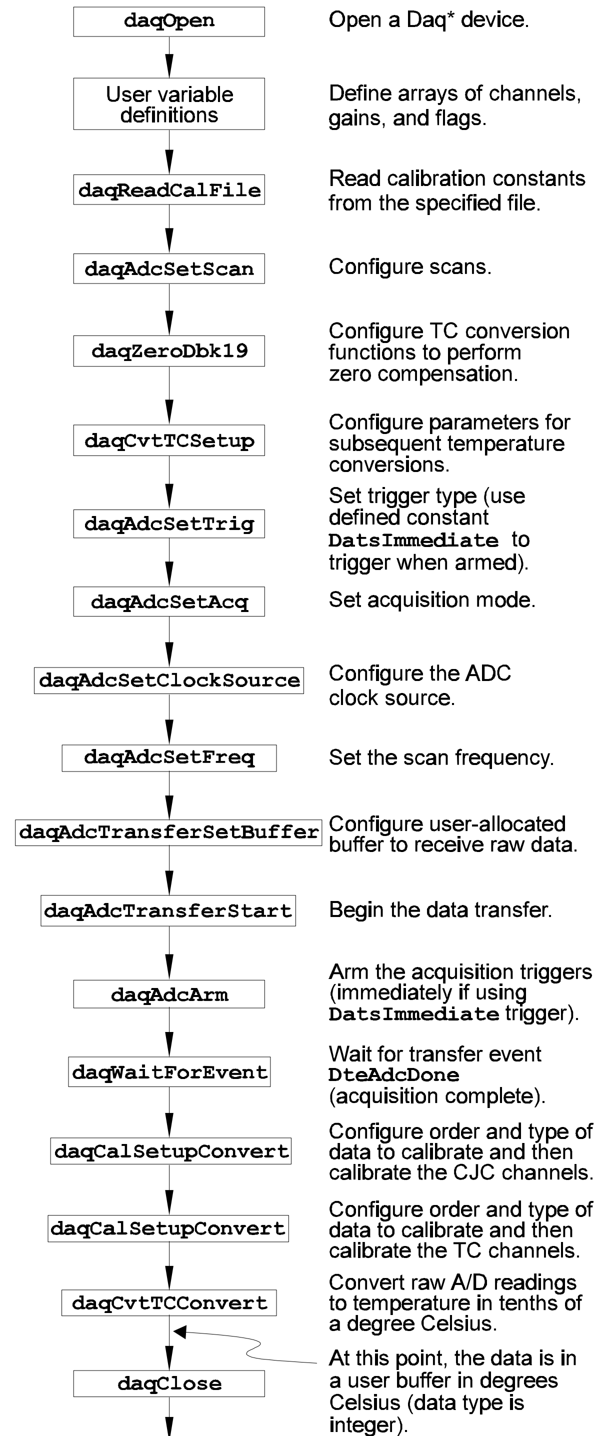
The next lines read the low nibble of port C on the base unit.

```
For x% = 0 To 3
  ret& = VBdaqIOReadBit&(handle&, DiodtLocal8255&,_
  Diodp8255CLow&, 0, 0, x%, bitVal&)
  Print "The value on bit "; x%; " of digital port C : &H";_
  Hex$(bitVal&)
Next x%
```

Temperature Measurements Using Single TC Type on a Single DBK19 Card

The 4 examples follow the same command sequence except for their arguments or program code for data output:

- Example 1 demonstrates repeated measurements of TC inputs.
- Example 2 demonstrates block averaging of the same TC inputs as example one. This example performs each reading 5 times and averages them together.
- Example 3 uses the same data as example 2; but rather than averaging the 5 scans, it outputs each of them to the screen.
- Example 4 gathers the same data as the previous examples but applies a moving average to that data.



DBK19 Example 1: Type J Thermocouples

In this example, we wish to repeatedly measure the temperatures sensed by 2 type J thermocouples attached to channels 18 and 19 through a DBK19 card. The DBK19 CJC signal is always the first signal on the card. The shorted channel (used for zero compensation) is always the second signal on the card. In this case, they are on channels 16 and 17. First we list the configuration (see table).

Card	Channel	Channel Type
DBK19	16 17 18 19	CJC Shorted (zero) Type J Type J
Local	0 1-15	Used for DBK19 Free for other uses

Now we must specify the scan, the sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the temperature channels; the scan must first include the CJC zero, thermocouple zero and CJC, and then the temperature channels (see table).

Scan Position	Channel Type	Channel
0	CJC Zero	17
1	Type J Zero	17
2	CJC	16
3	Type J	18
4	Type J	19

The thermocouples need not be scanned in any particular order. We might have specified channel 18 before channel 17, but keeping things in order will make the calibration easier.

For each scan position, we must specify the PGA gain code. Assuming the Daq* is configured for bipolar operation (to allow measurement of temperatures below the temperature at the DBK19 card), we choose the gain codes from the table and add them to the scan description.

Scan Position	Channel Type	Channel	Gain Code
0	CJC Zero	17	Dbk19BiCJC
1	Type J Zero	17	Dbk19BiTypeJ
2	CJC	16	Dbk19BiCJC
3	Type J	18	Dbk19BiTypeJ
4	Type J	19	Dbk19BiTypeJ

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input					
	Readings				
Measurement	0	1	2	3	4
1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	CJC Zero	Type J Zero	CJC	Type J	Type J
3	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...
10	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

Now we can configure the Daq* with this information:

```
Public Sub MeasureTC()
Const ScanLength& = 5   'Total channels per scan
Const ScanCount& = 10   'Number of scans to be acquired
Const TCcount& = 2      'Number of thermocouples per scan
Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim temp%(TCcount * ScanCount), ret&, daqAlias$
Dim daqHandle&, i&

'Open the device.
daqHandle& = VBdaqOpen&("daqbook0")

' Read calibration file
ret& = VBdaqReadCalFile&(daqHandle&, "daqbook.cal")

' Set arrays of channels, gains, and flags
' Grounded channel with CJC gain
chan&(0) = 17: gain&(0) = Dbk19BiCJC&: flag&(0) = DafBipolar&

' Grounded channel with TC gain
chan&(1) = 17: gain&(1) = Dbk19BiTypeJ&: flag&(1) = DafBipolar&

' CJC channel
chan&(2) = 16: gain&(2) = Dbk19BiCJC&: flag&(2) = DafBipolar&

'TC channel
chan&(3) = 18: gain&(3) = Dbk19BiTypeJ&: flag&(3) = DafBipolar&

'TC channel
chan&(4) = 19: gain&(4) = Dbk19BiTypeJ&: flag&(4) = DafBipolar&

' Load scan sequence FIFO.
```

```
ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
  ScanLength)

' Configure the TC convert functions for zero compensation.
ret& = VBdaqZeroDbk19&(1)

' Configure temperature conversion without averaging.
ret& = VBdaqCvtTCSetup&(ScanLength, 2, TCcount, Dbk19TCTypeJ&, 1, 1)

' Configure the trigger for an immediate trigger.
ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)

' Set the acquisition mode.
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)

' Set the Adc clock source.
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)

' Set the scan frequency to 100 Hz.
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)

' Configure user-allocated buffer for ADC data collection.
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
  DatmCycleOff& + DatmUpdateBlock&)

' Setup and start the data transfer.
ret& = VBdaqAdcTransferStart&(daqHandle&)

' Arm the acquisition. Also triggers if immediate trigger set.
ret& = VBdaqAdcArm&(daqHandle&)

' Wait until the transfer is complete.
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone&)

' Calibrate CJC: 1 chan. starting at position 2 for 10 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 2, 1, DcalTypeCJC&,
  Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)

' Calibrate TCs: 2 chans. starting at position 3 for 10 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 3, 2,
  DcalTypeDefault&, Dbk19BiTypeJ&, 18, 1, 1, buf%(), ScanCount)

' Convert 'scans' scans of counts to two temperatures.
ret& = VBdaqCvtTCConvert&(buf%(), ScanCount, temp%(), TCcount *
  ScanCount)

' Print the temperatures for 10 scans on the current form.
For i = 0 To 9
    Print "Channel 18: "; 0.1 * temp%(i * TCcount); "   Channel 19: ";
  0.1 * temp%(i * TCcount + 1)
Next i

'Close the device.
ret& = VBdaqClose&(daqHandle&)
End Sub
```

DBK19 Example 2: Block-Averaged TC readings

In this example, we want to acquire the same information as in example 1; however, we wish to use the Daq*'s speed to reduce noise by taking each reading 5 times and averaging them together.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

Assuming we are using the same thermocouples connected in the same way, the scan configuration is like example 1:

```
Public Sub BlockAvgTc()
' Block averages scans of Type J thermocouple readings
Const ScanLength& = 5   'Total channels per scan
Const ScanCount& = 5   'Number of scans acquired per acquisition.
Const TCcount& = 2      'Number of thermocouples per scan

Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim temp%(TCcount), ret&, daqAlias$
Dim daqHandle&, i&

'Open the device.
daqHandle& = VBdaqOpen&("daqbook0")

' Read calibration file
ret& = VBdaqReadCalFile&(daqHandle&, "daqbook.cal")

' Set arrays of channels, gains, and flags
' Grounded channel with CJC gain
chan&(0) = 17: gain&(0) = Dbk19BiCJC&: flag&(0) = DafBipolar&

' Grounded channel with TC gain
chan&(1) = 17: gain&(1) = Dbk19BiTypeJ&: flag&(1) = DafBipolar&

' CJC channel
chan&(2) = 16: gain&(2) = Dbk19BiCJC: flag&(2) = DafBipolar&

'TC channel
chan&(3) = 18: gain&(3) = Dbk19BiTypeJ&: flag&(3) = DafBipolar&

'TC channel
chan&(4) = 19: gain&(4) = Dbk19BiTypeJ&: flag&(4) = DafBipolar&

' Load scan sequence FIFO.
```

```
ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
 ScanLength)

' Configure the TC convert functions for zero compensation.
ret& = VBdaqZeroDbk19&(1)

' Configure temperature conversion with block averaging.
ret& = VBdaqCvtTCSetup&(ScanLength, 2, TCcount, Dbk19TCTypeJ&, 1, 0)

' Configure the trigger for an immediate trigger.
ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)

' Set the acquisition mode.
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)

' Set the Adc clock source.
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)

' Set the scan frequency to 100 Hz.
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)

' Configure user-allocated buffer for ADC data collection.
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
 DatmCycleOff& + DatmUpdateSingle&)

' Setup and start the data transfer.
ret& = VBdaqAdcTransferStart&(daqHandle&)

' Arm the acquisition. Also triggers if immediate trigger set.
ret& = VBdaqAdcArm(daqHandle&)

'Acquire 10 groups of 5 scans. Average each group and convert to temp.
For i = 1 To 10

' Wait until the transfer is complete.
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone&)

' Calibrate CJC: 1 chan. starting at position 2 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 2, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)

' Calibrate TCs: 2 chans. starting at position 3 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 3, 2,
 DcalTypeDefault&, Dbk19BiTypeJ&, 18, 1, 1, buf%(), ScanCount)

' Convert 'scans' scans of counts to two temperatures
ret& = VBdaqCvtTCConvert&(buf%(), ScanCount, temp%(), TCcount)

'Display the averaged temperatures
Print "Channel 18: "; 0.1 * temp%(0); "   Channel 19: "; 0.1 * temp%(1)

' Start transfer & rearm for the next group of 5 scans
ret& = VBdaqAdcTransferStart(daqHandle&)
ret& = VBdaqAdcArm(daqHandle&)

Next i

' Close the device
ret& = VBdaqClose&(daqHandle&)

End Sub
```

DBK19 Example 3: Multiple Sequential Measurement

In this example, we wish to collect the same data as in example 2; but instead of averaging the groups of 10 consecutive scans, we want to convert each scan's measurements into individual temperature values.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert			
		Results	
Measurement	Scan	0	1
1	1	Temp °C	Temp °C
1	2	Temp °C	Temp °C
1	3	Temp °C	Temp °C
1	4	Temp °C	Temp °C
1	5	Temp °C	Temp °C
2	1	Temp °C	Temp °C
2	2	Temp °C	Temp °C
...	...	...	...
10	5	Temp °C	Temp °C

The scan setup is the same as in examples 1 and 2. We again configure for the conversion to temperatures, this time (as in example 1) specifying no averaging (for brevity, some comments have been removed from the code):

```
Public Sub SequentialTC()
Const ScanLength& = 5   'Total channels per scan
Const ScanCount& = 5   'Number of scans to be acquired
Const TCcount& = 2      'Number of thermocouples per scan

Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim temp%(TCcount * ScanCount), ret&, daqAlias$
Dim daqHandle&, i&, j&

` Open the DaqBook
daqHandle& = VBdaqOpen&("daqbook0")

ret& = VBdaqReadCalFile&(daqHandle&, "daqbook.cal")

chan&(0) = 17: gain&(0) = Dbk19BiCJC&: flag&(0) = DafBipolar&
chan&(1) = 17: gain&(1) = Dbk19BiTypeJ&: flag&(1) = DafBipolar&
chan&(2) = 16: gain&(2) = Dbk19BiCJC&: flag&(2) = DafBipolar&
chan&(3) = 18: gain&(3) = Dbk19BiTypeJ&: flag&(3) = DafBipolar&
chan&(4) = 19: gain&(4) = Dbk19BiTypeJ&: flag&(4) = DafBipolar&
ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
 ScanLength)
ret& = VBdaqZeroDbk19&(1)

' Configure temperature conversion without averaging.
ret& = VBdaqCvtTCSetup&(ScanLength, 2, TCcount, Dbk19TCTypeJ&, 1, 1)

ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)
```

```
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
 DatmCycleOff& + DatmUpdateSingle&)

ret& = VBdaqAdcTransferStart&(daqHandle&)
ret& = VBdaqAdcArm&(daqHandle&)

'Acquire 10 groups of 5 scans and convert to temperature.
For i = 1 To 10
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone)

ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 2, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 3, 2,
 DcalTypeDefault&, Dbk19BiTypeJ&, 18, 1, 1, buf%(), ScanCount)
ret& = VBdaqCvtTCConvert&(buf%(), ScanCount, temp%(), TCcount *
 ScanCount)

'Display the temperatures
For j = 0 To 4
    Print "Channel 18: "; 0.1 * temp%(TCcount * j); "   Channel 19: ";
 0.1 * temp%(TCcount * j + 1)
Next j

'Rearm the device for the next group of 10 scans
ret& = VBdaqAdcArm&(daqHandle&)
ret& = VBdaqAdcTransferStart&(daqHandle&)
Next i

' Close the device
ret& = VBdaqClose&(daqHandle&)
End Sub
```

DBK19 Example 4: Moving Averaged Measurements

In this example, we wish to collect the same data as in example 3; but to reduce noise, we will use a moving average to average consecutive triplets of scans.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

The scan setup is the same as in the previous examples. Some comments have been omitted here for brevity. We again configure for the conversion to temperatures, this time (as in example 1) specifying moving averaging of 3 scans.

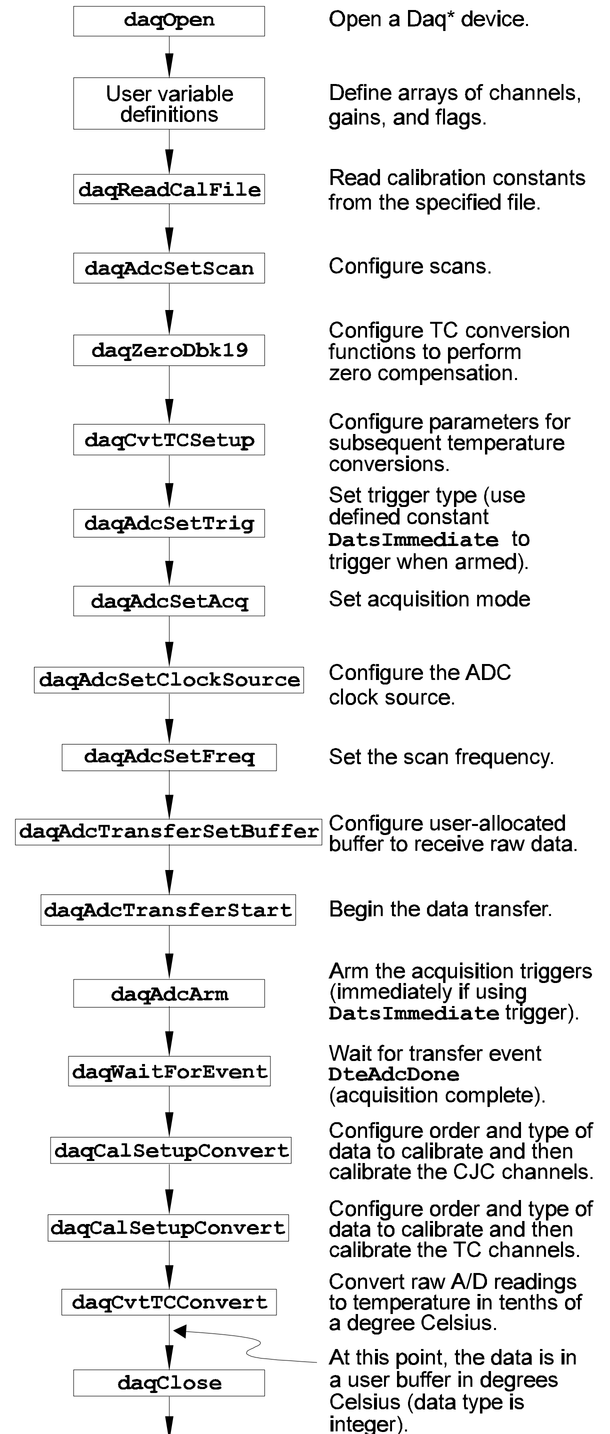
```
Public Sub MovingAvgTC()
' Applies a three-sample moving average to
' blocks of Type J thermocouple readings
Const ScanLength& = 5   'Total channels per scan
Const ScanCount& = 5   'Number of scans to be acquired
Const TCcount& = 2      'Number of thermocouples per scan
Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim temp%(TCcount * ScanCount), ret&, daqAlias$
Dim daqHandle&, i&, j&

daqHandle& = VBdaqOpen&("daqbook0")
ret& = VBdaqReadCalFile&(daqHandle&, "daqbook.cal")

chan&(0) = 17: gain&(0) = Dbk19BiCJC&: flag&(0) = DafBipolar&
chan&(1) = 17: gain&(1) = Dbk19BiTypeJ&: flag&(1) = DafBipolar&
chan&(2) = 16: gain&(2) = Dbk19BiCJC&: flag&(2) = DafBipolar&
chan&(3) = 18: gain&(3) = Dbk19BiTypeJ&: flag&(3) = DafBipolar&
chan&(4) = 19: gain&(4) = Dbk19BiTypeJ&: flag&(4) = DafBipolar&
ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
 ScanLength)
ret& = VBdaqZeroDbk19&(1)

' Configure TC conversion with 3-sample moving average.
ret& = VBdaqCvtTCSetup&(ScanLength, 2, TCcount, Dbk19TCTypeJ&, 1, 3)

ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
 DatmCycleOff& + DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart&(daqHandle&)
ret& = VBdaqAdcArm&(daqHandle&)

'Acquire 10 groups of 5 scans and convert to temperature
For i = 1 To 10
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone&)

ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 2, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 3, 2,
 DcalTypeDefault&, Dbk19BiTypeJ&, 18, 1, 1, buf%(), ScanCount)
ret& = VBdaqCvtTCConvert&(buf%(), ScanCount, temp%(), TCcount *
 ScanCount)

'Display the temperatures
For j = 0 To 4
    Print "Channel 18: "; 0.1 * temp%(TCcount * j); "   Channel 19: ";
 0.1 * temp%(TCcount * j + 1)
Next j

'Rearm the device for the next group of 10 scans
ret& = VBdaqAdcArm&(daqHandle&)
ret& = VBdaqAdcTransferStart(daqHandle&)
Next i

' Close the device
ret& = VBdaqClose&(daqHandle&)
End Sub
```

Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards

This program demonstrates temperature acquisitions using multiple TC types and multiple DBK19 cards. The two commands **daqTCSetup** and **daqTCConvert** have been combined into the one **daqTCSetupConvert** command. The sequence of the last 3 blocks on the flow chart must be used multiple times, once for each card (also, if there are multiple TC types on a card, once for each TC type on that card).

In this example, we wish to repeatedly measure the temperatures sensed by 2 Type J and 2 Type K thermocouples attached through 1 DBK19 card and 2 more Type J thermocouples attached through another DBK19. The DBK19 CJC signal is always the first signal on the card, and the shorted channel (used for zero compensation) is always the second channel on the card. First we list the configuration:

Now we must specify the scan, the sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the temperature channels; and so, the scan must first include the CJC and then immediately the temperature channels (see table).



Card	Channel	Channel Type
DBK19	16	CJC
	17	Shorted (zero)
	18	Type J
	19	Type J
	20	Type K
	21	Type K
DBK19	32	CJC
	33	Shorted (zero)
	34	Type J
	35	Type J
Local	0-1	Used for DBK19
	2-15	Free for other uses

The thermocouples are separated in the scan by type. The readings from each type are consecutive and immediately preceded by their CJC Zero, thermocouple zero, and CJC readings for calculation reference. It is not appropriate to consolidate the 4 Type J thermocouples because they are connected through 2 different DBK19s. Each DBK19 has its own CJC and offset errors as a reference for thermocouples attached to that DBK19.

For each scan position, we must specify the PGA gain. Assuming the Daq* is configured for bipolar operation (to allow measurement of temperatures below the temperature at the DBK19 cards), we choose the gain codes from the table and add them to the scan description.

Scan Position	Channel Type	Channel	Gain Code
0	CJC Zero	17	Dbk19BiCJC
1	Type J Zero	17	Dbk19BiTypeJ
2	CJC	16	Dbk19BiCJC
3	Type J	18	Dbk19BiTypeJ
4	Type J	19	Dbk19BiTypeJ
5	CJC Zero	17	Dbk19BiCJC
6	Type K	17	Dbk19BiTypeK
7	ZeroCJC	16	Dbk19BiCJC
8	Type K	20	Dbk19BiTypeK
9	Type K	21	Dbk19BiTypeK
10	CJC Zero	33	Dbk19BiCJC
11	Type J Zero	33	Dbk19BiTypeJ
12	CJC	32	Dbk19BiCJC
13	Type J	34	Dbk19BiTypeJ
14	Type J	35	Dbk19BiTypeJ

The following tables show the raw data input and the resulting temperature data output for this sample program.**Raw Data Input**

Measure-ment	Scan	Readings 0	1	2	3	4	5	6	7	8	9	10	(1-3)	14
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J

Results After daqTCConvert

Measurement	Results 0	1	2	3	4	5
1	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C
2	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C
...	...	...	...	...	...	...
10	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C

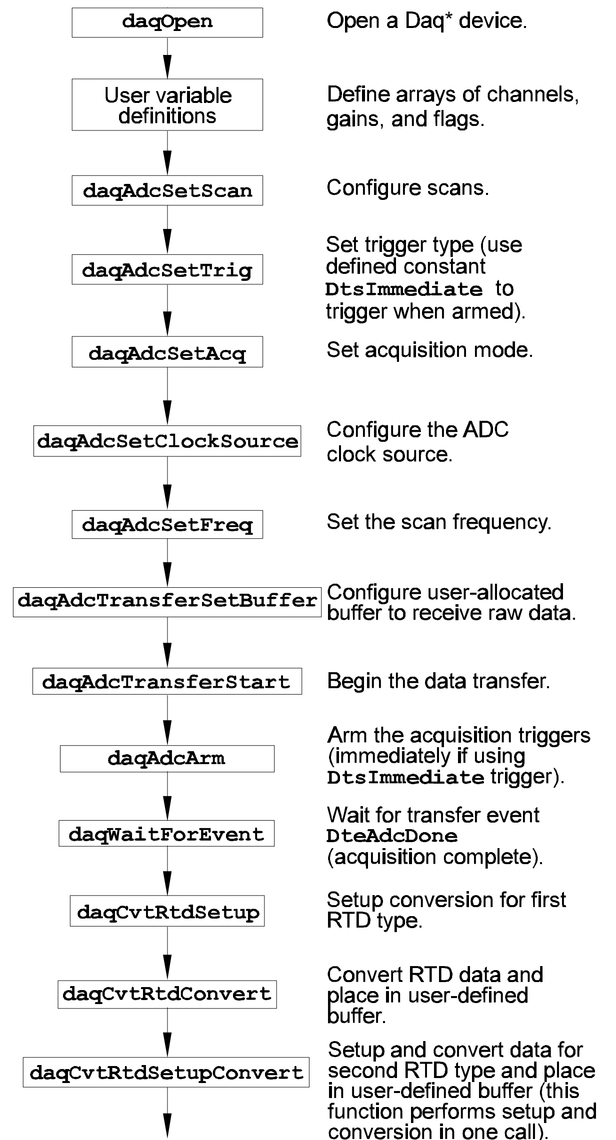
Now we can configure the DaqBook/DaqBoard with this information:

```
Public Sub MultDbk19()
Const ScanLength& = 15 'Total channels per scan
Const ScanCount& = 5   'Number of scans to be acquired
Const TCcount1& = 2    'Number of Type J TCs on first Dbk19
Const TCcount2& = 2    'Number of Type K TCs on first Dbk19
Const TCcount3& = 2    'Number of Type J TCs on second Dbk19
Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim temp1%(TCcount1), temp2%(TCcount2)
Dim temp3%(TCcount3)

Dim daqHandle&, i&, j&, ret&

daqHandle& = VBdaqOpen&("daqbook0")
```

```
ret& = VBdaqReadCalFile&(daqHandle&, "daqbook.cal")

chan&(0) = 17: gain&(0) = Dbk19BiCJC&: flag&(0) = DafBipolar&
chan&(1) = 17: gain&(1) = Dbk19BiTypeJ&: flag&(1) = DafBipolar&
chan&(2) = 16: gain&(2) = Dbk19BiCJC&: flag&(2) = DafBipolar&
chan&(3) = 18: gain&(3) = Dbk19BiTypeJ&: flag&(3) = DafBipolar&
chan&(4) = 19: gain&(4) = Dbk19BiTypeJ&: flag&(4) = DafBipolar&

chan&(5) = 17: gain&(5) = Dbk19BiCJC&: flag&(5) = DafBipolar&
chan&(6) = 17: gain&(6) = Dbk19BiTypeK&: flag&(6) = DafBipolar&
chan&(7) = 16: gain&(7) = Dbk19BiCJC&: flag&(7) = DafBipolar&
chan&(8) = 20: gain&(8) = Dbk19BiTypeK&: flag&(8) = DafBipolar&
chan&(9) = 21: gain&(9) = Dbk19BiTypeK&: flag&(9) = DafBipolar&

chan&(10) = 33: gain&(10) = Dbk19BiCJC&: flag&(10) = DafBipolar&
chan&(11) = 33: gain&(11) = Dbk19BiTypeJ&: flag&(11) = DafBipolar&
chan&(12) = 32: gain&(12) = Dbk19BiCJC&: flag&(12) = DafBipolar&
chan&(13) = 34: gain&(13) = Dbk19BiTypeJ&: flag&(13) = DafBipolar&
chan&(14) = 35: gain&(14) = Dbk19BiTypeJ&: flag&(14) = DafBipolar&

ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
 ScanLength)
ret& = VBdaqZeroDbk19&(1)

ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
 DatmCycleOff& + DatmUpdateSingle&)

ret& = VBdaqAdcTransferStart&(daqHandle&)
ret& = VBdaqAdcArm(daqHandle&)
ret& = VBdaqWaitForEvent(daqHandle&, DteAdcDone&)

For i = 1 To 10

' Calibrate CJC: 1 chan. starting at position 2 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 2, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)

' Calibrate type J TCs: 2 chans. starting at position 3 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 3, 2,
 DcalTypeDefault&, Dbk19BiTypeJ&, 18, 1, 1, buf%(), ScanCount)

' Calibrate CJC: 1 chan. starting at position 7 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 7, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 16, 1, 1, buf%(), ScanCount)

' Calibrate type K TCs: 2 chans. starting at position 3 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 8, 2,
 DcalTypeDefault&, Dbk19BiTypeK&, 20, 1, 1, buf%(), ScanCount)

' Calibrate CJC: 1 chan. starting at position 12 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 12, 1, DcalTypeCJC&,
 Dbk19BiCJC&, 32, 1, 1, buf%(), ScanCount)

' Calibrate type J TCs: 2 chans. starting at position 13 for 5 scans.
ret& = VBdaqCalSetupConvert&(daqHandle&, ScanLength, 13, 2,
 DcalTypeDefault&, Dbk19BiTypeJ&, 34, 1, 1, buf%(), ScanCount)

' Convert the first type J TC readings to temperatures
ret& = VBdaqCvtTCSetupConvert&(ScanLength, 2, TCcount1, Dbk19TCTypeJ&, 1,
 0, buf%(), ScanCount, temp1%(), TCcount1)
Print "Channel 18: "; 0.1 * temp1(0); "  Channel 19: "; 0.1 * temp1(1)

' Convert the first type K TC readings to temperatures
```

```
ret& = VBdaqCvtTCSetupConvert&(ScanLength, 7, TCcount2, Dbk19TCTypeK&, 1,
 0, buf%(), ScanCount, temp2%(), TCcount2)
Print "Channel 20: "; 0.1 * temp2(0); "  Channel 21: "; 0.1 * temp2(1)

' Convert the first type J TC readings to temperatures
ret& = VBdaqCvtTCSetupConvert&(ScanLength, 12, TCcount3, Dbk19TCTypeJ&,
 1, 0, buf%(), ScanCount, temp3%(), TCcount3)
Print "Channel 34: "; 0.1 * temp3(0); "  Channel 35: "; 0.1 * temp3(1)

ret& = VBdaqAdcArm(daqHandle&)
ret& = VBdaqWaitForEvent(daqHandle&, DteAdcDone&)
Next i

End Sub
```

Temperature Measurements Using Multiple RTDs on a Single DBK9 Card



This program demonstrates temperature acquisitions using multiple RTD types and a single DBK9 card. After this program configures and arms the DBK card, it begins acquiring data. At this point, program execution is suspended until all the data is gathered. The program demonstrates the conversion of data as both a two-step process and a single-step process. Note the conversion routines need to be called for each type of RTD in the scan. The temperature at the RTD is derived from 4 voltage values.

In this example, we wish to acquire some temperature readings from 3 RTDs. There are two 100-ohm RTDs attached to channels 16 and 17 of the DBK9 and one 1000-ohm RTD attached to channel 18. The configuration looks like this:

Card	Channel	Channel Type
DBK9	16 17 18	100 ohm RTD 100 ohm RTD 1000 ohm RTD
Local	0 1-15	Used for DBK9 Free for other uses

First we must specify the scan sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the RTD channels. The scan must include the 4 voltage readings in the correct order for each channel (see table).

Scan Position	Channel Number	*Channel Gain
0	16	Dbk9VoltageA
1	16	Dbk9VoltageB
2	16	Dbk9VoltageD
3	16	Dbk9VoltageD
4	17	Dbk9VoltageA
5	17	Dbk9VoltageB
6	17	Dbk9VoltageD
7	17	Dbk9VoltageD
8	18	Dbk9VoltageA
9	18	Dbk9VoltageB
10	18	Dbk9VoltageD
11	18	Dbk9VoltageD

* These are not actual gains. They are used to select voltages A-D for each RTD channel.

Note that the RTDs need not be scanned in any particular order, but the 4 readings for each RTD must be placed in the scan sequentially. We might have specified channel 17 before channel 16. It is best to group all the RTD reading groups of the same value together because this makes using the temperature conversion functions easier.

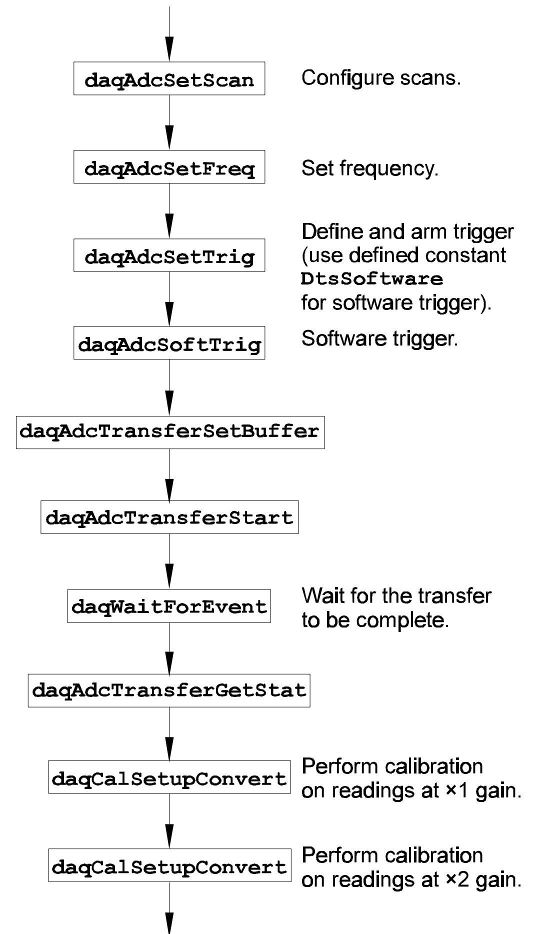
Now we can configure the Daq* with this information. First we will define some constants that will make the program easier to modify.

```
Public Sub MeasureRTD()
Const RdsPerRTD = 4
Const nRTDs = 3
Const FirstRTDChanNo = 16
Const Nscans = 10
Const vaOffset = 0
Const vbOffset = 1
```

```
Const vcOffset = 2
Const vdOffset = 3
Const ReadingsPerScan = nRTDs * RdsPerRTD
Const bufSize = Nscans * ReadingsPerScan

Dim chan&(ReadingsPerScan), gain&(ReadingsPerScan),
 flag&(ReadingsPerScan)
Dim buf%(bufSize), temp1%(Nscans% * 2), temp2%(Nscans), i%, j%, ret&
Dim daqHandle&, tmpTemperature!

daqHandle& = VBdaqOpen("daqbook0")

' Set arrays of channels, gains, and flags.
For i = 0 To nRTDs - 1
    For j = 0 To RdsPerRTD
        chan(i * RdsPerRTD + j) = i + FirstRTDChanNo
    Next j
gain(i * RdsPerRTD + vaOffset) = Dbk9VoltageA&
gain(i * RdsPerRTD + vbOffset) = Dbk9VoltageB&
gain(i * RdsPerRTD + vcOffset) = Dbk9VoltageC&
gain(i * RdsPerRTD + vdOffset) = Dbk9VoltageD&

flag(i * RdsPerRTD + vaOffset) = DafBipolar&
flag(i * RdsPerRTD + vbOffset) = DafBipolar&
flag(i * RdsPerRTD + vcOffset) = DafBipolar&
flag(i * RdsPerRTD + vdOffset) = DafBipolar&
Next i

ret& = VBdaqAdcSetScan(daqHandle&, chan(), gain(), flag(),
 ReadingsPerScan)
ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot, 0, Nscans)
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), Nscans,
 DatmCycleOff& + DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart&(daqHandle&)
ret& = VBdaqAdcArm(daqHandle&)
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone&)

' Setup conversion for first 2 RTD's starting at position 0.
ret& = VBdaqCvtRtdSetup&(ReadingsPerScan, 0, 2, Dbk9RtdType100&, 1)

' Convert the data for the first 2 RTD's
ret& = VBdaqCvtRtdConvert&(buf%(), Nscans, temp1%(), Nscans * 2)

' Setup and convert the data for the 1000 ohm RTD in one step
ret& = VBdaqCvtRtdSetupConvert&(ReadingsPerScan, 8, 1, Dbk9RtdType1K&, 1,
 buf%(), Nscans, temp2%(), Nscans)

' Display the temperatures for the RTD's
For i = 0 To Nscans - 1
    Print "Scan: "; i; ":   ";
    ' Display the 100 Ohm RTD temperatures
    For j = 0 To 1
        tmpTemperature! = temp1%(i * 2 + j) / 10
        Print tmpTemperature; "   ";
    Next j
    'Display the 1000 Ohm RTD temperature
    tmpTemperature! = temp2%(i) / 10
    Print tmpTemperature
Next i
ret& = VBdaqClose&(daqHandle&)
End Sub
```

Using DBK Card Calibration Files

Software calibration functions are designed to adjust Daq* readings to compensate for gain and offset errors. Calibration constants are calculated at the factory by measuring the gain and offset errors of a card at each programmable gain setting. These constants are stored in a calibration text file which can be read by a program at runtime. This allows new boards to be configured for calibration by updating this calibration file rather than recompiling the program. Calibration constants and instructions are shipped with the related DBK boards. Programs like DaqView support this calibration and use the same constants.



The calibration operation removes static gain and offset errors that are inherent in the hardware. The calibration constants are measured at the factory and do not change during the execution of a program. These constants are different for each card and programmable-gain setting; they may even be different for each channel, depending on the design of the expansion card. **Note**: the DBK19 is shipped with calibration constants. Other cards use on-board potentiometers to perform hardware calibration.

The calibration process has 3 steps:

- **Initialization** consists of reading the calibration file.
- **Setup** describes the characteristics of the data to be calibrated.
- **Conversion** does the actual calibration of the data.

Function prototypes, return error codes, and parameter definitions are located in the DAQX.H header file for C (or similar files for other languages).

Cards that support the calibration functions are shipped with a diskette containing a calibration constants file. The name of the file will be the serial number of the card shipped with it. This file holds the calibration constants for each programmable-gain setting of that card. These constants should be copied to a calibration text file (DAQBOOK.CAL) located in the same directory as the program performing the calibration.

To set up the calibration file, perform the following steps:

1. Locate the diskette containing the calibration constants file.
2. Configure the card according to the hardware configuration section of the DBK chapter.
3. Edit the calibration file, DAQBOOK.CAL, using a text editor.
4. Add the card number information within brackets, as listed in the calibration file.
5. Add the calibration constants immediately after the card number. (These should be entered in the order given in the calibration file.)
6. Repeat steps 4 and 5 for each card.
7. Verify that no two cards are configured with the same card/channel number.

The table shows an example of a calibration file for configuring the main Daq* unit and two DBK19 cards connected to Daq* expansion channels 3 and 5.

```
[MAIN]
32760,32769
32801,32750
32740,32777
32810,32768

[EXP3]
32780,32779
32800,32756
32768,32780
32750,32742

[EXP5]
32752,32764
32783,32757
32749,32767
32777,32730
```

The initialization function for reading-in the calibration constants from the calibration text file is daqReadCalFile. The C language version of daqReadCalFile is similar to other languages and works as follows:

The filename with optional path information of the calibration file. If calfile is NULL or empty (“”), the default calibration file DAQBOOK.CAL will be read. This function is usually called once at the beginning of a program and will read all the calibration constants from the specified file. If calibration constants for a specific channel number and gain setting are not contained in the file, ideal calibration constants will be used (essentially not calibrating that channel). If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and a non-zero error code will be returned by the **daqReadCalFile** function.

Channel	Channel Type
0	Voltage1 @ X1 gain
1	Voltage2 @ X2 gain
2	Voltage3 @ X2 gain
3	Voltage4 @ X2 gain

Once the calibration constants have been read from the cal file, they can be used by the **daqCalSetup** and **daqCalConvert** functions. The **daqCalSetup** function will configure the order and type of data to be calibrated. This function requires data to be from consecutive channels configured for the same gain, polarity, and channel type. The calibration can be configured to use only the gain calibration constant and not the offset constant. This allows the offset to be removed at runtime using the zero compensation functions described later in this section.

Scan Position	Channel Type	Channel	Gain Code
0	Voltage1 @ X1 gain	0	**DgainX1**
1	Voltage2 @ X2 gain	1	**DgainX2**
2	Voltage3 @ X2 gain	2	**DgainX2**
3	Voltage4 @ X2 gain	3	**DgainX2**

In this example, several Daq* channels will be read and calibrated. This example assumes the calibration file has been created according to the initializing calibration constants section of this chapter. Expansion cards can perform the same type of calibration if the calibration constants are available for the card and a specified channel number. First list the configuration:

Now specify the scan (the sequence of channel numbers and gains that are to be gathered as one burst of readings). In this example, all the channels at each gain will be read together (in consecutive order) to make the calibration easier.

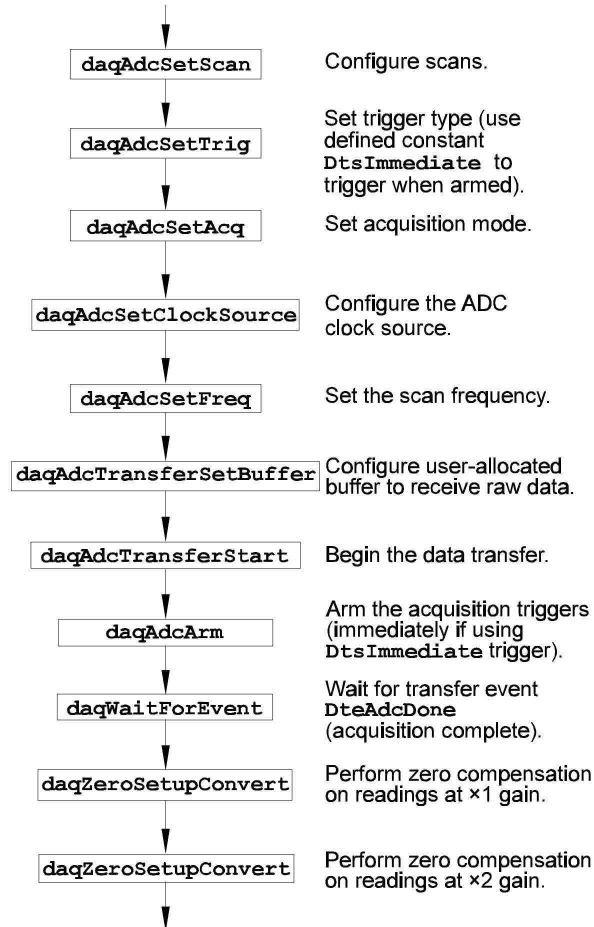
Now configure the Daq* with this information, and read 5 scans of data:

```
Dim chans&(4), gains&(4), buf%(20)

handle& = VBdaqOpen&("DaqBook0")

' Set array of channels and gains
chans&(0) = 0
gains&(0) = DgainX1&
chans&(1) = 1
gains&(1) = DgainX2&
chans&(2) = 2
gains&(2) = DgainX2&
chans&(3) = 3
gains&(3) = DgainX2&

' Load scan sequence FIFO :
ret& = VBdaqAdcSetScan&(handle&, chans&(), gains&(), 4)

' Set Clock
ret& = VBdaqAdcFreq&(handle&, 10)

' Define and arm trigger :
ret& = VBdaqAdcSetTrig&(handle&, DtsSoftware&, 0, 0, 0, 0)

' Trigger
ret& = VBdaqAdcSoftTrig&(handle&)

' Read the data
` 5 indicates the number of scans
```

```
' single mode for scans less than 500
ret& = VBdaqAdcTransferSetBuffer&(handle&, buf%(), 5, DatmCycleOff& +
 DatmSingleMode&)

ret& = VBdaqAdcTransferStart&(handle&)

'specifies to wait for the transfer to be complete
ret& = VBdaqWaitForEvent&(handle&, DteAdcDone&)

ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)

' Print the first scan of unconverted data
PRINT "Before Calibration:"
PRINT "Channel 0 at x1 gain: "; buf%(0)
PRINT "Channel 1 at x2 gain: "; buf%(1)
PRINT "Channel 2 at x2 gain: "; buf%(2)
PRINT "Channel 3 at x2 gain: "; buf%(3)

'Perform zero compensation on readings sampled at x1 gain
ret& = VBdaqCalSetupConvert&(handle&, 4, 0, 1, 0, DgainX1&, 0, 1, 0,
 buf%(), 5)

'Perform zero compensation on readings sampled at x2 gain
ret& = VBdaqCalSetupConvert&(handle&, 4, 1, 3, 0, DgainX2&, 1, 1, 0,
 buf%(), 5)

' Print the first scan of converted data
PRINT "After Calibration:"
PRINT "Channel 0 at x1 gain: "; buf%(0)
PRINT "Channel 1 at x2 gain: "; buf%(1)
PRINT "Channel 2 at x2 gain: "; buf%(2)
PRINT "Channel 3 at x2 gain: "; buf%(3)
```

Zero Compensation

Zero compensation removes offset errors while a program is running. This is useful in systems where the offset of a channel may change due to temperature changes, long-term drift, or hardware calibration changes. Reading a shorted channel on the same card at the same gain as the desired channel removes the offset at run-time.



Note: Zero compensation is not available for all expansion cards. The DBK19 has channel 1 permanently shorted for zero compensation; other cards require a channel to be shorted manually.

The zero-compensation functions require a shorted channel and a number of other channels to be sampled from the same card at the same gain as the shorted channel. These functions will work with cards (such as the DBK12, DBK13, and DBK19) that have one analog path from the input to the A/D converter. Other cards do not support the zero compensation functions because they have offset errors unique to each channel. The DBK19 is designed with channel 1 already shorted for performing zero compensation.

The **daqZeroSetup** function configures the location of the shorted channel and the channels to be zeroed within a scan, the size of the scan, and the number of readings to zero compensate. (This function does not do the conversion.) A non-zero return value indicates an invalid parameter error.

In this example, several Daq* channels will be read using various gains and zero-compensated to remove any offset errors. This example assumes that channel 0 of the Daq* has been manually shorted. Expansion cards could perform the same type of zero compensation as this example by shorting a channel on the expansion card and specifying card channel numbers. First list the configuration:

Channel	Channel Type
0	Shorted Channel
1	Voltage1 @ X1 gain
2	Voltage2 @ X2 gain
3	Voltage3 @ X2 gain
4	Voltage4 @ X2 gain

Now specify the scan, the sequence of channel numbers, and gains that are to be gathered as one burst of readings. In this example, we will first read the shorted channel at each gain that we plan on using, in this case ×1 and ×2. All the channels at each gain will be read together to make the actual zero compensation easier.

Scan Position	Channel Type	Channel	Gain Code
0	Shorted Channel @ X1	0	DgainX1
1	Shorted Channel @ X2	0	DgainX2
2	Voltage1 @ X1 gain	1	DgainX1
3	Voltage2 @ X2 gain	2	DgainX2
4	Voltage3 @ X2 gain	3	DgainX2
5	Voltage4 @ X2 gain	4	DgainX2

```
Public Sub ZeroComp()
' Performs zero compensation on ADCs readings
Const ScanLength& = 6  'Total channels per scan
Const ScanCount& = 5   'Number of scans to acquire
Dim chan&(ScanLength), gain&(ScanLength)
Dim flag&(ScanLength), buf%(ScanLength * ScanCount)
Dim ret&, daqHandle&
```

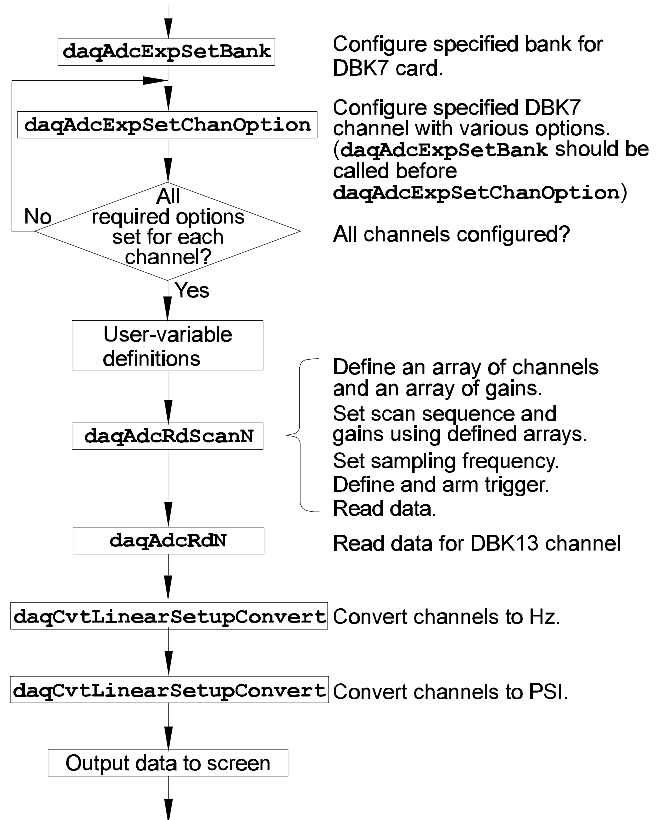
```
daqHandle& = VBdaqOpen&("daqbook0")

' Channel zero must be shorted to ground
' Use DafClearLSNibble flag to clear 4 least significant
' bits when using 12-bit A/D converters
chan&(0) = 0: gain&(0) = DgainX1&: flag&(0) = DafBipolar& +
 DafClearLSNibble&
chan&(1) = 0: gain&(1) = DgainX2&: flag&(1) = DafBipolar& +
 DafClearLSNibble&
chan&(2) = 1: gain&(2) = DgainX1&: flag&(2) = DafBipolar& +
 DafClearLSNibble&
chan&(3) = 2: gain&(3) = DgainX2&: flag&(3) = DafBipolar& +
 DafClearLSNibble&
chan&(4) = 3: gain&(4) = DgainX2&: flag&(4) = DafBipolar& +
 DafClearLSNibble&
chan&(5) = 4: gain&(5) = DgainX2&: flag&(5) = DafBipolar& +
 DafClearLSNibble&
ret& = VBdaqAdcSetScan&(daqHandle&, chan&(), gain&(), flag&(),
 ScanLength)
ret& = VBdaqAdcSetTrig&(daqHandle&, DatsImmediate&, 1, 0, 0, 0)
ret& = VBdaqAdcSetAcq&(daqHandle&, DaamNShot&, 0, ScanCount)
ret& = VBdaqAdcSetClockSource&(daqHandle&, DacsAdcClock&)
ret& = VBdaqAdcSetFreq&(daqHandle&, 100!)
ret& = VBdaqAdcTransferSetBuffer&(daqHandle&, buf%(), ScanCount,
 DatmCycleOff& + DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart&(daqHandle&)
ret& = VBdaqAdcArm&(daqHandle&)
ret& = VBdaqWaitForEvent&(daqHandle&, DteAdcDone&)

' Print the first scan of unconverted data
Print "Channel zero shorted to ground"
Print "Channel 0 at X1 gain: "; IntToUint(buf%(0))
Print "Channel 0 at X2 gain: "; IntToUint(buf%(1))
Print
Print "Before zero compensation"
Print "Channel 1 at X1 gain: "; IntToUint(buf%(2))
Print "Channel 2 at X2 gain: "; IntToUint(buf%(3))
Print "Channel 3 at X2 gain: "; IntToUint(buf%(4))
Print "Channel 4 at X2 gain: "; IntToUint(buf%(5))
Print
' Perform zero compensation on readings sampled at x1 gain.
' 1 reading at position 2. Zero reading at position 0.
ret& = VBdaqZeroSetupConvert&(ScanLength, 0, 2, 1, buf%(), ScanCount)

' Perform zero compensation on readings sampled at x2 gain.
' 3 readings at position 3. Zero reading at position 1.
ret& = VBdaqZeroSetupConvert&(ScanLength, 1, 3, 3, buf%(), ScanCount)

' Print the first scan of converted data
Print "After zero compensation"
Print "Channel 1 at X1 gain: "; IntToUint(buf%(2))
Print "Channel 2 at X2 gain: "; IntToUint(buf%(3))
Print "Channel 3 at X2 gain: "; IntToUint(buf%(4))
Print "Channel 4 at X2 gain: "; IntToUint(buf%(5))
Print
' Close the device
ret& = VBdaqClose&(daqHandle&)
End Sub

Function IntToUint(intval As Integer) As Long
' Converts 16-bit signed integer to unsigned long integer
If 0 <= intval Then
    IntToUint = intval
Else
    IntToUint = 65535 + CLng(intval) + 1
End If
End Function
```

Linear Conversion

Several DBKs use conversions from A/D readings to corresponding values that are a linear (straight-line) relationship. (Non-linear relationships for RTDs and thermocouples require special conversion functions—refer to the *Thermocouple and RTD Linearization* section later in this chapter.) The linear conversion functions are built into the API.

Six parameters are used to specify a linear relationship: the A/D input range (minimum and maximum values), and the transducer input signal level and output voltage at two points in the range.

Three functions are used to perform linear conversions: **daqCvtLinearSetup, daqCvtLinearConvert,** and **daqCvtLinearSetupConvert**. These functions are defined in the following pages. After their definitions, parameter examples and a program example show how they work.



DBK7 programmed for 50 to 60 Hz:

Measurement	Signal	Voltage
1	50 Hz	-5 V
2	60 Hz	+5 V

The DBK7 output range is from -5 V to +5 V, and the Daq* must be configured for bipolar operation at a gain of ×1 for the DBK7 channels. Thus, the input range -5 V to +5 V corresponds to the **ADmin** and **ADmax** settings. When a DBK7 programmed for a 50 to 60 Hz range measures a 50 Hz input signal, it outputs -5 V. With a 60 Hz input signal, it outputs +5 V. Thus, **signal1** is 50, **voltage1** is -5, **signal2** is 60, and **voltage2** is 5.

Pressure-transducer:

Measurement	Signal	Voltage
1	0 psi	1 mV
2	1000 psi	4 mV

Assume that a pressure transducer outputs 1 to 4 mV to represent 0 to 1000 psi, and that a DBK13 with a gain of ×1000 is used with a Daq* in bipolar mode to measure the signal. In bipolar mode, at a gain of 1000, the analog signal input range is -5 to 5 mV and the output range from the DBK13 is -5 to 5 Volts. Thus, **ADmin** is -5.000, and **ADmax** is 5.000. A pressure of 0 psi generates an output of 1 mV, and 1000 psi generates 4 mV. Thus **signal1** is 0, **voltage1** is 1.000, **signal2** is 1000 and **voltage2** is 4.000.

This program uses the linear conversion functions to convert voltage readings from a DBK7 frequency-to-voltage card and a DBK13 voltage input card with a pressure transducer to actual frequencies (Hz) and pressures (psi).

```
Public Sub LinearConvert()
Dim buffer1%(80), buffer2%(80), flags&(3), hz!(20), psi!(10)
Dim ret&, handle&, chan&, x%

' Initialize DaqBook
handle& = VBdaqOpen&("DaqBook0")

'Set Channel 16 to be a DBK7.  This will configure and auto-
'calibrate all channels on the DBK7 which includes channels
'16,17,18, and 19.  This step not required for a DBK13
ret& = VBdaqAdcExpSetBank(handle&, 16, DbankDbk7&)
```

```
'Set channel option common to all DBK7 channels.
'This step not required by a DBK13.
For chan& = 16 To 19
   ret& = VBdaqAdcExpSetChanOption(handle&, chan&, DcotDbk7Slope&, 1)
   ret& = VBdaqAdcExpSetChanOption(handle&, chan&, DcotDbk7DebounceTime&,
 DcovDbk7DebounceNone&)
   ret& = VBdaqAdcExpSetChanOption(handle&, chan&, DcotDbk7MinFreq&, 50!)
   ret& = VBdaqAdcExpSetChanOption(handle&, chan&, DcotDbk7MaxFreq&, 60!)
Next chan&

'Channel configuration:
'DaqBook Channels 16, 17: DBK7  channels 0,1
'DaqBook Channel      32: DBK13 channel  0
'Configure the pacer clock, arm the trigger, and acquire 10
'scans. The gain setting of Dbk7X1 (X1 gain) will be applied
'to all channels.  The acquisition frequency is set to 100 Hz.
'All channels are unsigned - bipolar.
ret& = VBdaqAdcRdScanN&(handle&, 16, 17, buffer1%(), 10, DatsAdcClock&,
 0, 0, 100!, Dbk7X1&, DafUnsigned& + DafBipolar&)

'Now do the same for the DBK13 channel, using gain Dbk13X1000
ret& = VBdaqAdcRdN(handle, 32, buffer2%(), 10, DatsAdcClock&, 0, 0, 100!,
 Dbk13X1000&, DafUnsigned& + DafUnipolar&)

'Convert channels 16 and 17 to Hertz where -5 volts corresponds
'to 50 Hz and 5 volts corresponds to 60 Hz.
ret& = VBdaqCvtLinearSetupConvert(2, 0, 2, 50!, -5!, 60!, 5!, 1,
 buffer1%(), 10, hz!(), 20)

'Convert channel 32 to PSI where 1mV corresponds to 0 PSI and
'4 mV corresponds to 1000 PSI. DBK13 channel 0 has 1000x gain,
'so 1mV at Dbk13 input gives 1V output at DaqBook input.
ret& = VBdaqCvtLinearSetupConvert(1, 0, 1, 0!, 1!, 1000!, 4!, 1,
 buffer2%(), 10, psi!(), 10)

'Print results
Print "Results:"
For x = 0 To 9
    Print Format(hz!(x * 2), "#0.00 Hz "); Format(hz!(x * 2 + 1), "#0.00
 Hz "); Format(psi(x), "0000.0 psi")
Next x

ret& = VBdaqClose(handle&)
End Sub
```

Summary Guide of Selected Enhanced API Functions

The following table organizes the enhanced API functions by type and includes notes on when to use them.

Simple One-Step Routines

For single gain, consecutive channel, foreground transfers, use the following functions:

Foreground Operation	Single Scan	Multiple Scans
Single Channel	`daqAdcRd`	`daqAdcRdN`
Consecutive Multiple Channels	`daqAdcRdScan`	`daqAdcRdScanN`

Complex A/D Scan Group Configuration Routines

For non-consecutive channels, high-speed digital channels, multiple gain settings, or multiple polarity settings, use the SetScan functions.

`daqAdcSetScan`	Set scan sequence using arrays of channel and gain values.
`daqAdcSetMux`	Set a contiguous scan sequence using single gain, polarity and channel flag values

Trigger Options

After the scan is set, the trigger needs to be set. The two triggering modes are one-shot or continuous.

- In one-shot mode, a trigger is required to start each A/D scan.
- A single trigger starts the scans, and the pacer clock determines the rate between scans.

Note: If the trigger source is analog, a trigger level is also required.

`daqAdcSetTrig`	Configure the trigger event using source, level, rising and channel values.
`daqAdcCalcTrig`	Using the selected trigger voltage, trigger direction, channel gain, and reference voltage, return the analog trigger source and value which can be used with `daqAdcSetTrig`.

If a software trigger is selected, the start time of the scan depends on the application calling `daAdcSoftTrig`.

Multiple Scan Timing

If the acquisition is to have multiple scans and the trigger mode is one-shot, the pacer clock needs to be set with one of the following functions:

`daqAdcSetRate`	Set/Get the specified frequency or period for the specified mode.
`daqAdcSetFreq`	Set the pacer clock to the given frequency.

A/D Acquisition

A/D acquisition settings are not active until the acquisition is armed.

`daqAdcArm`	Arm an A/D acquisition using the current configuration. If the trigger source was set to be immediate, the acquisition will be triggered immediately.
`daqAdcDisarm`	Disarm the current acquisition if one is active. This command will disarm the current acquisition and terminate any current A/D transfers.
`daqAdcSetAcq`	Define the mode of the acquisition and set the pre-trigger and post-trigger acquisition counts, if applicable.
`daqAdcAcqGetStat`	Return the current state of the acquisition as well as the total number of scans transferred thus far as well as the trigger scan position, if applicable.

A/D Data Transfer

After the acquisition is started, the data needs to be transferred to the application buffer. Three routines are used:

`daqAdcTransferSetBuffer`	Configure a buffer for A/D transfer. Allows configuration of the buffer for block and single reading update modes as well as linear and circular buffer definitions.
`daqAdcTransferStart`	Start a transfer from the Daq* device to the buffer specified in the `daqAdcTransferSetBuffer` command
`daqAdcTransferStop`	Stop a transfer from the Daq* device to the buffer specified in the `daqAdcTransferSetBuffer` command

To find out whether a background A/D transfer is complete or to stop transfers, use the following function:

`daqAdcTransferGetStat`	Return current A/D transfer status as well as a count representing the total number of transferred scans or the number of scans available.

D/A Conversions

The 2 D/A outputs are multiplying DACs. The voltage output is a fraction of the voltage reference. This fraction is the digital value sent to the DAC divided by 4096. Using the internal -5 V reference, any voltage between 0 and 4.9988 V can be set. Two routines are used to set the D/A outputs:

`daqDacWt`	Set a single DAC.
`daqDacWtBoth`	Set both DACs.

DAC1 is also set by any A/D routine which uses analog triggering. This DAC is used to set the comparison level.

Digital Functions

Several routines read and write the digital inputs and outputs. The first routine to call is the configure routine:

`daqIOSetConf`	Using the 4 port input/output direction selections, return a configuration byte.
`daqIOConf`	Set the input/output configuration of a local or expansion port group.

After the digital group is configured, the ports can be read or written a byte at a time. (Port C low/high and P1 digital I/O are accessed a nibble at a time.) A single bit of a digital channel can be read or written using the following routines:

`daqIORdBit`	Return indicated bit from selected channel.
`daqIOWrBit`	Send indicated bit to selected channel.

Counter Functions

Three counter/timer elements are in a DaqBook/112; and 9 counter/timer elements are in a DaqBook/100/200. Two counters are the ADC pacer clock. The FOUT counter element is a simple square-wave generator. Counter 0 is capable of more complex waveform and counter operations. Counters 1 through 5 are full-fledged counter/timer elements with many operating modes.

Counter 1 - Counter 5 Functions - For the DaqBook/100/200 Only

Counters 1 through 5 are binary/BCD, up/down 16-bit counters that can be internally cascaded. Each counter is capable of 24 modes including: hardware and software triggered strobes, rate generator, retriggerable and non-retriggerable one-shots, software and hardware-triggered delayed one-shots, variable duty-cycle rate generator, rate generator with sync, frequency-shift keying, and hardware save. Most modes can be gated. Counters 1 and 2 can be set up as a time-of-day counter, with 100 Hz resolution. Counters 1 and 2 are also capable of alarm outputs. In the alarm mode, whenever the counter value equals the alarm value, the counter output is set. This can be used with the time-of-day mode to cause an alarm at a particular time of day. To use counters 1 through 5 or the FOUT square-wave generator, the master mode register must be set:

`daq9513SetMasterMode`	Set FOUT source and scaler. Also set the counters 1 and 2 alarm mode and time-of-day mode.
`daq9513SetAlarm`	Set the alarm comparison value for counter 1 or 2.

9513 Counter-Timer Functions

The low-level counter functions allow custom-programming of the 9513 counters. After setting the Master Mode, counters 1 through 5 can be programmed using the following commands:

`daq9513SetCtrMode`	Set counter to given mode.
`daq9513SetLoad`	Set counter load register.
`aq9513SetHold`	Set counter hold register.

To read back a given counter, use one or both of:

`daq9513MultCtrl`	Issue a command to the indicated counters. To read the current contents of a counter, issue the `DmccSave` command, and read the hold register.
`daq9513GetHold`	Read a given hold register.

Notes

Daq* Command Reference (Enhanced API) 3

Overview

The first part of this chapter describes the Daq* driver commands for Windows95 and WindowsNT in 32-bit Enhanced mode (this is the **Enhanced API** and is not to be confused with the **Standard** API). The first table lists the commands by their function types as defined in the driver header files. Then, the prototype commands are described in alphabetical order as indexed below.

Beginning on page 73, several reference tables define parameters for: A/D channel descriptions, event-handling definitions, hardware definitions, A/D gain and miscellaneous definitions, general I/O definitions, digital I/O port connection, the API error codes, etc.

Function	Description	Page
Device Initialization Prototypes		
`daqOpen`	Open a session with the Daq*	3-66
`daqClose`	End communication with the Daq*	3-38
`daqOnline`	Check online status of the Daq*	3-65
`daqGetDeviceCount`	Return the number of currently configured devices	3-60
`daqGetDeviceList`	Return the list of currently configured devices	3-61
`daqGetDeviceProperties`	Return the properties of specified device	3-61
Error Handler Function Prototypes		
`daqSetDefaultErrorHandler`	Set the default error handler	3-67
`daqSetErrorHandler`	Specify a user defined routine to call when an error occurs in any command	3-67
`daqProcessError`	Process a driver defined error condition	3-66
`daqGetLastError`	Return the last logged error condition	3-62
`daqDefaultErrorHandler`	Call the default error handler	3-59
`daqFormatError`	Return text string for specified error	3-60
Event Handling Function Prototypes		
`daqSetTimeout`	Set the time-out value for the Daq* operation	3-68
`daqWaitForEvent`	Wait for specified Daq* device event	3-70
`daqWaitForEvents`	Wait for multiple specified Daq* device events	3-70
Utility Function Prototypes		
`daqGetDriverVersion`	Return the software version	3-62
`daqGetHardwareInfo`	Return the hardware version	3-62
Expansion Configuration Prototypes		
`daqAdcExpSetBank`	Set bank specific configurations	3-16
`daqAdcExpSetChanOption`	Set channel specific configurations	3-16
`daqAdcExpSetModuleOption`	Set module specific configurations	3-17
`daqSetOption`	Set options for a device's channel/signal path configuration	3-68
Custom ADC Acquisition Prototypes - Scan Sequence		
`daqAdcSetMux`	Configure a scan specifying start and end channels	3-25
`daqAdcSetScan`	Configure up to 256 channels making up an A/D or HS digital input scan	3-26
`daqAdcGetScan`	Read the current scan configuration	3-18
Custom ADC Acquisition Prototypes - Trigger		
`daqAdcCalcTrig`	Calculate the trigger level and trigger source for an analog trigger	3-15
`daqAdcSetTrig`	Configure an A/D trigger	3-27
`daqAdcSetTrigEnhanced`	Configure an A/D trigger with multiple trigger-event conditions	3-28
`daqAdcSoftTrig`	Save a software trigger command to the DaqBook/DaqBoard	3-29
Custom ADC Acquisition Prototypes - Scan Rate and Source		
`dacAdcSetRate`	Configure the ADC scan rate with the `mode` parameter	3-25
`daqAdcSetFreq`	Configure the pacer clock frequency in Hz	3-24
`daqAdcGetFreq`	Read the current pacer clock frequency	3-17
`daqAdcSetClockSource`	Configure the clock source	3-23
Custom ADC Acquisition Prototypes - Scan Count, Rate and Source		
`daqAdcSetAcq`	Set acquisition configuration information	3-22
Custom ADC Acquisition Prototypes - Direct-to-Disk		
`daqAdcSetDiskFile`	Specify the disk file for direct-to-disk transfers	3-24
Custom ADC Acquisition Prototypes - Acquisition Control		
`daqAdcArm`	Arm an acquisition	3-13
`daqAdcDisarm`	Disarm an acquisition	3-15

Function	Description	Page
Custom ADC Acquisition Prototypes - Data Transfer without Buffer Allocation		
`daqAdcTransferBufData`	Transfer scans from driver-allocated buffer to user-specified buffer	3-30
`daqAdcTransferSetBuffer`	Setup a destination buffer for an ADC transfer	3-32
`daqAdcTransferStart`	Start an ADC transfer	3-33
`daqAdcTransferGetStat`	Retrieve status of an ADC transfer	3-31
`daqAdcTransferStop`	Stop an ADC transfer	3-33
Custom ADC Acquisition Prototypes - Buffer Manipulation		
`daqAdcBufferRotate`	Reorganize a circular buffer so that oldest data is oriented towards the front	3-14
One-Step ADC Acquisition Prototypes		
`daqAdcRd`	Configure an A/D acquisition and read one sample from a channel	3-18
`daqAdcRdScan`	Configure an A/D acquisition and read one scan	3-20
`daqAdcRdN`	Configure an A/D acquisition and read multiple scans from a channel	3-19
`daqAdcRdScanN`	Configure an A/D acquisition and read multiple scans	3-21
Data Format and Conversion Prototypes		
`daqAdcSetDataFormat`	Set the raw and post-acquisition data formats	3-23
`daqCvtRawDataFormat`	Convert raw data to a specified format	3-41
`daqCvtSetAdcRange`	Set the ADC Voltage Range for the conversion routines	3-44
DAC Global Configuration Prototype		
`daqDacSetOutputMode`	Set the output mode for DAC FIFO	3-49
DAC Voltage Output Mode Prototypes		
`daqDacWt`	Output a D/A value	3-58
`daqDacWtMany`	Output D/A values to several DACs	3-59
DAC Waveform Prototypes - Trigger, Update Rate and Count		
`daqDacWaveSetTrig`	Configure the trigger to initiate waveform output	3-57
`daqDacWaveSoftTrig`	Trigger the DAC waveform output via software	3-58
`daqDacWaveSetClockSource`	Set the clock source for DAC waveform output frequency	3-54
`daqDacWaveSetFreq`	Set the DAC waveform output frequency	3-55
`daqDacWaveGetFreq`	Get the current DAC waveform output frequency	3-52
`daqDacWaveSetMode`	Set the DAC waveform output mode	3-55
DAC Waveform Prototypes - Buffer Management		
`daqDacWaveSetDiskFile`	Set DAC waveform output source to disk file	3-54
`daqDacWaveSetPredefWave`	Specify a predefined DAC waveform for output	3-56
`daqDacWaveSetUserWave`	Specify a user-defined DAC waveform for output	3-57
`daqDacWaveSetBuffer`	Setup a buffer for DAC waveform output	3-53
DAC Waveform Prototypes - Waveform Control		
`daqDacWaveArm`	Arm triggering for DAC waveform output	3-51
`daqDacWaveDisarm`	Disarm triggering for DAC waveform output	3-52
DAC Transfer Prototypes - Dynamic Waveform Data Transfer		
`daqDacTransferStart`	Start a DAC waveform output	3-50
`daqDacTransferGetStat`	Get status of a current DAC waveform output	3-50
`daqDacTransferStop`	Stop the current DAC waveform output	3-51
Linear Conversion Prototypes		
`daqCvtLinearSetup`	Save data required for daqCvtLinearConvert	3-39
`daqCvtLinearConvert`	Convert ADC readings into floating point numbers	3-39
`daqCvtLinearSetupConvert`	Combine setup and conversion into one function	3-40
Software Calibration Prototypes		
`daqCalSetup`	Configure the order and type of data to be calibrated	3-37
`daqCalConvert`	Perform the actual calibration of one or more scans	3-37
`daqCalSetupConvert`	Perform both the setup and convert steps with one call	3-38
`daqReadCalFile`	Read all the calibration constants from the specified file	3-67
`daqCalSelectCalTable`	Select calibration-table source for the device	3-35
`daqCalSelectInputSignal`	Select input signal source for user calibration	3-36
`daqCalGetConstants`	Get calibration constants from selected calibration table	3-34
`daqCalSetConstants`	Set user-accessible calibration constants	3-36
`daqCalSaveConstants`	Save current calibration table	3-35
Zero Offset Prototypes		
`daqZeroSetup`	Configure data for zero compensation	3-72
`daqZeroConvert`	Perform zero compensation on one or more scans	3-71
`daqZeroSetupConvert`	Perform both the setup and convert steps with one call	3-72
`daqZeroDbk19`	Configure the thermocouple linearization functions to automatically perform zero compensation	3-71
RTD Conversion Prototypes		
`daqCvtRtdConvert`	Convert raw A/D readings from RTDs to temperature readings	3-42

Function	Description	Page
`daqCvtRtdSetup`	Set up parameters for subsequent RTD temperature conversions	3-43
`daqCvtRtdSetupConvert`	Set up and convert raw A/D readings from RTDs into temperature readings	3-44
Thermocouple Conversion Prototypes		
`daqCvtTCConvert`	Convert raw A/D readings from thermocouples to temperature readings	3-45
`daqCvtTCSetup`	Set up parameters for subsequent thermocouple temperature conversions	3-47
`daqCvtTCSetupConvert`	Set up and convert raw A/D readings from thermocouples into temperature readings	3-48
General I/O Prototypes - Read/Write		
`daqIOReadBit`	Read a DIO bit (channel)	3-64
`daqIORead`	Read a DIO byte (8 channels)	3-63
`daqIOWriteBit`	Write a DIO bit (channel)	3-65
`daqIOWrite`	Write a DIO byte (8 channels)	3-64
`daqIOGet8255Conf`	Get the current configuration of the DIO	3-63
9513 Counter/Timer Prototypes		
`daq9513SetMasterMode`	Initialize various counter/timer values	3-11
`daq9513SetAlarm`	Set the specified alarm register	3-6
`daq9513SetCtrMode`	Set the 9513's mode register for the specified counter	3-7
`daq9513MultCtrl`	Simultaneously configure multiple counters	3-4
`daq9513GetHold`	Read the hold register of the specified counter	3-3
`daq9513SetHold`	Output a value to the counter hold register	3-10
`daq9513SetLoad`	Output a value to the counter load register	3-10
`daq9513RdFreq`	Read up to 9 frequency inputs	3-5
Test Prototypes		
`daqTest`	Perform a specified test on a Daq* device	3-69

Commands in Alphabetical Order

The following pages give the details for each API command. Listed in alphabetical order, each section starts with a table that summarizes the main features of the command (C, Visual BASIC, and Delphi language prototypes and their related parameters). An explanation follows with related information and in some cases a programming example. **Typographic note**: Commands, parameters, values, and code all use a bold, mono-spaced **`Courier`** font to distinguish characters that can be ambiguous in other fonts.

daq9513GetHold

DLL Function	`daq9513GetHold(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
C	`daq9513GetHold(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
Visual BASIC	`VBdaq9513GetHold&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal ctrNum&, ctrVal%)`
Delphi	`daq9513GetHold(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; ctrNum:DWORD; var crtVal:WORD)`
Parameters	
`handle`	Handle to the device to get the 9513 hold register
`deviceType`	Specifies the 9513 device type
`whichDevice`	Specifies which 9513
`ctrNum`	The counter number Valid values: 1 - 5
`ctrVal`	The value read from the hold register of the selected counter is placed in this variable Valid values: 0 - 65535
Returns	`DerrInvCtrNum` - Invalid counter `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daq9513SetCtrMode`
Program References	None
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daq9513GetHold` reads the hold register of the specified counter. This register is used in event-counting applications to store accumulated counter values. Without interrupting the process, the hold register can be read while the count process is running.

daq9513MultCtrl

DLL Function	`daq9513MultCtrl(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, Daq9513MultCtrCommand ctrCmd, BOOL ctr1, BOOL ctr2, BOOL ctr3, BOOL ctr4, BOOL ctr5);`
C	`daq9513MultCtrl(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, Daq9513MultCtrCommand ctrCmd, BOOL ctr1, BOOL ctr2, BOOL ctr3, BOOL ctr4, BOOL ctr5);`
Visual BASIC	`VBdaq9513MultCtrl&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal ctrCmd&, ByVal ctr1&, ByVal ctr2&, ByVal ctr3&, ByVal ctr4&, ByVal ctr5&)`
Delphi	`daq9513MultCtrl(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; ctrCmd:Daq9513MultCtrCommand; ctr1:longbool; ctr2:longbool; ctr3:longbool; ctr4:longbool; ctr5:longbool)`
Parameters	
`handle`	Handle to the device for which to set multiple counter commands
`deviceType`	Specifies the 9513 device type (DiodtLocal9513)
`whichDevice`	Specifies which 9513
`ctrCmd`	The counter command (see below)
`ctr1`	A flag that if non-zero enables the counter command to be executed on counter 1, or if 0 do nothing to counter 1
`ctr2`	A flag that if non-zero enables the counter command to be executed on counter 2, or if 0 do nothing to counter 2
`ctr3`	A flag that if non-zero enables the counter command to be executed on counter 3, or if 0 do nothing to counter 3
`ctr4`	A flag that if non-zero enables the counter command to be executed on counter 4, or if 0 do nothing to counter 4
`ctr5`	A flag that if non-zero enables the counter command to be executed on counter 5, or if 0 do nothing to counter 5
Multiple Counter Commands	
Description	Value
`DmccArm`	`0`
`DmccLoad`	`1`
`DmccLoadArm`	`2`
`DmccDisarmSave`	`3`
`DmccSave`	`4`
`DmccDisarm`	`5`
Returns	`DerrInvCtrCmd` - Invalid counter command `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCtrSetCtrMode, daqCtrSetMasterMode`
Program References	DAQEX.FRM (VB) CTREX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daq9513MultCtrl` performs a command including: loading, latching, saving, enabling, and disabling on multiple counters simultaneously. The commands work as follows:

- The load command can transfer the initial counter value from the load or hold register.
- The arm command will enable the counter to begin counting.
- The disarm command will disable the counter.
- The save command will transfer the current counter value to the hold register, where it can be read without disturbing the counters.

daq9513RdFreq

DLL Function	`daq9513RdFreq(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD interval, Daq9513CountSource cntSource, PDWORD count);`	
C	`daq9513RdFreq(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD interval, Daq9513CountSource cntSource, PDWORD count);`	
Visual BASIC	`VBdaq9513RdFreq&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal interval&, ByVal cntSource&, count&)`	
Delphi	`daq9513RdFreq(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; interval:DWORD; cntSource:Daq9513CountSource; var count:DWORD)`	
Parameters		
`handle`	Handle to the device in which to get 9513 frequency	
`deviceType`	Specifies 9513 device type (DiodtLocal9513)	
`whichDevice`	Specifies which 9513	
`interval`	The gate interval in milliseconds Valid values: `1 - 32767`	
`cntSource`	The count source (see below)	
`count`	A variable to hold the number of counts accumulated in the gating interval Valid values: `0 - 65535`	
Count Source Definitions		
Definition	**Value**	**Description**
`DcsSrc1`	`1`	Counter 1 input (pin 36 of P3)
`DcsSrc2`	`2`	Counter 2 input (pin 19 of P3)
`DcsSrc3`	`3`	Counter 3 input (pin 17 of P3)
`DcsSrc4`	`4`	Counter 4 input (pin 15 of P3)
`DcsSrc5`	`5`	Counter 5 input (pin 13 of P3)
`DcsGate1`	`6`	Counter 1 gate (pin 37 of P3)
`DcsGate2`	`7`	Counter 2 gate (pin 18 of P3)
`DcsGate3`	`8`	Counter 3 gate (pin 16 of P3)
`DcsGate4`	`9`	Counter 4 gate (pin 14 of P3)
Returns	`DerrInvInterval` - Invalid interval `DerrInvCntSource` - Invalid source `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)	
See Also		
Program References	None	
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A	

`daq9513RdFreq` is used to read the frequency of one of 9 external inputs. The 9 available inputs include the 5 counter inputs (P3 pins 36, 19, 17, 15, or 13) and the gates of counters 1 to 4 (P3 pins 37, 18, 16, and 14). This function counts the number of pulses on the specified input within a specified time interval, thereby providing the frequency of the signal. This frequency can be obtained by dividing the number of pulses by the interval (frequency in kHz = count/interval).

Note: The counter 4 output (P3 pin 32) must be externally connected to the counter 5 gate (P3 pin 12). This function will reconfigure counters 4 and 5.

daq9513SetAlarm

DLL Function	`daq9513SetAlarm(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD alarmNum, DWORD alarmVal);`
C	`daq9513SetAlarm(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD alarmNum, DWORD alarmVal);`
Visual BASIC	`VBdaq9513SetAlarm&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal alarmNum&, ByVal alarmVal&)`
Delphi	`daq9513SetAlarm(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; alarmNum:DWORD; alarmVal:DWORD)`
Parameters	
`handle`	Handle to the device which to set the 9513 alarm
`deviceType`	Specifies the 9513 device type (DiodtLocal9513)
`whichDevice`	Specifies which 9513.
`AlarmNum`	The alarm register number Valid values: `1 - 2`
`alarmVal`	The value to write to the selected alarm register Valid values: `0 - 65535`
Returns	`DerrInvCtrNum` - Invalid counter number `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCtrSetMasterMode`
Program References	None
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daq9513SetAlarm` sets the specified alarm register. This alarm register can be used with the comparators described in **`daq9513SetMasterMode`**. The alarm register is only used if the corresponding comparator has been enabled using the **`daq9513SetMasterMode`** function.

daq9513SetCtrMode

DLL Function	`daq9513SetCtrMode(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, Daq9513GatingControl gateCtrl, BOOL cntEdge, Daq9513CountSource cntSource, BOOL specGate, BOOL reload, BOOL cntRepeat, BOOL cntType, BOOL cntDir, Daq9513OutputControl outputCtrl);`
C	`daq9513SetCtrMode(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, Daq9513GatingControl gateCtrl, BOOL cntEdge, Daq9513CountSource cntSource, BOOL specGate, BOOL reload, BOOL cntRepeat, BOOL cntType, BOOL cntDir, Daq9513OutputControl outputCtrl);`
Visual BASIC	`VBdaq9513SetCtrMode&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal ctrNum&, ByVal gateCtrl&, ByVal cntEdge&, ByVal cntSource&, ByVal specGate&, ByVal reload&, ByVal cntRepeat&, ByVal cntType&, ByVal cntDir&, ByVal outputCtrl&)`
Delphi	`daq9513SetCtrMode(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; ctrNum:DWORD; gateCtrl:Daq9513GatingControl; cntEdge:longbool; cntSource:Daq9513CountSource; specGate:longbool; reload:longbool; cntRepeat:longbool; cntType:longbool; cntDir:longbool; outputCtrl:Daq9513OutputControl)`

Parameters

Parameter	Description
`handle`	Handle to the device to set the 9513 counter mode
`deviceType`	Specifies the 9513 device type (DiodtLocal9513)
`whichDevice`	Specifies which 9513
`ctrNum`	The counter number; Valid values: `1 - 5`
`gateCtrl`	The gating control mode (see below)
`cntEdge`	A flag that if non-zero will select a positive count edge, or if `0` will select a negative count edge
`cntSource`	The count source (see below)
`specGate`	A flag that if non-zero will enable the special gate, or if `0` will disable it
`reload`	A flag that if non-zero will select reload from load or hold, or if `0` will select reload from load
`cntRepeat`	A flag that if non-zero will select count repetitively, or if `0` will select count once
`cntType`	A flag that if non-zero will select a BCD count, or if `0` will select a binary count
`cntDir`	A flag that if non-zero will select count up, or if `0` will select count down
`outputCtrl`	The output control mode (see below)

Gating Control Definitions:

Definition	Value	Description
`DgcNoGating`	0	Gating Disabled
`DgcHighTCNM1`	1	Active level high of TC toggled output of previous (N-1) counter
`DgcHighLevelGateNP1`	2	Active level high of gate of next (N+1) counter
`DgcHighLevelGateNM1`	3	Active level high of gate of next (N-1) counter
`DgcHighLevelGateN`	4	Active level high of gate of selected (N) counter
`DgcLowLevelGateN`	5	Active level low of gate of selected (N) counter
`DgcHighEdgeGateN`	6	Active rising edge of gate of selected (N) counter
`DgcLowEdgeGateN`	7	Active falling edge of gate of selected (N) counter

Count Source Definitions

Definition	Value	Description
`DcsTCNM1`	0	TC toggled output of previous (N-1) counter (not valid with daq9513SetMasterMode or daq9513RdFreq)
`DcsSrc1`	1	Counter 1 input (pin 36 of P3)
`DcsSrc2`	2	Counter 2 input (pin 19 of P3)
`DcsSrc3`	3	Counter 3 input (pin 17 of P3)
`DcsSrc4`	4	Counter 4 input (pin 15 of P3)
`DcsSrc5`	5	Counter 5 input (pin 13 of P3)
`DcsGate1`	6	Counter 1 gate (pin 37 of P3)
`DcsGate2`	7	Counter 2 gate (pin 18 of P3)
`DcsGate3`	8	Counter 3 gate (pin 16 of P3)
`DcsGate4`	9	Counter 4 gate (pin 14 of P3)
`DcsGate5`	10	Counter 5 gate (pin 12 of P3) (not valid with daq9513RdFreq)
`DcsF1`	11	Onboard 1 MHz clock (not valid with daq9513RdFreq)
`DcsF2`	12	Onboard 100 kHz clock (not valid with daq9513RdFreq)
`DcsF3`	13	Onboard 10 kHz clock (not valid with daq9513RdFreq)
`DcsF4`	14	Onboard 1 kHz clock (not valid with daq9513RdFreq)
`DcsF5`	15	Onboard 100 Hz clock (not valid with daq9513RdFreq)

Output Control Definitions:

Definition	Value	Description
`DocInactiveLow`	0	Inactive - Always low
`DocHighTermCntPulse`	1	High impulse on terminal count
`DocTCToggled`	2	Toggled on terminal count
`DocInactiveHighImp`	3	Inactive - High impedance
`DocLowTermCntPulse`	4	Low pulse on terminal count

Returns

`DerrInvCtrNum` - Invalid channel
`DerrInvGateCtrl` - Invalid gate

	`DerrInvCntSource` - Invalid source `DerrInvOutputCntrl` - Invalid output `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCtrSetLoad, daqCtrSetHold, daqCtrGetHold, daqCtrMultCtrl`
Program References	DAQEX.FRM (VB), CTREX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

daq9513SetCtrMode is used to set the 9513's mode register for a specified counter. Setting this register defines how the specific counter works for a variety of square waves, pulse generation, and event counting. To set the initial counter values, this function is often followed by **daq9513SetLoad** or **daq9513SetHold**. Finally, the **daq9513MultCtrl** function is called to load and arm multiple counters. **daq9513MultCtrl** can also be used to count events.

The gate control parameter controls how the counter will use its gate input (P3 pins 37, 18, 16, 14 and 12) or another counter's gate input.

- If the gate is disabled using the **DgcNoGating** definition, it will be ignored and the counter will run as long as it is armed.
- If a level gate control is selected (using the **DgcHighLevelGateNPI**, **DgcHighLevelGateNMI**, or **DgcHighLevelGateN** definitions), the counter will operate only while armed and the selected high or low level is applied to the gate.
- If an edge-sensitive gate control is selected using the **DgcHighEdgeGate** or **DgcHighEdgeGateN** definitions, the counter will operate after a rising or falling edge is detected on the gate input.

Most gate control modes select gate N (gate of the selected counter) or gate inputs of the previous (N-1) and next (N+1) counters. Thus, counter 3 could use the gate input of counter 2 by selecting N-1; counter 4 by selecting N+1; or its own gate input by selecting N. Counter 1 and counter 5 are considered adjacent when selecting gate input N+1 or N-1. The final gate control mode allows the TC-toggled output (see output control description) of the previous counter (N-1) to be the gate. The selected counter will operate only when the previous counter's TC-toggled output is high.

The Count Edge (**cntEdge**) flag selects whether the counter will count when it receives a rising or falling edge on its count source (see the count source description).

The Count Source (**cntSource**) selects the source used as input to the specified counter. The Count Edge selects whether the rising or falling edge of this source is counted. The Count Source can be any one of the counter inputs, **Src1** to **Src5** (P3 pins 36, 19, 17, 15 or 13), any one of the counter gates, **Gate1** to **Gate5** (P3 pins 37, 18, 17, 16 or 14), an internal frequency, **F1** to **F5**, or the TC-toggled output (see the output control description) of the previous counter (N-1). The internal frequencies are divide-by-10 divisions of the onboard oscillator which is by default 1 MHz, but can be jumpered to 10 MHz. The sources **F1** through **F5** correspond to the frequencies 1 MHz, 100 kHz, 10 kHz, 1 kHz and 100 Hz. The TC-toggled output of the previous counter can be used as a source—allowing counters to be cascaded without external connections.

The Count Direction (**cntDir**) selects whether the counter will count up or down. The counter is normally configured for down counting when generating a pulse or square wave. The load register would be set to a positive value which will decrement to zero, defining the duration or width of the waveform. In event counting, the counter would initially be set to zero and configured to count up (in this case, the hold register would contain the number of events received).

The Count Type (**cntType**) selects binary or BCD counting. Binary format accepts a 16-bit number ranging from **0** to **65,535**. BCD (binary-coded decimal) accepts four 8-bit numbers representing **0** to **10**, back in 16-bits, ranging from **0** to **9999**.

The Output Control (**outputCtrl**) parameter controls the state of the counter output (P3 pins 35, 34, 33, 32, 31). There are 2 inactive and 3 active output modes. If the output is inactive, it can either be driven low or it can be high impedance. The active modes are all associated with the terminal count (TC) which is the moment in time when the counter reaches 0. This can happen by counting up past 65535 in binary count mode or 9999 in BCD count mode, or counting down past 1. The output can be driven: high during the TC and low otherwise, low during the TC and high otherwise, or toggle the output every time a TC occurs. The TC-toggled mode is used to generate variable duty-cycle square waves.

The Count Repeat (**cntRepeat**), Reload (**reload**) and Special Gate (**specGate**) parameters

have complex relationships that define the operation of the counter. The Count Repeat flag enables/disables re-arming the counter after TC occurs. Applications such as software re-triggerable 1-shots would disable the repeat flag so the 1-shot occurs only after the counter arm command is sent. Other applications (such as rate generators, square waves and hardware re-triggerable 1-shots) would enable the count repeat so that the counter will run until disarmed.

The Reload flag programs the counter to use the count value in the load and/or hold registers for counting. If the reload flag is disabled, the counter will use the contents of the load register only for counting. Enabling the reload flag will allow the counter to use the contents of either or both registers depending on the special gate flag. If the reload flag is enabled and the special gate is disabled, the counter will alternate between registers. This allows a variable duty-cycle output waveform depending on the relative values of the hold and load registers. If the reload flag is enabled and the special gate is enabled, the operation will depend on the gate control parameter. In this situation, an active gate control will allow hardware re-triggering on the active-going edge. An inactive gate control will configure the counter to use the hold register for counting if the counter's gate is high or to use the load register if the gate is low. Refer to the *Am9513A/AM9513 Technical Manual* for further reference.

The next table summarizes the operating modes of the counter/timer.

Counter Mode Operating Summary

Counter Mode	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Special Gate (CM7)	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
Reload Source (CM6)	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1
Repetition (CM5)	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
Gate Control (CM15-CM-13); N=no gating; L=level; E=edge	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E
Count to TC once, then disarm	X	X	X											X	X									
Count to TC twice, then disarm							X	X	X										X					
Count to TC repeatedly without disarming				X	X	X				X	X	X					X	X				X		X
Gate input does not gate counter input	X			X			X			X									X			X		
Count only during active gate level		X			X			X			X			X			X							
Start count on active gate edge and stop count on next TC			X			X									X			X						X
Start count on active gate edge and stop count on second TC									X			X												
No hardware re-triggering	X	X	X	X	X	X	X	X	X	X	X	X							X			X		X
Reload counter from Load Register on TC	X	X	X	X	X	X								X	X		X	X						X
Reload counter on each TC, alternating reload source between Load and Hold Registers							X	X	X	X	X	X												
Transfer Load Register into counter on each TC that gate is LOW, transfer Hold Register into counter on each TC that gate is HIGH																			X			X		
On active gate edge transfer counter into Hold Register and then reload counter from Load Register														X	X		X	X						
On active gate edge transfer counter into Hold Register, but counting continues																								X

daq9513SetHold

DLL Function	`daq9513SetHold(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
C	`daq9513SetHold(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
Visual BASIC	`VBdaq9513SetHold&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal ctrNum&, ByVal ctrVal%)`
Delphi	`daq9513SetHold(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; ctrNum:DWORD; crtVal:WORD)`
Parameters	
`handle`	Handle to the device in which to set 9513 hold register
`deviceType`	Specifies the 9513 device type (DiodtLocal9513)
`whichDevice`	Specifies which 9513
`ctrNum`	The counter number Valid values: `1 - 5`
`ctrVal`	The value to write to the hold register of the selected counter Valid values: `0 - 65535`
Returns	`DerrInvCtrNum` - Invalid channel `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daq9513SetMasterMode, daq9513SetCtrMode`
Program References	DAQEX.FRM (VB), CTREX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daq9513SetHold` outputs a value to the hold register of the specified counter. The hold register can be used to set the counter's initial value using the **`daq9513MultCtrl`** function. The **`daq9513SetMasterMode`** and **`daq9513SetCtrMode`** functions describe various uses of the hold register.

daq9513SetLoad

DLL Function	`daq9513SetLoad(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
C	`daq9513SetLoad(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD ctrNum, PWORD ctrVal);`
Visual BASIC	`VBdaq9513SetLoad&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal ctrNum&, ByVal ctrVal%)`
Delphi	`daq9513SetLoad(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; ctrNum:DWORD; crtVal:WORD)`
Parameters	
`handle`	Handle to the device in which to set the 9513 load
`deviceType`	Specifies the 9513 device type (DiodtLocal9513)
`whichDevice`	Specifies which 9513
`ctrNum`	The counter number Valid values: `1 - 5`
`ctrVal`	The value to write to the load register of the selected counter Valid values: `0 - 65535`
Returns	`DerrInvCtrNum` - Invalid channel `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daq9513SetMasterMode, daq9513SetCtrMode`
Program References	DAQEX.FRM (VB), CTREX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daq9513SetLoad` outputs a value to the load register of the specified counter. The load register can be used to set the counter's initial value using the **`daq9513MultCtrl`**. **`daq9513SetMasterMode`** and **`daq9513SetCtrMode`** describe various uses of the load register.

daq9513SetMasterMode

DLL Function	daq9513SetMasterMode(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD foutDiv, Daq9513CountSource cntSource, BOOL comp1, BOOL comp2, Daq9513TimeOfDay tod);
C	daq9513SetMasterMode(DaqHandleT handle, DaqIODeviceType deviceType, DWORD whichDevice, DWORD foutDiv, Daq9513CountSource cntSource, BOOL comp1, BOOL comp2, Daq9513TimeOfDay tod);
Visual BASIC	VBdaq9513SetMasterMode&(ByVal handle&, ByVal deviceType&, ByVal whichDevice&, ByVal foutDiv&, ByVal cntSource&, ByVal comp1&, ByVal comp2&, ByVal tod&)
Delphi	daq9513SetMasterMode(handle:DaqHandleT; deviceType:DaqIODeviceType; whichDevice:DWORD; foutDiv:DWORD; cntSource:Daq9513CountSource; comp1:longbool; comp2:longbool; tod:Daq9513TimeOfDay)
Parameters	
handle	Handle to the device in which to set the 9513 master mode
deviceType	Specifies the 9513 device type (**DiodtLocal9513**)
whichDevice	Specifies which 9513
foutDiv	The fout divider. A divider of 0 selects divide by 16 Valid values: **1 -16**
cntSource	The fout source
comp1	A flag that if non-zero will enable the compare **1** operation, or if **0** will disable it
comp2	A flag that if non-zero will enable the compare **2** operation, or if **0** will disable it
tod	The time of day mode

Count Source Definitions:

Definition	Value	Description
DcsTcnM1	00h	Not valid with daq8513SetMasterMode or daq9513RdFreq
DcsSrc1	01h	Counter 1 input (pin 36 of P3)
DcsSrc2	02h	Counter 2 input (pin 19 of P3)
DcsSrc3	03h	Counter 3 input (pin 17 of P3)
DcsSrc4	04h	Counter 4 input (pin 15 of P3)
DcsSrc5	05h	Counter 5 input (pin 13 of P3)
DcsGate1	06h	Counter 1 gate (pin 37 of P3)
DcsGate2	07h	Counter 2 gate (pin 18 of P3)
DcsGate3	08h	Counter 3 gate (pin 16 of P3)
DcsGate4	09h	Counter 4 gate (pin 14 of P3)
DcsGate5	0Ah	Counter 5 gate (pin 12 of P3) (not valid with daq9513RdFreq)
DcsF1	0Bh	Onboard 1 MHz clock (not valid with daq9513RdFreq)
DcsF2	0Ch	Onboard 100 kHz clock (not valid with daq9513RdFreq)
DcsF3	0Dh	Onboard 10 kHz clock (not valid with daq9513RdFreq)
DcsF4	0Eh	Onboard 1 kHz clock (not valid with daq9513RdFreq)
DcsF5	0Fh	Onboard 100 Hz clock (not valid with daq9513RdFreq)

Time-Of-Day Definitions:

Description	Value
DtodDisabled	00h
DtodDivideBy5	01h
DtodDivideBy6	02h
DtodDivideBy10	03h

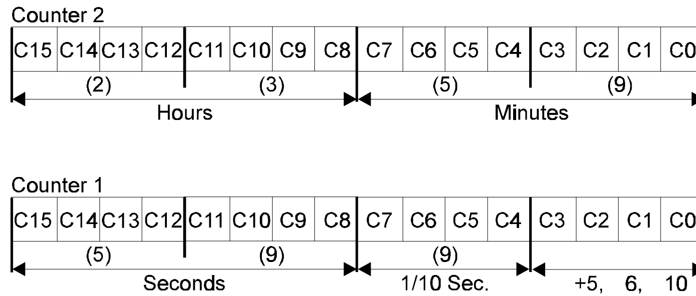
Returns	**DerrInvCntSource** - Invalid source **DerrInvTod** - Invalid time of day mode **DerrInvDir** - Invalid divisor **DerrNotCapable** - No 9513 available **DerrNoError** - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daq9513SetLoad, daq9513MultCtr, daq9513GetHold, daq9513SetCntMode
Program References	DAQEX.FRM (VB), CTREX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

daq9513SetMasterMode is used to set the counter's master mode register. This register is used to configure the fout pin (P3 pin 30), the comparators of counter 1 and 2, and the time-of-day operation of the 9513 chip. The master mode parameters default to zero after **daqOpen**.

The fout source selects what signal will be output on the fout pin. The fout source can be any one of the counter inputs, **Src1** to **Src5** (P3 pins 36, 19, 17, 15 or 13); any one of the counter gates, **Gate1** to **Gate5** (P3 pins 37, 18, 17, 16 or 14); or an internal frequency, **F1** to **F5** (1 MHz, 100 kHz, 10 kHz, 1 kHz and 100 Hz). The fout divider will divide the selected source by 1 to 16 before outputting the signal on fout.

The 2 comparator flags control the comparators associated with counter 1 and 2. If a comparator is

enabled, the value in the corresponding alarm register, set with the **`daqCtrSetAlarm`** function, will be compared with the value in the counter. The output of the corresponding counter will go **`true`** when the value in the counter reaches the value in the alarm register and remain **`true`** until the counter value changes. The polarity of the output depends on the output control, set with the **`daqCtrSetCtrMode`** function, configuration of counter 1 or 2. When output control is high, terminal count pulsed, or terminal count toggled, then the output will be high while the comparator is **`true`**. When the output control is low and terminal count pulsed, the output will be low while the comparator is **`true`**.



Time-of-Day Configuration

The time-of-day parameter is used to enable or disable the time-of-day operation. The time-of-day operation is a special mode which causes counters 1 and 2 to turn over at counts that generate 24-hour time-of-day accumulations. The resolution of the time-of-day operation is 0.1 seconds. A 100 Hz, 60 Hz or 50 Hz signal must be applied to the input of counter 1 (P3 pin 36), while in the divide-by-10, divide-by-6 and divide-by-5 time-of-day modes respectively. This will produce the 10 Hz clock source needed to drive the time-of-day clock. The hold registers of counters 1 and 2 will hold the 24-hour time.

The following steps must be performed to use the time-of-day operation:

1. Set the master mode register as described above.
2. For general-purpose time keeping, configure counter 1 using **`daqCtrSetCtrMode`** with the no gating, count on rising edge, special gating disabled, reload from hold only, count repetitively, BCD counting and count up. The count source can be any of the available sources. The output control does not affect time-of-day operation.
3. Set the mode of counter 2 with the same settings as counter 1, except the count source should be TC toggled of the previous (N-1) counter. This allows internal concatenation of counter 1 to counter 2.
4. Set the load registers of counter 1 and 2 to zero, using the **`daqCtrSetLoad`** function.
5. Initialize the current 24-hour time-of-day by setting the load registers of counters 1 and 2, using the format shown in the figure above (again using **`daqCtrSetLoad`**).
6. Repeat step 4.

daqAdcArm

DLL Function	`daqAdcArm(DaqHandleT handle);`
C	`daqAdcArm(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcArm&(ByVal handle&)`
Delphi	`daqAdcArm(handle:DaqHandleT)`
Parameters	
`handle`	Handle to the device to which configured ADC acquisition is to be armed
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcDisarm`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

daqAdcArm allows you to arm an ADC acquisition by enabling the currently defined ADC configuration for acquisition. ADC acquisition will occur when the trigger event (as specified by **daqAdcSetTrig**) is satisfied. All ADC acquisition configuration information must be specified prior to the **daqAdcArm** command. For a previously configured acquisition, the **daqAdcArm** command will use the specified parameters. If no previous configuration was given, or it is desirable to change any or all acquisition parameters, then those commands relating to the desired ADC acquisition configuration must be issued prior to calling **daqAdcArm**.

daqAdcBufferRotate

DLL Function	`daqAdcBufferRotate(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD chanCount, DWORD retCount);`
C	`daqAdcBufferRotate(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD chanCount, DWORD retCount);`
Visual BASIC	`VBdaqAdcBufferRotate&(ByVal handle&, buf%(), ByVal scanCount&, ByVal chanCount&, ByVal retCount&)`
Delphi	`daqAdcBufferRotate(handle:DaqHandleT; buf:PWORD; scanCount:DWORD; chanCount:DWORD; retCount:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC transfer buffer is to be rotated
`buf`	Pointer to the buffer to rotate
`scanCount`	Total number of scans in the buffer
`chanCount`	Number of channels in each scan
`retCount`	Last value returned in the retCount parameter of the daqAdcTransferGetStat function
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferGetStat, daqAdcTransferSetBuffer`
Program References	None
Used With	All devices

`daqAdcBufferRotate` allows you to linearize a circular buffer acquired via a transfer in cycle mode. This command will organize the circular buffer chronologically. In other words, it will order the data from oldest-first to newest-last in the buffer. When scans are acquired using **`daqAdcBufferTransfer`** with a non-zero cycle parameter, the buffer is used as a circular buffer; once it is full, it is re-used, starting at the beginning of the buffer. Thus, when the acquisition is complete, the buffer may have been overwritten many times and the last acquired scan may be any place within the buffer.

For example, during the acquisition of 1000 scans in a buffer that only has room for 60 scans, the buffer is filled with scans 1 through 60. Then scan 61 overwrites scan 1; scan 62 overwrites scan 2; and so on until scan 120 overwrites scan 60. At this point, the end of the buffer has been reached again and so scan 121 is stored at the beginning of the buffer, overwriting scan 61. This process of overwriting and re-using the buffer continues until all 1000 scans have been acquired. At this point, the buffer has the following contents:

Buffer Position	1	2	3	...	39	40	41	42	...	59	59	60
Scan	961	962	963	...	999	1000	941	942	...	958	959	960

In this case, because the total number of scans is not an even multiple of the buffer size, the oldest scan is not at the beginning of the buffer and the last scan is not at the end of the buffer. **`daqAdcBufferRotate`** can rearrange the scans into their natural, chronological order:

Buffer Position	1	2	3	...	39	40	41	42	...	59	59	60
Scan	941	942	943	...	979	980	981	982	...	998	999	1000

If the total number of acquired scans is no greater than the buffer size, then the scans have not overwritten earlier scans and the buffer is already in chronological order. In this case, **`daqAdcBufferRotate`** does not modify the buffer.

Note: `daqAdcBufferRotate` only works on unpacked samples.

daqAdcCalcTrig

DLL Function	`daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);`
C	`daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);`
Visual BASIC	`VBdaqAdcCalcTrig&(ByVal handle&, ByVal bipolar&, ByVal gainVal!, ByVal voltageLevel!, triggerLevel%)`
Delphi	`daqAdcCalcTrig(handle:DaqHandleT; bipolar:longbool; gainVal:single; voltageLevel:single; var triggerLevel:DWORD)`
Parameters	
`handle`	Handle to the device for which the trigger level is to be calculated
`bipolar`	A flag that should be non-zero if the trigger channel is bipolar, or zero if it is unipolar
`gainVal`	A gain value of the trigger channel
`voltageLevel`	Voltage level to trigger at.
`triggerLevel`	Returned count to program the trigger using the daqAdcSetTrig function
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetTrig`
Program References	None
Used With	All devices

`daqAdcCalcTrig` calculates the trigger level and source for an analog trigger. The result of **`daqAdcCalcTrig`** is the **`triggerLevel`** parameter. The **`triggerLevel`** parameter can then be passed to the **`daqAdcSetTrig`** function to configure the analog trigger.

The **`triggerLevel`** parameter is calculated from: the unipolar/bipolar and gain settings of the trigger channel, the desired analog voltage setpoint and trigger polarity, and the external reference voltage of D/A channel 1. The trigger channel is automatically the first channel in the current A/D scan group for DaqBooks and DaqBoards.

The **`bipolar`** parameter should be set according to the current bipolar/unipolar setting of the trigger channel. This parameter is jumper-selectable when using a DaqBook/100/112 and DaqBoard/100A/112A and software-programmable when using the DaqBook/200/200A.

The **`gainVal`** parameter sent to the **`daqAdcCalcTrig`** should be the actual gain of the trigger channel, not the gain definition used by the rest of the Daq* A/D functions. For example, if the trigger channel uses the gain definition **`DgainX8`**, the gain parameter of **`daqAdcCalcTrig`** should be 8.

The **`voltageLevel`** defines the analog voltage at which the Daq* will trigger. The setpoint must be within the valid input range of the trigger channel. For example, the setpoint range for a bipolar channel with unity gain would be 0 to 10 V (for ×8 gain, the range would be 0 to 1.25 V) for a DaqBook or a DaqBoard. **Note**: When using the Daq PCMCIA, the **`bipolar`** parameter is ignored.

daqAdcDisarm

DLL Function	`daqAdcDisarm(DaqHandleT handle);`
C	`daqAdcDisarm(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcDisarm&(ByVal handle&)`
Delphi	`daqAdcDisarm(handle:DaqHandleT)`
Parameters	
`handle`	handle to the device to disable ADC acquisitions
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcArm`
Program References	None
Used With	All devices

`daqAdcDisarm` allows you to disarm an ADC acquisition if one is currently active.

- If the specified trigger event has not yet occurred, the trigger event will be disabled and no ADC acquisition will be performed.
- If the trigger event has occurred, the acquisition will be halted and the data transfer stopped and no more ADC data will be collected.

daqAdcExpSetBank

DLL Function	`daqAdcExpSetBank(DaqHandleT handle, DWORD chan, DaqAdcExpType bankType);`
C	`daqAdcExpSetBank(DaqHandleT handle, DWORD chan, DaqAdcExpType bankType);`
Visual BASIC	`VBdaqAdcExpSetBank&(ByVal handle&, ByVal chan&, ByVal bankType&)`
Delphi	`daqAdcExpSetBank(handle:DaqHandleT; chan:DWORD; bankType:DaqAdcExpType)`
Parameters	
`handle`	Handle to the device for which to set the expansion bank
`chan`	Channel number on the DBK card. Channel numbers are in groups of 16 channels per bank.
`bankType`	Type of channel bank.
Returns	`DerrInvChan` - Invalid Channel Number (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcExpSetChanOption, daqAdcExpSetModuleOption`
Program References	None
Used With	All devices

`daqAdcExpSetBank` internally programs intelligent DBK card channels so the Daq* gains may be set just before the acquisition. A bank consists of 16 channels, but **`daqAdcExpSetBank`** must be called once for each card in the bank. For example, if four 4-channel cards (such as a DBK7) are used in the first expansion bank, you must call **`daqAdcExpSetBank`** 4 times with channels 16, 20, 24, and 28. With only one such card, you cannot fill the remainder of the bank with another type of device. See the *DBK Card Definition* table for **`bankType`** settings.

daqAdcExpSetChanOption

DLL Function	`daqAdcExpSetChanOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
C	`daqAdcExpSetChanOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
Visual BASIC	`VBdaqAdcExpSetChanOption&(ByVal handle&, ByVal chan&, ByVal optionType&, ByVal optionValue!)`
Delphi	`daqAdcExpSetChanOption(handle:DaqHandleT; chan:DWORD; const optionType:DaqChanOptionType; optionValue:single)`
Parameters	
`handle`	Handle to the device for which to set the channel option
`chan`	The number of the channel to be configured.
`optionType`	The configurable option to be set (see table DBK Card Definitions)
`optionValue`	The configurable option to be set (see table DBK Card Definitions)
Returns	`DerrNoError` - No Errors (also, refer to *API Error Codes* on page 3-83) `DerrInvChan` - Invalid Channel Number
See Also	`daqAdcExpSetModuleOption`
Program References	None
Used With	All devices

`daqAdcExpSetChanOption` allows you to configure channel parameters for DBK modules with software-configurable settings on a per channel basis. See the *DBK Card Definition* table for **`optionType`** and **`optionValue`** settings.

daqAdcExpSetModuleOption

DLL Function	`daqAdcExpSetModuleOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
C	`daqAdcExpSetModuleOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
Visual BASIC	`VBdaqAdcExpSetModuleOption&(ByVal handle&, ByVal chan&, ByVal optionType&, ByVal optionValue!)`
Delphi	`daqAdcExpSetModuleOption(handle:DaqHandleT; chan:DWORD; const optionType:DaqChanOptionType; optionValue:single)`
Parameters	
`handle`	Handle to the device for which to set the module option.
`chan`	Any channel on the module (expansion chassis) to be configured.
`optionType`	The configurable option to be set (see table *DBK Card Definitions*).
`optionValue`	The configurable option to be set (see table *DBK Card Definitions*).
Returns	An error number, or 0 if no error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcExpSetChannelOption`
Program References	None
Used With	All devices

`daqAdcExpSetModuleOption` allows you to configure parameters that apply to the whole module (for DBK modules with software-configurable settings) on a per expansion module basis. See the *DBK Card Definition* table for **`optionType`** and **`optionValue`** settings.

daqAdcGetFreq

DLL Function	`daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);`
C	`daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);`
Visual BASIC	`VBdaqAdcGetFreq&(ByVal handle&, freq!)`
Delphi	`daqAdcGetFreq(handle:DaqHandleT; var freq:single)`
Parameters	
`handle`	Handle to the device for which to get the current frequency setting
`freq`	A variable to hold the currently defined sampling frequency in Hz Valid values: `100000.0 - 0.0002`
Returns	`DerrNoError` - No errors (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetFreq, daqAdcSetClock`
Program References	None
Used With	All devices

`daqAdcGetFreq` reads the sampling frequency of the pacer clock.

Note: **`daqAdcSetFreq`** assumes that the 1 MHz/10 MHz jumper is set to the default position of 1 MHz.

daqAdcGetScan

DLL Function	`daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);`
C	`daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);`
Visual BASIC	`VBdaqAdcGetScan&(ByVal handle&, channels&(), gains&(), flags&(), chanCount&)`
Delphi	`daqAdcGetScan( handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:PDWORD )`
Parameters	
`handle`	Handle to the device for which to get the current scan configuration.
`channels`	An array to hold up to 512 channel numbers or 0 if the channel information is not desired.
`*gains`	An array to hold up to 512 gain values or 0 if the channel gain information is not desired
`flags`	Channel configuration flags in the in the form of a bit mask
`chanCount`	A variable to hold the number of values returned in the chans and gains arrays
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetScan, daqAdcSetMux`
Program References	None
Used With	All devices

`daqAdcGetScan` reads the current scan group consisting of all channels currently configured. The returned parameter settings directly correspond to those set using the daqAdcSetScan function. For further description of these parameters, refer to **`daqAdcSetScan`**. See *ADC Flags Definition* table for channel flag definitions.

daqAdcRd

DLL Function	`daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRd&(ByVal handle&, ByVal chan&, sample%, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRd(handle:DaqHandleT; chan:DWORD; var sample:WORD; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC reading is to be acquired
`chan`	A single channel number
`sample`	A pointer to a value where an A/D sample is stored. Valid values: (See **`daqAdcSetTag`**)
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrFIFOFull` - Buffer Overrun `DerrInvGain` - Invalid gain `DerrInvChan` - Invalid channel `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetMux, daqAdcSetTrig, daqAdcSoftTrig`
Program References	DACEX.PAS (Delphi)
Used With	All devices

`daqAdcRd` is used to take a single reading from the given local A/D channel. This function will use a software trigger to immediately trigger and acquire one sample from the specified A/D channel.

- The **`chan`** parameter indicates the channel for which to take the sample.
- The **`sample`** parameter is a pointer to where the collected sample should be stored.
- The **`gain`** parameter indicates the channel's gain setting.
- The **`flags`** parameter allows the setting of channel-dependent options. See *ADC Flags Definition* table for channel **`flags`** definitions.

daqAdcRdN

DLL Function	`daqAdcRdN(DaqHandleT handle, DWORD chan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdN(DaqHandleT handle, DWORD chan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdN&(ByVal handle&, ByVal chan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdN(handle:DaqHandleT; chan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC channel samples are to be acquired
`chan`	A single channel number
`buf`	An array where the A/D scans will be returned
`scanCount`	The number of scans to be taken Valid values: `1 - 32767`
`triggerSource`	The trigger source
`rising`	Boolean flag to indicate the rising or falling edge for the trigger source
`level`	The trigger level if an analog trigger is specified Valid values: `0 -4095`
`freq`	The sampling frequency in Hz (`100000.0 to 0.0002`)
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrFIFOFull` - Buffer overrun `DerrInvGain` -Invalid gain `DerrIncChan` - Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid level (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetFreq, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig`
Program References	None
Used With	All devices

`daqAdcRdN` is used to take multiple scans from a single A/D channel. This function will:

- Configure the pacer clock
- Configure all channels with the specified **`gain`** parameter
- Configure all channel options with the channel **`flags`** specified
- Arm the trigger
- Acquire **`count`** scans from the specified A/D channel

See *ADC Flags Definition* table (in *ADC Miscellaneous Definitions*) for channel **`flags`** parameter definition.

daqAdcRdScan

DLL Function	`daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device from which the ADC scan is to be acquired
`startChan`	The starting channel of the scan group
`endChan`	The ending channel of the scan group
`buf`	An array where the A/D scans will be placed
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask.
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcRdNScan, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig`
Program References	DACEX.PAS (Delphi)
Used With	All devices

`daqAdcRdScan` reads a single sample from multiple channels. This function will use a software trigger to immediately trigger and acquire 1 scan consisting of each channel, starting with **`startChan`** and ending with **`endChan`**. The **`gain`** setting will be applied to all channels. See *ADC Flags Definition* table for channel **`flags`** definitions.

daqAdcRdScanN

DLL Function	`daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device from which ADC scans are to be acquired
`startchan`	The starting channel of the scan group (see table at end of chapter)
`endchan`	The ending channel of the scan group (see table at end of chapter)
`buf`	An array where the A/D scans will be placed
`scanCount`	The number of scans to be read Valid values: `1 - 65536`
`triggerSource`	The trigger source (see table at end of chapter)
`rising`	Boolean flag to indicate the rising or falling edge for the trigger source
`level`	The trigger level if an analog trigger is specified Valid values: `0 -4095`
`freq`	The sampling frequency in Hz Valid values: `100000.0 - 0.0002`
`gain`	The channel gain (See tables at end of chapter).
`flags`	Channel configuration flags in the form of a bit mask.
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid Level `DerrFIFOFull` -Buffer Overrun `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcRd, daqAdcRdN, daqAdcRdScan, daqAdcSetClock, daqAdcSetTrig`
Program References	None
Used With	All devices

`daqAdcRdScanN` reads multiple scans from multiple A/D channels. This function will configure the pacer clock, arm the trigger and acquire count scans consisting of each channel, starting with **`startChan`** and ending with **`endChan`**. The **`gain`** setting will be applied to all channels. The **`freq`** parameter is used to set the acquisition frequency. See *ADC Flags Definition* table for channel **`flags`** parameter definition.

daqAdcSetAcq

DLL Function	daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);
C	daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);
Visual BASIC	VBdaqAdcSetAcq&(ByVal handle&, ByVal mode&, ByVal preTrigCount&, ByVal postTrigCount&)
Delphi	daqAdcSetAcq(handle:DaqHandleT; mode:DaqAdcAcqMode; preTrigCount:DWORD; postTrigCount:DWORD)
Parameters	
handle	Handle to the device for which the ADC acquisition is to be configured
mode	Selects the mode of the acquisition
preTrigCount	Number of pre-trigger ADC scans to be collected
postTrigCount	Number of post-trigger ADC scans to be collected
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqAdcArm, daqAdcDisarm, daqAdcSetTrig
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

daqAdcSetAcq allows you to characterize the acquisition mode and the pre- and post-trigger durations. The **mode** parameter describes the style of data collection. The **preTrigCount** and **postTrigCount** parameters specify the respective durations, or lengths, of the pre-trigger and post-trigger acquisition states.

Acquisition modes can be defined as follows:

- **DaamNShot** - Once triggered, continue acquisition until the specified post-trigger count has been satisfied. Once the post-trigger count has been satisfied, the acquisition will be automatically disarmed.
- **DaamNShotRearm** - Once triggered, continue the acquisition for the specified post-trigger count, then re-arm the acquisition with the same acquisition configuration parameters as before. The automatic re-arming of the acquisition may be disabled at any time by issuing a **daqAdcDisarm.**
- **DaamInfinitePost** - Once triggered, continue the acquisition indefinitely until the acquisition is disabled by the **daqAdcDisarm** function.
- **DaamPrePost** - Begin collecting the specified number of pre-trigger scans immediately upon issuance of the **daqAdcArm** function. The trigger will not be enabled until the specified number of pre-trigger scans have been collected. Once triggered, the acquisition will then continue collecting post-trigger data until the post-trigger count has been satisfied. Once the post-trigger count has been satisfied, the acquisition will be automatically disarmed.

daqAdcSetClockSource

DLL Function	daqAdcSetClockSource(DaqHandleT handle, DaqAdcClockSource clockSource);
C	daqAdcSetClockSource(DaqHandleT handle, DaqAdcClockSource clockSource);
Visual BASIC	VBdaqAdcSetClockSource&(ByVal handle&, ByVal clockSource&)
Delphi	daqAdcSetClockSource(handle:DaqHandleT; clockSource:DaqAdcClockSource)
Parameters	
handle	Handle to the device for which to set the ADC clock source.
clockSource	Specifies the clock source for ADC acquisitions
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqAdcSetFreq
Program References	None
Used With	All devices

daqAdcSetClockSource allows you to set up the clock source to be used to drive the ADC acquisition frequency.

daqAdcSetDataFormat

DLL Function	daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);
C	daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);
Visual BASIC	VBdaqAdcSetDataFormat &(ByVal handle&, ByVal rawFormat&, ByVal postProcFormat&)
Delphi	daqAdcSetDataFormat(Handle:DaqHandleT; rawFormat:DaqAdcRawDataFormatT rawFormat; postProcFormat:DaqAdcPostProcDataFormatT);
Parameters	
handle	The handle to the device for which to set the option
rawFormat	The channel number on the device for which the option is to be set
postProcFormat	Flags specifying the options to use
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqCvtRawDataFormat,daqCvtRawDataFormat
Program References	None
Used With	All devices

daqAdcSetDataFormat allows the setting of the raw and the post-acquisition data formats which will be returned by the acquisition transfer functions. **Note**: Certain devices may be limited to the types of raw and post-acquisition data formats which can be presented.

The **rawFormat** parameter indicates how the raw data format is to be presented. Normally, the raw-data format represents the data from the A/D converter. The default value for this parameter is **DardfNative** where the raw-data format follows the native-data format of the A/D for the particular device. An optional parameter is **DardfPacked** where raw A/D values are compressed to make full use of all unused bits for any native format that leaves unused bits in the byte-aligned count value. For instance, a 12-bit raw A/D value (which would normally be represented in a 16-bit word, 2-byte count value) will be compressed so that 4 12-bit A/D raw counts can be represented in 3 16-bit word count values. Currently, only the WaveBook/512 supports this packed format (used with the generic functions of the form **daqAdcTransfer)**.

The **postProcFormat** parameter specifies the format for which post-acquisition data will be presented. This format is used by the one-step functions of the form **daqAdcRd** . The default value is **DappdfRaw** where the post-acquisition data format will follow the **rawFormat** parameter.

daqAdcSetDiskFile

DLL Function	`daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);`
C	`daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);`
Visual BASIC	`VBdaqAdcSetDiskFile&(ByVal handle&, ByVal filename$, ByVal openMode&, ByVal preWrite&)`
Delphi	`daqAdcSetDiskFile(handle:DaqHandleT; filename:PChar; openMode:DaqAdcOpenMode; preWrite:DWORD)`
Parameters	
`handle`	Handle to the device for which direct to disk ADC acquisition is to be performed.
`filename`	String representing the path and name of the file to place the raw ADC acquisition data.
`openMode`	Specifies how to open the file for writing
`preWrite`	Specifies the amount to pre-write(in bytes) the file
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferGetStat, daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqAdcSetDiskFile` allows you to set a destination file for ADC data transfers. ADC data transfers will be directed to the specified disk file. The **`filename`** parameter is a string representing the path\name of the file to be opened. The **`openMode`** parameter indicates how the file is to be opened for writing data. Valid file open modes are defined as follows:

- **`DaomAppendFile`** - Open an existing file to append subsequent ADC transfers. This mode should only be used when the existing file has a similar ADC channel group configuration as the subsequent transfers.
- **`DoamWriteFile`** - Rewrite or write over an existing file. This operation will destroy the original contents of the file.
- **`DoamCreateFile`**- Create a new file for subsequent ADC transfers. This mode does not require that the file exist beforehand.

The **`preWrite`** parameter may, optionally, be used to specify the amount that the file is to be pre-written before the actual data collection begins. Specifying the pre-write amount may increase the data-to-disk performance of the acquisition if it is known beforehand how much data will be collected. If no pre-write is to be done, then the **`preWrite`** parameter should be set to **`0`**.

daqAdcSetFreq

DLL Function	`daqAdcSetFreq(DaqHandleT handle, FLOAT freq);`
C	`daqAdcSetFreq(DaqHandleT handle, FLOAT freq);`
Visual BASIC	`VBdaqAdcSetFreq&(ByVal handle&, ByVal freq!)`
Delphi	`daqAdcSetFreq(handle:DaqHandleT; freq:single)`
Parameters	
`handle`	Handle to the device for which the ADC acquisition frequency is to be set.
`freq`	The sampling frequency in Hz Valid values: **`100000.0 - 0.0002`**
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcGetFreq, daqAdcSetClockSource`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqAdcSetFreq` calculates and sets the frequency of the pacer clock using the frequency specified in Hz. The frequency is converted to two counter values that control the frequency of the pacer clock (in this conversion, some resolution of the frequency may be lost). **`daqAdcRdFreq`** can be used to read the exact frequency setting of the pacer clock. **`daqAdcSetClock`** can be used to explicitly set the two counter values of the pacer clock. The pacer clock can be used to control the sampling rate of the A/D converter.

daqAdcSetMux

DLL Function	`daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);`
C	`daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcSetMux&(ByVal handle&, ByVal startChan&, ByVal endChan&, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcSetMux(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which to configure the ADC channel scan group
`startChan`	The starting channel of the scan group
`endChan`	The ending channel of the scan group
`gain`	The gain value for all channels
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrInvGain` - Invalid gain `DerrIncChan` - Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetScan, daqAdcGetScan`
Program References	DACEX1.C, DAQEX.FRM (VB)
Used With	All devices

`daqAdcSetMux` sets a simple scan sequence of local A/D channels from **`startChan`** to **`endChan`** with the specified **`gain`** value. This command provides a simple alternative to **`daqAdcSetScan`** if only consecutive channels need to be acquired. The **`flags`** parameter is used to set channel dependent options. See *ADC Flags Definition* table for channel **`flags`** definitions.

daqAdcSetRate

DLL Function	`daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);`
C	`daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);`
Visual BASIC	`VBdaqAdcSetRate(ByVal handle&, ByVal mode&, ByVal acqState&, ByVal reqRate!, actualRate!);`
Delphi	`daqAdcSetRate(handle: DaqHandleT; mode: DaqAdcRateMode, acqState: DaqAdcAcqState; reqRate:FLOAT; actualRate:PFLOAT);`
Parameters	
`handle`	Handle to the device for which to set ADC scanning frequency.
`mode`	Specifies the rate mode (frequency or period).
`acqState`	Specifies the acquisition state to which the rate is to be applied.
`reqRate`	Specifies the requested rate.
`actualRate`	Returns the actual rate applied. This may be different from the requested rate.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetAcq, daqAdcSetTrig, daqAdcArm, daqAdcSetFreq, daqAdcGetFreq`
Program References	
Used With	All devices

`daqAdcSetRate` configures the ADC scan rate using the rate mode specified by the **`mode`** parameter. Currently, the valid modes are:

- **`DarmPeriod`** - Defines the requested rate to be in periods/sec.
- **`DarmFrequency`** - Defines the requested rate to be a frequency.

This function will set the ADC acquisition rate requested by the **`reqRate`** parameter for the acquisition state specified by the **`acqState`** parameter. Currently, the following acquisition states are valid:

- **`DaasPreTrig`** - Sets the pre-trigger ADC acquisition rate to the requested rate.
- **`DaasPostTrig`** - Sets the post-trigger ADC acquisition rate to the requested rate.

If the requested rate is unattainable on the specified device, a rate will be automatically adjusted to the device's closest attainable rate. If this occurs, the **`actualRate`** parameter will return the actual rate for which the device has been programmed.

daqAdcSetScan

DLL Function	daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);
C	daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);
Visual BASIC	VBdaqAdcSetScan&(ByVal handle&, channels&(), gains&(), flags&(), ByVal chanCount&)
Delphi	daqAdcSetScan(handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:DWORD)
Parameters	
handle	Handle to the device for which ADC scan group is to be configured
channels	An array of up to 512 channel numbers
*gains	An array of up to 512 gain values
flags	Channel configuration flags in the form of a bit mask
chanCount	The number of values in the chans and gains arrays Valid values: 1 -512
Returns	DerrNotCapable - No high speed digital DerrInvGain - Invalid gain DerrInvChan - Invalid channel DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqAdcGetScan, daqAdcSetMux
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

DaqAdcSetScan configures an A/D scan group consisting of multiple channels. As many as 512 channel entries can be made in the A/D scan group configuration. Any analog input channel can be included in the scan group configuration at any valid gain setting. Scan group configuration may be composed of local or expansion channels and (for the DaqBook/DaqBoard) the high-speed digital I/O port.

The **channels** parameter is a pointer to an array of up to 512 channel values. Each entry represents a channel number in the scan group configuration. Channels can be entered multiple times at the same or different gain setting.

The **gains** parameter is a pointer to an array of up to 512 gain settings. Each gain entry represents the gain to be used with the corresponding channel entry. Gain entry can be any valid gain setting for the corresponding channel.

The **flags** parameter is a pointer to an array of up to 512 channel flag settings. Each flag entry represents a 4-byte-wide bit map of channel configuration settings for the corresponding channel entry. The channel flags can be used to set channel specific configuration settings (such as polarity). See the *ADC Flags Definition* table for valid channel flag values.

The **chanCount** parameter represents the total number of channels in the scan group configuration. This number also represents the number of entries in each of the **channels, gains** and **flags** arrays.

daqAdcSetTrig

DLL Function	`daqAdcSetTrig(DaqHandleT handle, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, WORD hysteresis, DWORD channel);`
C	`daqAdcSetTrig(DaqHandleT handle, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, WORD hysteresis, DWORD channel);`
Visual BASIC	`VBdaqAdcSetTrig&(ByVal handle&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal hysteresis%, ByVal channel&)`
Delphi	`daqAdcSetTrig(handle:DaqHandleT; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; hysteresis:WORD; channel:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC acquisition trigger is to be configured.
`triggerSource`	Sets the trigger source.
`rising`	Boolean flag to indicate the rising or falling edge for the trigger source
`level`	The trigger level (in A/D counts) for an analog level trigger
`hysteresis`	hysteresis value for analog level trigger (if selected)
`channel`	Channel for which the analog level trigger(if selected) is to be detected.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetAcq`
Program References	ADCEX1.C, DACEX1.C, DAQEX.FRM (VB), ADCEX.PAS, ERREX.PAS (Delphi)
Used With	All devices

`daqAdcSetTrig` sets and arms the trigger of the A/D converter. Several trigger sources and several mode flags can be used for a variety of acquisitions. **`daqAdcSetTrig`** will stop current acquisitions, empty acquired data, and arm the Daq* using the specified trigger.

Trigger detection for the given trigger source will not begin until the acquisition has been armed with the **`daqAdcArm`** function. Trigger sources may be defined as follows:

- **`DatsImmediate`** - Trigger the acquisition immediately upon issuance of the **`daqAdcArm`** function. This trigger mode is used to begin collecting data immediately upon configuration of the acquisition.
- **`DatsSoftware`** - Trigger the acquisition upon issuance of the daqAdcSoftTrig function. This trigger mode can be used to initiate a trigger upon some form of user or application program input.
- **`DatsAdcClock`** - Trigger the acquisition upon ADC pacer clock input. This trigger mode can be used to synchronize the trigger event with the ADC pacer clock.
- **`DatsExternalTTL`** - Trigger the acquisition upon sensing a rising or falling (depending on state of **`rising`** flag) signal on an external TTL input signal (trig0 - pin 25 on P1).
- **`DatsHardwareAnalog`** - Trigger upon detection of a rising or falling (depending on the state of the **`rising`** flag) analog signal (whose count is defined by the **`level`** parameter). This trigger mode is detected in hardware to allow generally faster acquisition frequencies than the **`DatsSoftwareAnalog`** trigger source. However, use of this mode is restricted to channel level triggering on only the first channel within the channel scan (defined by the channel parameter). **Note**: This mode is not available on Daq PCMCIA product lines.
- **`DatsSoftwareAnalog`** - Trigger upon detection of a rising or falling (depending on the state of the **`rising`** flag) analog signal (whose count is defined by the **`level`** parameter). This trigger mode is detected in software and generally will not allow the acquisition speeds of the **`DatsHardwareAnalog`** trigger source. However, this mode has no trigger channel restrictions. Any valid channel in the scan group can be configured as the trigger channel by specifying it in the **`channel`** parameter.

Note: The **`level`** parameter is only used for the analog trigger modes. **`level`** is a count representing the A/D count level trigger threshold to be passed through in order to satisfy the analog trigger event. A number of factors are used to determine its proper value. For help in calculating this analog count level properly, see the **`daqAdcCalcTrig`** function.

daqAdcSetTrigEnhanced

DLL Function	`daqAdcSetTrigEnhanced(DaqHandleT handle, DaqAdcTriggerSource *triggerSources, PDWORD gains, PDWORD adcRanges, DaqEnhTrigDef trigDef, PFLOAT levels, PFLOAT hysteresis, PDWORD channels,DWORD chanCount, char *opStr);`
C	`daqAdcSetTrigEnhanced(DaqHandleT handle, DaqAdcTriggerSource *triggerSources, PDWORD gains, PDWORD adcRanges, DaqEnhTrigDef trigSense, PFLOAT levels, PFLOAT hysteresis, PDWORD channels,DWORD chanCount, char *opStr);`
Visual BASIC	`VBdaqAdcSetTrigEnhanced&(ByVal handle&, triggerSources&, gains&, adcRanges&, trigSense&, levels!, hysteresis!, channels&, chanCount&, opStr$)`
Delphi	`daqAdcSetTrigEnahanced(handle:DaqHandleT; triggerSources:DaqAdcTriggerSource; gains: PDWORD; adcRanges: PDWORD; trigSense:DaqEnhTrigDef; levels : PFLOAT; hysteresis : PFLOAT; channels:PDWORD; chanCount:DWORD; opStr: String )`
Parameters	
`handle`	Handle to the device for which the ADC acquisition trigger is to be configured.
`triggerSource`	A pointer to an array of trigger sources for each defined trigger channel.
`gains`	A pointer to an array of gains for each defined A/D trigger channel.
`levels`	A pointer to an array of A/D analog trigger levels for each defined A/D trigger channel.
`hysteresis`	A pointer to an array of hysteresis values for each defined A/D trigger channel.
`trigSense`	A pointer to an array of trigger sensitivity flags for each defined A/D channel trigger source.
`adcRanges`	A pointer to an array of polarity flag definitions for each defined A/D channel.
`channels`	A pointer to an array of trigger channels representing the actual A/D trigger channels to trigger on.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetAcq, daqAdcSetTrig, daqAdcSetScan`
Program References	
Used With	WaveBook/512, WaveBook/516

`daqAdcSetTrigEnhanced` configures the device for enhanced triggering. Enhanced trigger configuration allows the device to be configured to detect A/D triggering formed with multiple A/D channel trigger-event conditions. The enhanced trigger event may be defined as a combination of multiple A/D analog-level event conditions that are logically **and**'d or **or**'d.

The trigger event is formulated based on the channel trigger event for each channel in the trigger sequence. The total number of trigger channels is defined by the **`chanCount`** parameter. Each channel trigger configuration parameter definition is a pointer to an array of **`chanCount`** length and is defined as follows:

- **`channels`** - Defines a pointer to an array of actual A/D channel numbers for which to configure the corresponding trigger events.
- **`triggerSources`** - Defines a pointer to an array of trigger sources for which to configure the corresponding A/D trigger events for the corresponding channel in the channels array. See the *ADC Trigger Source Definitions* table for valid triggers.
- **`gains`** - Defines a pointer to an array of gains corresponding to the actual A/D channels in the corresponding A/D channel number in the channels array.
- **`adcRanges`** - Defines a pointer to an array of A/D ranges for the A/D channels defined in the corresponding channels array.
- **`hysteresis`** - Defines a pointer to an array of hysteresis values for each corresponding A/D channel defined in the channels array.
- **`levels`** - Defines a pointer to an array of A/D levels for which, when satisfied, will set the trigger event for the corresponding channel defined in the channels array.
- **opStr** - Defines a string that defines the logical relationship between the individual channel trigger events and the global A/D trigger condition. Currently, the string can be defined as "*" to perform an **and** operation or "+" to perform an **or** operation on the individual channel trigger events to formulate the global A/D trigger condition.
- **`trigSense`** - Defines an array of trigger sensitivity definitions for satisfying the defined trigger event for the corresponding channel defined in the channels array. Currently, the valid trigger sensitivity values are as follows:

`DatdRisingEdge`	Trigger the channel on the rising edge of the signal at the specified level.
`DatdFallingEdge`	Trigger the channel on the falling edge of the signal at the specified level.
`DatdAboveLevel`	Trigger the channel when the signal is above the specified level.
`DatdBelowLowel`	Trigger the channel when the signal is below the specified level.
`DatdRisingEdgeLatched`	Trigger the channel on the rising edge of the signal at the specified level and latch the channel trigger event.

`DatdFallingEdgeLatched`	Trigger the channel on the falling edge of the signal at the specified level and latch the channel trigger event.
`DatdAboveLevelLatched`	Trigger the channel when the signal is above at the specified level and latch the channel trigger event.
`DatdBelowLevelLatched`	Trigger the channel when the signal is below at the specified level and latch the channel trigger event.
Note: The **Latched** trigger sensitivities indicate the device will maintain the trigger event for the given channel regardless of subsequent states of the input signal. After the channel has triggered, it will remain in a triggered state while the current acquisition is active. The **non-latched** trigger sensitivities will only indicate a channel trigger event while the input signal for the given channel is in the triggered state.	

daqAdcSoftTrig

DLL Function	`daqAdcSoftTrig(DaqHandleT handle);`
C	`daqAdcSoftTrig(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcSoftTrig&(ByVal handle&)`
Delphi	`daqAdcSoftTrig(handle:DaqHandleT)`
Parameters	
`handle`	Handle to the device to which the ADC software trigger is to be applied
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcSetTrig, daqAdcSetAcq`
Program References	None
Used With	All devices

`daqAdcSoftTrig` is used to send a software trigger command to the Daq* device. This software trigger can be used to initiate a scan or an acquisition from a program after configuring the software trigger as the trigger source. This function may only be used if the trigger source for the acquisition has been set to **`DatsSoftware`** with the **`daqAdcSetTrig`** function.

daqAdcTransferBufData

DLL Function	`daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);`
C	`daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);`
Visual BASIC	`VBdaqAdcTransferBufData(ByVal handle, buf%, ByVal scanCount&, ByVal bufMask&, retCount&);`
Delphi	`daqAdcTransferBufData(handle: DaqHandleT; buf : PWORD, scanCount : DWORD, bufMask: DaqAdcBufferXferMask; retCount: PDWORD);`
Parameters	
`handle`	Handle to the device for which the ADC buffer should be retrieved.
`buf`	Pointer to an application-supplied buffer to place the buffered data.
`scanCount`	Number of scans to retrieve from the acquisition buffer.
`bufMask`	A mask defining operation depending on the current state of the acquisition buffer
`retCount`	A pointer to the total number of scans returned, if any.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferSetBuffer, daqAdcTransferGetStat`
Program References	
Used With	All devices

`daqAdcTransferBufData` requests a transfer of **`scanCount`** scans from the driver-allocated ADC acquisition buffer to the specified user-supplied buffer. The **`bufMask`** parameter can be used to specify the conditions for the transfer as follows:

- **`DabtmWait`** - Instructs the function to wait until the requested number of scans are available in the driver-allocated acquisition buffer. When the requested number of scans are available, the function will return with **`retCount`** set to **`scanCount`**, the number of scans requested. ADC data will be returned in the memory referred to by the **`buf`** parameter.
- **`DabtmNoWait`** - Instructs the function to return immediately if the specified number of scans are not available when the function is called. If the entire amount requested is not available, the function will return with no data and **`retCount`** will be set to 0. If the requested number of scans are available in ADC acquisition buffer, the function will return with **`retCount`** set to **`scanCount`**, the number of scans requested. ADC data will be returned in the memory referred to by the **buf** parameter.
- **`DabtmRetAvail`** - Instructs the function to return immediately, regardless of the number of scans available in the driver-allocated acquisition buffer. The **`retCount`** parameter will return the total number of scans retrieved. **`retCount`** can return anything from 0 to **`scanCount`**, the number of scans requested. ADC data will be returned in the memory referred to by the **`buf`** parameter.

The driver-allocated acquisition buffer must have been allocated prior to calling this function. This is performed via the **`daqAdcTransferSetBuffer`**. Refer to **`daqAdcTransferSetBuffer`** for more details on specifying the driver-allocated acquisition buffer.

daqAdcTransferGetStat

DLL Function	`daqAdcTransferGetStat(DaqHandleT handle, PDWORD active, PDWORD retCount);`
C	`daqAdcTransferGetStat(DaqHandleT handle, PDWORD active, PDWORD retCount);`
Visual BASIC	`VBdaqAdcTransferGetStat&(ByVal handle&, active&, retCount&)`
Delphi	`daqAdcTransferGetStat( handle:DaqHandleT; var active:DWORD; var retCount:DWORD )`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`active`	A pointer to the transfer-state flags in the form of a bit mask
`retCount`	A pointer to the total number of ADC scans acquired (or available) in the current transfer
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqAdcTransferGetStat` allows you to retrieve the current state of an ADC acquisition transfer.

The **`active`** parameter will indicate the current state of the transfer in the form of a bit mask. Refer to the *ADC Acquisition/Transfer Active Flag Definitions* (in the *ADC Miscellaneous Definitions* table) for valid bit-mask states.

The **`retCount`** parameter will return the total number of scans acquired in the current transfer if the transfer is in user-allocated buffer mode or will return the total number of unread scans in the acquisition buffer if the transfer is in driver-allocated buffer mode. Refer to the **`daqAdcTransferSetBuffer`** function for more information on buffer allocation modes.

The transfer state and return count values will continue to be updated until any of the following occurs:

- the transfer count is satisfied
- the transfer is stopped (**`daqAdcStopTransfer`**)
- the acquisition is disarmed (**`daqDisarm`**)

daqAdcTransferSetBuffer

DLL Function	`DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);`
C	`DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);`
Visual BASIC	`VBdaqAdcTransferSetBufferAllocMem&(ByVal handle&, ByVal scanCount&, ByVal transferMask&)`
Delphi	`daqAdcTransferSetBufferAllocMem(handle:DaqHandleT; scanCount:DWORD; transferMask:DWORD)`
Parameters	
`handle`	Handle to the device for which an ADC transfer is to be performed.
`buf`	Pointer to the buffer for which the acquired data is to be placed.
`scanCount`	The total length of the buffer (in scans).
`transferMask`	Configures the buffer transfer mode.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferStart, daqAdcTransferStop, daqAdcTransferGetStat, daqAdcSetAcq, daqAdcTransferBufData`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqAdcTransferSetBuffer` allows you to configure transfer buffers for ADC data acquisition. This function can be used to configure the specified user- or driver-allocated buffers for subsequent ADC transfers.

If a user-allocated buffer is to be used, two conditions apply:

- The buffer specified by the **`buf`** parameter must have **already been allocated** by the user prior to calling this function.
- The allocated buffer must be **large enough to hold the number of ADC scans** as determined by the current ADC scan group configuration.

The **`scanCount`** parameter is the total length of the transfer buffer in scans. The scan size is determined by the current scan group configuration. Refer to the **`daqAdcSetScan`** and **`daqAdcSetMux`** functions for further information on scan group configuration.

The character of the transfer can be configured via the **`transferMask`** parameter. Among other things, the **`transferMask`** specifies the update, layout/usage, and allocation modes of the buffer. The modes can be set as follows:

- **`DatmCycleOn`** - Specifies the buffer to be a circular buffer in buffer-cycle mode; allows the transfer to continue when the end of the transfer buffer is reached by wrapping the transfer of ADC data back to the beginning of the buffer. In this mode, the ADC transfer buffer will continue to be wrapped until the post-trigger count has been reached (specified by **`daqAdcSetAcq`**) or the transfer/acquisition is halted by the application (**`daqAdcTransferStop`**, **`daqAdcDisarm`**). The default setting is **`DatmCycleOff`**.
- **`DatmUpdateSingle`** - Specifies the update mode as single sample. The update mode can be set to update for every sample or for every block of ADC data. The update-on-single setting allows the ADC transfer buffer to be updated for each sample collected by the ADC. Compared to the block mode, this setting provides a higher degree of real-time transfer-buffer updating at the expense of slower aggregate-data throughput rates. The default setting is **`DatmUpdateBlock.`**
- **`DatmDriverBuf`** - Specifies that the driver allocate the ADC acquisition buffer as a circular buffer whose length is determined by the **`scanCount`** parameter with current scan group configuration. This option allows the driver to manage the circular acquisition buffer rather than placing the burden of buffer management on the user. This option should be used with the **`daqAdcTransferBufData`** to access the ADC acquisition buffer. The **`daqAdcTransferStop`** or the **`daqAdcDisarm`** function will stop the current transfer and de-allocate the driver-supplied ADC acquisition buffer. The default setting is **`DatmUserBuf.`** The **`DatmUserBuf`** option specifies a user-allocated ADC acquisition buffer. Here, buffer management must be done in user code. This option should be used with the **`daqAdcTransferStart`** function to perform the ADC data transfer operation.

daqAdcTransferStart

DLL Function	`daqAdcTransferStart(DaqHandleT handle);`
C	`daqAdcTransferStart(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcTransferStart&(ByVal handle&)`
Delphi	`daqAdcTransferStart( handle:DaqHandleT )`
Parameters	
`handle`	Handle to the device to initiate an ADC transfer
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTranferSetBuffer, daqAdcTransferGetStat, daqAdcTransferStop`
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqAdcTransferStart` allows you to initiate an ADC acquisition transfer. The transfer will be performed under the current active acquisition. If no acquisition is currently active, the transfer will not initiate until an acquisition becomes active (via the **`daqAdcArm`** function). The transfer will be characterized by the current settings for the transfer buffer. The transfer buffer can be configured via the **`daqAdcSetTransferBuffer`** function.

daqAdcTransferStop

DLL Function	`daqAdcTransferStop(DaqHandleT handle);`
C	`daqAdcTransferStop(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcTransferStop&(ByVal handle&)`
Delphi	`daqAdcTransferStop( handle:DaqHandleT )`
Parameters	
`handle`	Handle to the device for which the Adc data transfer is to be stopped
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferGetStat`
Program References	None
Used With	All devices

`daqAdcTransferStop` allows you to stop a current ADC buffer transfer, if one is active. The current transfer will be halted and no more data will transfer into the transfer buffer. Though the transfer is stopped, the acquisition will remain active. Transfers can be re-initiated with **`daqAdcStartTransfer`** after the stop, as long as the current acquisition remains active. The acquisition can be halted by calling the **`daqAdcDisarm`** function.

daqCalGetConstants

DLL Function	`daqCalGetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, PWORD gainConstant, PSHORT offsetConstant );`
C	`daqCalGetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, PWORD gainConstant, PSHORT offsetConstant );`
Visual BASIC	`VBdaqCalGetConstants(ByVal handle&, ByVal channel&, ByVal gain&, ByVal range&, al gainConstant%, offsetConstant% );`
Delphi	`daqCalGetConstants(handle: DaqHandleT;channel:DWORD; gain: DaqAdcGain; range: DaqAdcRangeT; gainConstant:PWORD; offsetConstant:PSHORT );`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`channel`	Channel number to apply the calibration settings
`gain`	Gain range to apply the calibration settings
`range`	A/D input range to apply the calibration settings
`gain`	Pointer to the gain value for the current entry
`offset`	Pointer to the offset value for the current entry
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCalSetConstants, daqCalSelectCalTable, daqCalSelectInputSignal,daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

`daqCalGetConstants` gets the calibration constants from the currently selected calibration table chosen by the **`daqCalSetConstants`** command.

The user-calibration constants are gains and offsets that are applied to the input data. The data comes in, is multiplied by the gain, then the offset is added to it. The resulting data is the conversion between the raw A/D data and the data that is presented during the acquisition. Each channel, gain, and bipolar/unipolar setting has a different pair of gain and offset values. The first three parameters of the **`daqCalGetConstants`** function specify which set of constants are to be retrieved. The last two parameters are the actual constants. These constants are in a particular binary format. The gain constant is 32768 times the gain. For a gain of ×1, the gain constant is 32768 or 0x8000. The maximum gain is approximately ×2 (65535/32768), and the minimum gain is ×0 (0/32768). The offset (a left-justified signed 12-bit number) is added to the final result. A single least-significant bit has an integer value of 16 or 0x0010.

daqCalSaveConstants

DLL Function	`daqCalSaveConstants(DaqHandleT handle, DWORD channel );`
C	`daqCalSaveConstants(DaqHandleT handle, DWORD channel );`
Visual BASIC	`VBdaqCalSaveConstants(ByVal handle&, ByVal channel& )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; channel: DWORD)`
Parameters	
`handle`	Handle to the device for which the calibration constants are to be saved.
`channel`	Channel to save to the current calibration settings for
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectInputSignal, daqCalSelectCalTable`
Program References	None
Used With	WaveBook/512, WaveBook/516

`daqCalSaveConstants` will save the current calibration table as set by the **`daqCalSelectCalTable`** routine. Current calibration constants can be updated or modified with the **`daqCalSetConstants`** routine. The working calibration table should only be saved after all desired calibration constants have been updated for the device.

daqCalSelectCalTable

DLL Function	`daqCalSelectCalTable(DaqHandleT handle, DaqCalTableTypeT tableType);`
C	`daqCalSelectCalTable(DaqHandleT handle, DaqCalTableTypeT tableType);`
Visual BASIC	`VBdaqCalSelectCalTable(ByVal handle&, ByVal tableType as DaqCalTableTypeT )`
Delphi	`daqCalSelectCalTable( handle: DaqHandleT; tableType : DaqCalTableTypeT)`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`tableType`	Calibration table type to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectInputSignal, daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

`daqCalSelectCalTable` allows the selection of the calibration-table source for the device. Currently, there are two valid calibration-table types which are selected via the **`tableType`** parameter:

- **`DcttFactory`** - Selects the factory calibration table. The factory calibration table reflects factory calibration constants for the selected device. This is the default setting.
- **`DcttUser`** - Selects the user-calibration table. The user-calibration table reflects calibration constants defined by the user or the device's user-calibration application. Refer to the calibration documentation for specific settings.

This function should be used to set the current calibration table for the device. The current calibration table at any time will be set to the calibration table last selected during the current device session.

daqCalSelectInputSignal

DLL Function	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
C	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
Visual BASIC	`VBdaqCalSelectInputSignal(ByVal handle&, ByVal input as DaqCalInputT )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; input: DaqCalInputT)`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`input`	Calibration input signal source to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectCalTable, daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

`daqCalSelectInputSignal` allows the selection of the input signal source for user calibration. The input signal source is specified by the **`input`** parameter. Please refer to the *Calibration Input Signal Sources* table for valid parameters on input signal sources.

daqCalSetConstants

DLL Function	`daqCalSetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, WORD gainConstant, SHORT offsetConstant );`
C	`daqCalSetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, WORD gainConstant, SHORT offsetConstant );`
Visual BASIC	`VBdaqCalSetConstants(ByVal handle&, ByVal channel&, ByVal gain&, ByVal range&, ByVal gainConstant%, ByVal offsetConstant% );`
Delphi	`daqCalSetConstants(handle: DaqHandleT;channel:DWORD; gain: DaqAdcGain; range: DaqAdcRangeT; gainConstant:WORD; offsetConstant:SHORT );`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`channel`	Channel number to apply the calibration settings
`gain`	Gain range to apply the calibration settings
`range`	A/D input range to apply the calibration settings
`gain`	Gain value to apply
`offset`	Offset value to apply
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCalGetConstants, daqCalSelectCalTable, daqCalSelectInputSignal,daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

`daqCalSetConstants` sets the user-accessible calibration constants. These calibration constants are gains and offsets that are applied to the input data. The data comes in, is multiplied by the gain, then the offset is added to it. The resulting data is the conversion between the raw A/D data and the data that is presented during the acquisition. Each channel, gain, and bipolar/unipolar setting has a different pair of gain and offset values. The first three parameters of the **`daqCalSetConstants`** function specify which set of constants are to be changed. The last two parameters are the actual constants. These constants are in a particular binary format. The gain constant is 32768 times the gain. For a gain of ×1, the gain constant is 32768 or 0x8000. The maximum gain is approximately ×2 (65535/32768), and the minimum gain is ×0 (0/32768). The offset (a left-justified signed 12-bit number) is added to the final result. A single least-significant bit has an integer value of 16 or 0x0010. Setting the calibration constants affects subsequent acquisitions until another **`daqOpen`** is performed. After **`daqOpen`**, the original calibration constants are re-read from the NVRAM in the WaveBook and expansion chassis; then, the working copy as set by **`daqCalSetCalConstants`** is overwritten.

daqCalConvert

DLL Function	`daqCalConvert(DaqHandleT handle,PWORD counts, DWORD scans);`
C	`daqCalConvert(DaqHandleT handle,PWORD counts, DWORD scans);`
Visual BASIC	`VBdaqCalConvert&(ByVal handle&, counts%(), ByVal scans&)`
Delphi	`daqCalConvert(handle:DaqHandleT; counts:PWORD; scans:DWORD)`
Parameters	
`handle`	Handle to the device to be calibrated.
`counts`	The raw data from one or more scans.
`scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqReadCalFile, daqCalSetup, daqCalSetupConvert`
Program References	None
Used With	All devices

`daqCalConvert` performs the actual calibration of one or more scans according to the previously called **`daqCalSetup`** function. This function will modify the array of data passed to it. This function should be preceded by the **`daqCalSetup`** function.

The **`counts`** parameter specifies a pointer to an array of the raw A/D counts retrieved during an acquisition. Upon return, the **`counts`** array will hold calibrated data.

The **`scans`** parameter indicates the number of scans (as defined by the current scan group configuration) in the acquisition.

daqCalSetup

DLL Function	`daqCalSetup(DaqHandleT handle,DWORD nscan, DWORD readingsPos, DWORD nReadings, DcalType chanType, DaqAdcGain chanGain, DWORD startChan, BOOL bipolar, BOOL noOffset);`
C	`daqCalSetup(DaqHandleT handle,DWORD nscan, DWORD readingsPos, DWORD nReadings, DcalType chanType, DaqAdcGain chanGain, DWORD startChan, BOOL bipolar, BOOL noOffset);`
Visual BASIC	`VBdaqCalSetupConvert&(ByVal handle&, ByVal nscan&, ByVal readingsPos&, ByVal nReadings&, ByVal chanType&, ByVal chanGain&, ByVal startChan&, ByVal bipolar&, ByVal noOffset&, counts%(), ByVal scans&)`
Delphi	`daqCalSetupConvert(handle:DaqHandleT; nscan:DWORD; readingsPos:DWORD; nReadings:DWORD; chanType:DcalType; chanGain:DaqAdcGain; startChan:DWORD; bipolar:longbool; noOffset:longbool; counts:PWORD; scans:DWORD)`
Parameters	
`handle`	Handle to the device to be calibrated
`nscan`	The number of readings in a single scan.
`readingsPos`	The position of the readings to be calibrated within the scan.
`nReadings`	The number of readings to calibrate.
`chanType`	The type of channel/board from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel of a DBK14 or DBK19, and 0 when reading any other channel.
`chanGain`	The gain setting of the channels to be calibrated.
`startChan`	The channel number of the first channel to be converted.
`bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`noOffset`	If non-zero, the offset cal constant will not be used to calibrate the readings.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqReadCalFile, daqCalConvert, daqCalSetupConvert`
Program References	None
Used With	All devices

`daqCalSetup` will configure the order and type of data to be calibrated. This function requires all data to be calibrated to come from consecutive channels configured for the same gain, polarity, and channel type. The calibration can be configured to use only the gain calibration constant and not the offset constant. This allows the offset to be removed at runtime using the zero compensation functions.

daqCalSetupConvert

DLL Function	`daqCalSetupConvert(DaqHandleT handle,DWORD nscan, DWORD readingsPos, DWORD nReadings, DcalType chanType, DaqAdcGain chanGain, DWORD startChan, BOOL bipolar, BOOL noOffset, PWORD counts, DWORD scans);`
C	`daqCalSetupConvert(DaqHandleT handle,DWORD nscan, DWORD readingsPos, DWORD nReadings, DcalType chanType, DaqAdcGain chanGain, DWORD startChan, BOOL bipolar, BOOL noOffset, PWORD counts, DWORD scans);`
Visual BASIC	`VBdaqCalSetupConvert&(ByVal handle&, ByVal nscan&, ByVal readingsPos&, ByVal nReadings&, ByVal chanType&, ByVal chanGain&, ByVal startChan&, ByVal bipolar&, ByVal noOffset&, counts%(), ByVal scans&)`
Delphi	`daqCalSetupConvert(handle:DaqHandleT; nscan:DWORD; readingsPos:DWORD; nReadings:DWORD; chanType:DcalType; chanGain:DaqAdcGain; startChan:DWORD; bipolar:longbool; noOffset:longbool; counts:PWORD; scans:DWORD)`
Parameters	
`handle`	Handle to the device to be calibrated
`nscan`	The number of readings in a single scan.
`readingsPos`	The position of the readings to be calibrated within the scan.
`nReadings`	The number of readings to calibrate.
`chanType`	The type of channel/board from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel of a DBK14 or DBK19, and 0 when reading any other channel.
`chanGain`	The gain setting of the channels to be calibrated.
`startChan`	The channel number of the first channel to be converted.
`bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`noOffset`	If non-zero, the offset cal constant will not be used to calibrate the readings.
`counts`	The raw data from one or more scans.
`scans`	The number of scans of raw data in the counts array.
Returns	DerrZCInvParam - Invalid parameter value DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqReadCalFile, daqCalSetup, daqCalConvert
Program References	None
Used With	All devices

`daqCalSetupConvert` allows you to perform both the setup and convert steps with one call to **`daqCalSetupConvert`**. This is useful when the calibration needs to be performed multiple times because data was read from non-consecutive channels or at different gains.

daqClose

DLL Function	`daqClose(DaqHandleT handle);`
C	`daqClose(DaqHandleT handle);`
Visual BASIC	`VBdaqClose&(ByVal handle&)`
Delphi	`daqClose(handle:DaqHandleT)`
Parameters	
`handle`	Handle to the device to be closed
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqOpen
Program References	ADCEX1.C, DACEX1.C, DIGEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS, ERREX.PAS (Delphi)
Used With	All devices

`daqClose` is used to close a Daq* device. Once the specified device has been closed, no subsequent communication with the device can be performed. In order to re-establish communications with a closed device, the device must be re-opened with the **`daqOpen`** function.

daqCvtLinearConvert

DLL Function	`daqCvtLinearConvert(PWORD counts, DWORD scans, PFLOAT fValues, DWORD nValues);`
C	`daqCvtLinearConvert(PWORD counts, DWORD scans, PFLOAT fValues, DWORD nValues);`
Visual BASIC	`VBdaqCvtLinearConvert&(counts%(), ByVal scans&, fValues!(), ByVal nValues&)`
Delphi	`daqCvtLinearConvert(counts:PWORD; scans:DWORD; fValues:PSINGLE; nValues:DWORD)`
Parameters	
`counts`	The acquired ADC readings to be converted.
`scans`	The number of scans to be converted.
`fValues`	An array to hold the converted readings.
`nValues`	The size of the reading array.
Returns	Refer to *API Error Codes on page 3-81*
See Also	`daqCvtLinearSetup, daqCvtLinearSetupConvert`
Program References	None
Used With	All devices

`daqCvtLinearConvert` converts the ADC readings into floating point numbers using the linear relationship that was specified with **`daqCvtLinearSetup`**. **`daqCvtLinearConvert`** may be invoked repeatedly to perform multiple conversions, each using the same linear relationship.

daqCvtLinearSetup

DLL Function	`daqCvtLinearSetup(DWORD nscan, DWORD readingsPos, DWORD nReadings, FLOAT signal1, FLOAT voltage1, FLOAT signal2, FLOAT voltage2, DWORD avg);`
C	`daqCvtLinearSetup(DWORD nscan, DWORD readingsPos, DWORD nReadings, FLOAT signal1, FLOAT voltage1, FLOAT signal2, FLOAT voltage2, DWORD avg);`
Visual BASIC	`VBdaqCvtLinearSetupConvert&(ByVal nscan&, ByVal readingsPos&, ByVal nReadings&, ByVal signal1!, ByVal voltage1!, ByVal signal2!, ByVal voltage2!, ByVal avg&, counts%(), ByVal scans&, fValues!(), ByVal nValues&)`
Delphi	`daqCvtLinearSetupConvert(nscan:DWORD; readingsPos:DWORD; nReadings:DWORD; signal1:single; voltage1:single; signal2:single; voltage2:single; avg:DWORD; counts:PWORD; scans:DWORD; fValues:PSINGLE; nValues:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan (1 to 512)
`readingsPos`	The position within the scan of the first reading to convert (0 to `nscan` - 1)
`nReadings`	The number of consecutive ADC readings to convert (1 to `nscan` - `readingPos`)
`signal1`	The transducer input signal that produces `voltage1`
`voltage1`	The transducer output voltage for input `signal1`
`signal2`	The transducer input signal that produces `voltage2`
`voltage2`	The transducer output voltage for input `signal2`
`avg`	The type of averaging to use. 0 = block averaging, 1 = no averaging, 2 or greater = moving average. “0” specifies block averaging in which all of the scans are averaged together to compute a single value for each channel. “1” specifies no averaging. Each scan’s readings are converted into measured signals. “2” (or more) specifies moving average of the specified number of scans. Each scan’s readings are averaged with the avg-1 preceding scans’ readings before conversion. The first scan is not averaged because there is not enough data. For example, if `avg` is “3”, then the results from the first scan are not averaged at all; the results from the second scan are averaged with the first scan; the results from the third and subsequent scans are averaged with the preceding two scans as shown in the next table.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCvtLinearSetup, daqCvtLinearSetupConvert`
Program References	None
Used With	All devices

`daqCvtLinearSetup` saves the data required for **`daqCvtLinearConvert`** to perform conversions. Four parameters are used to specify a linear relationship: the transducer input signal level and output voltage at 2 points in the range.

	Readings from Channel		Results from Channel	
Scan	**0**	**1**	**0**	**1**
1	1A	2A	1A	2A
2	1B	2B	(1A+1B)/2	(2A+2B)/2
3	1C	2C	(1A+1B+1C)/3	(2A+2B+2C)/3
4	1D	2D	(1B+1C+1D)/3	(2B+2C+2D)/3
5	1E	2E	(1C+1D+1E)/3	(2C+2D+2E)/3
6	1F	2F	(1D+1E+1F)/3	(2D+2E+2F)/3

daqCvtLinearSetupConvert

DLL Function	`daqCvtLinearSetupConvert(DWORD nscan, DWORD readingsPos, DWORD nReadings, FLOAT signal1, FLOAT voltage1, FLOAT signal2, FLOAT voltage2, DWORD avg, PWORD counts, DWORD scans, PFLOAT fValues, DWORD nValues);`
C	`daqCvtLinearSetupConvert(DWORD nscan, DWORD readingsPos, DWORD nReadings, FLOAT signal1, FLOAT voltage1, FLOAT signal2, FLOAT voltage2, DWORD avg, PWORD counts, DWORD scans, PFLOAT fValues, DWORD nValues);`
Visual BASIC	`VbdaqCvtLinearSetupConvert&(ByVal nscan&, ByVal readingsPos&, ByVal nReadings&, ByVal signal1!, ByVal voltage1!, ByVal signal2!, ByVal voltage2!, ByVal avg&, counts%(), ByVal scans&, fValues!(), ByVal nValues&)`
Delphi	`daqCvtLinearSetupConvert(nscan:DWORD; readingsPos:DWORD; nReadings:DWORD; signal1:single; voltage1:single; signal2:single; voltage2:single; avg:DWORD; counts:PWORD; scans:DWORD; fValues:PSINGLE; nValues:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan (1 to 512).
`readingsPos`	The position within the scan of the first reading to convert (0 to `nscan` - 1).
`nReadings`	The number of consecutive ADC readings to convert (1 to `nscan - readingPos`)
`signal1`	The transducer input signal that produces `voltage1`.
`voltage1`	The transducer output voltage for input `signal1`.
`signal2`	The transducer input signal that produces `voltage2`.
`voltage2`	The transducer output voltage for input `signal2`.
`avg`	The type of averaging to use. 0 = block averaging, 1 = no averaging, 2 or greater = moving average. “0” specifies block averaging in which all of the scans are averaged together to compute a single value for each channel. “1” specifies no averaging. Each scan’s readings are converted into measured signals. “2” (or more) specifies moving average of the specified number of scans. Each scan’s readings are averaged with the avg-1 preceding scans’ readings before conversion. The first scan is not averaged because there is not enough data. For example, if `avg` is “3”, then the results from the first scan are not averaged at all; the results from the second scan are averaged with the first scan; the results from the third and subsequent scans are averaged with the preceding two scans as shown in the next table.
`counts`	The acquired ADC readings to be converted.
`scans`	The number of scans to be converted.
`fValues`	An array to hold the converted readings.
`nValues`	The size of the reading array.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCvtLinearConvert, daqCvtLinearSetup`
Program References	None
Used With	All devices

`daqCvtLinearSetupConvert` combines the linear setup and conversion processes into one function.

daqCvtRawDataFormat

DLL Function	daqCvtRawDataFormat(PWORD buf, DaqAdcCvtAction **action, DWORD lastRetCount, DWORD scanCount, DWORD chanCount);**
C	daqCvtRawDataFormat(PWORD buf, DaqAdcCvtAction **action, DWORD lastRetCount, DWORD scanCount, DWORD chanCount);**
Visual BASIC	VBdaqCvtRawDataFormat&(buf%, ByVal action&, ByVal lastRetCount&,ByVal scanCount&, ByVal chanCount&)
Delphi	daqCvtRawDataFormat(PWORD buf, action:DaqAdcCvtAction; lastRetCount:DWORD; scanCount:DWORD: chanCount:DWORD);
Parameters	
buf	Pointer to the buffer containing the raw data
action	The type of conversion action to perform on the raw data
lastRetCount	The last retCount returned from **daqAdcTransferGetStat**
scanCount	The length of the raw data buffer in scans
chanCount	The number of channels per scan in the raw data buffer
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 3-83)
See Also	daqAdcSetDataFormat
Program References	None
Used With	All devices

daqCvtRawDataFormat allows the conversion of raw data to a specified format. This function should be called after the raw data has been acquired. See the transfer data functions (**daqAdcTransfer**…) for more details on the actual collection of raw data.

The **buf** parameter specifies the pointer to the data buffer containing the raw data. Prior to calling this function, this user-allocated buffer should already contain the entire raw data transfer. Upon completion, this data buffer will contain the converted data (the buffer must be able to contain all the converted data).

The **action** parameter specifies the type of conversion to perform. The **DacaUnpack** value can be used de-compress raw data. The **DacaRotate** can be used to reformat a circular buffer into a linear buffer.

The **scanCount** parameter specifies the length of the raw buffer in scans. Since the converted data will overwrite the raw data in the buffer, make sure the specified buffer is large enough, physically, to contain all of the converted data.

The **chanCount** parameter specifies the number of channels in each scan.

daqCvtRtdConvert

DLL Function	`daqCvtRtdConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
C	`daqCvtRtdConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
Visual BASIC	`VBdaqCvtRtdConvert&(counts%(), ByVal scans&, temp%(), ByVal ntemp&)`
Delphi	`daqCvtRtdConvert(counts:PWORD; scans:DWORD; temp:PWORD; nTemp:DWORD)`
Parameters	
`counts`	Raw A/D data from one or more scans
`scans`	Number of scans of raw data in counts
`temp`	Variable array to hold converted temperatures
`ntemp`	Size of temperature array (should be number of RTDs specified in setup times the number of scans)
Returns	`DerrRtdNoSetup` - Setup was not called `DerrRtdTArraySize` - Temperature array is not large enough `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`DaqRtdSetup, DaqRtdSetupConvert`
Program References	None
Used With	All devices

`daqRTDConvert` takes raw A/D readings from RTDs and converts them to temperature readings in tenths-of-degrees Celsius (0.1°C). **Note**: The total number of conversions [scan * (RTD channels per scan) * 4] must be less than 32K.

The Daq* measures temperatures sensed by RTDs attached via a DBK9 RTD expansion card. Up to 8 RTDs can attach to each DBK9. Up to 32 DBK9s may be attached to a single Daq* for a maximum of 256 temperatures. The software currently supports 100-, 500-, and 1000-ohm RTDs.

The RTD measurement functions are designed for simple temperature measurement in which each RTD channel is read 4 times. These 4 readings must be grouped together in the scan and in order: **`Dbk9VoltageA`** (gain=0), **`Dbk9VoltageB`** (gain=1), **`Dbk9VoltageD`** (gain=3), **`Dbk9VoltageD`** (gain=3). The RTDs must be of the same type, and the reading groups must follow each other in the scan sequence.

The temperature conversion functions use input data from one or more Daq* scans. They take 4 voltage readings for each RTD channel, apply the appropriate averaging method, convert the voltages to a resistance and then (using the appropriate curves for the RTD type) convert the resistance into a temperature. For example, assume the following readings:

Scan	Readings Channel 0				Readings Channel 1			
	0	1	2	3	4	5	6	7
1	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
2	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
3	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
4	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
5	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd

The 4 readings for each channel are grouped together in order. If this scan data is passed to **`daqRtdConvert`** (through the counts parameter) with averaging disabled (**`avg`** parameter set to 1), the function will return the temp parameters shown in the table. **Note**: Temperatures returned will be in tenths of a degree Celsius.

Temperatures		
Scan	0	1
1	Ch 0 °C	Ch 1 °C
2	Ch 0 °C	Ch 1 °C
3	Ch 0 °C	Ch 1 °C
4	Ch 0 °C	Ch 1 °C
5	Ch 0 °C	Ch 1 °C

If the scan data is passed to **`daqRtdConvert`** (in the **`counts`** parameter) with averaging set to block averaging (**`avg`** parameter set to 0), the function will return the **`temp`** parameters shown in the table.

Temperatures		
	0	1
Average of all Temperatures	Ch 0 °C	Ch 1 °C

The conversion process is divided into two steps: setup and conversion. *Setup* describes the characteristics of the temperature measurement; *Conversion* changes raw readings into temperatures. For convenience, both setup and conversion can be performed at once by **`daqRtdSetupConvert`**. All of the functions return error codes, which are defined in Daqx.h.

daqCvtRtdSetup

DLL Function	`daqCvtRtdSetup(DWORD nScan, DWORD startPosition, DWORD nRtd, RtdType, rtdType, DWORD avg);`
C	`daqCvtRtdSetup(DWORD nScan, DWORD startPosition, DWORD nRtd, RtdType rtdType, DWORD avg);`
Visual BASIC	`VBdaqCvtRtdSetup&(ByVal nscan&, ByVal startPosition&, ByVal nRtd&, ByVal rtdType&, ByVal avg&)`
Delphi	`daqCvtRtdSetup(nScan:DWORD; startPosition:DWORD; nRtd:DWORD; rtdType:RtdType; avg:DWORD)`
Parameters	
`nScan`	The total number of readings in a scan. valid range 1-512
`startPosition`	Position of the first RTD reading group in the scan. Valid range 1-509
`nRtd`	Number of RTD reading groups in the scan. Valid range 1-128
`rtdType`	Value of RTD being used. `Dbk9RtdType100` - 100 ohm RTD `Dbk9RtdType500` - 500 ohm RTD `Dbk9RtdType1K` - 1000 ohm RTD
`avg`	Type of averaging to be used. 0 = block averaging 1 = no averaging 2 to (number of scans -1) = moving average
Returns	`DerrRtdParam` - Setup parameter out of range `DerrRtdValue` - Invalid RTD type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`DaqRtdSetupConvert, DaqRtdSetupConvert`
Program References	None
Used With	All devices

`daqCvtRtdSetup` sets up parameters for subsequent RTD temperature conversions. Refer to the discussion of **`daqRTDConvert.`**

daqCvtRtdSetupConvert

DLL Function	`daqCvtRtdSetupConvert(DWORD nScan, DWORD startPosition, DWORD nRtd, RtdType rtdType, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
C	`daqCvtRtdSetupConvert(DWORD nScan, DWORD startPosition, DWORD nRtd, RtdType rtdType, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
Visual BASIC	`VBdaqCvtRtdSetupConvert&(ByVal nscan&, ByVal startPosition&, ByVal nRtd&, ByVal rtdType&, ByVal avg&, counts%(), ByVal scans&, temp%(), ByVal ntemp&)`
Delphi	`daqCvtRtdSetupConvert(nScan:DWORD; startPosition:DWORD; nRtd:DWORD; rtdType:RtdType; avg:DWORD; counts:PWORD; scans:DWORD; temp:PWORD; ntemp:DWORD)`
Parameters	
`nScan`	The total number of readings in a scan. valid range 1-512
`startPosition`	Position of the first RTD reading group in the scan. Valid range 1-509
`nRtd`	Number of RTD reading groups in the scan. Valid range 1-128
`rtdType`	Value of RTD being used. `Dbk9RtdType100` - 100 ohm RTD `Dbk9RtdType500` - 500 ohm RTD `Dbk9RtdType1K` - 1000 ohm RTD
`avg`	Type of averaging to be used 0 = block averaging 1 = no averaging 2 to (number of scans -1) = moving average
`counts`	Raw A/D data readings from one or more scans.
`scans`	Number of scans of raw data in contained in *counts.
`temp`	Array to hold converted temperatures.
`ntemp`	Size of temperature array. Should be the number of RTDs times the number of scans for no averaging and moving averages or the number of RTDs for block averaging.
Returns	`DerrRtdParam` - Setup parameter out of range `DerrRtdValue` - Invalid RTD type `DerrRtdTArraySize` - temperature storage array not large enough `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqRtdSetup, daqRtdConvert`
Program References	None
Used With	All devices

`daqCvtRtdSetupConvert` sets up and converts raw A/D readings from RTDs into temperature readings. Refer to the discussion of **`daqRTDConvert`**.

daqCvtSetAdcRange

DLL Function	`daqCvtSetAdcRange(FLOAT Admin, FLOAT Admax);`
C	`daqCvtSetAdcRange(FLOAT Admin, FLOAT Admax);`
Visual BASIC	`VBdaqCvtSetAdcRange&(ByVal ADmin!, ByVal ADmax!)`
Delphi	`daqCvtSetAdcRange(Admin:single; Admax:single)`
Parameters	
`Admin`	A/D minimum voltage range
`Admax`	A/D maximum voltage range
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	
Program References	None
Used With	

`daqCvtSetAdcRange` allows you to set the current ADC range for use by the **`daqCvt`**… functions. This function should not need to be called if used for data collected by the Daq* devices.

daqCvtTCConvert

DLL Function	`daqCvtTCConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
C	`daqCvtTCConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
Visual BASIC	`VBdaqCvtTCConvert&(counts%(), ByVal scans&, temp%(), ByVal ntemp&)`
Delphi	`daqCvtTCConvert(counts:PWORD; scans:DWORD; temp:PWORD; ntemp:DWORD)`
Parameters	
`counts`	An array of one or more scans of raw data as received from the Daq. The ADC data bits are in the 12 most significant bits of the 16-bit integers, and the tag bits (which are discarded) are in the 4 least-significant bits. Valid range: Each raw data item may be any 16-bit value.
`scans`	The number of scans of data in `counts` array. Valid range: 1 to 32768/nscan (counts is limited to 64 Kbytes).
`temp`	Variable array to hold converted temperature results. The integer values are 10 times the temperatures in °C. For example, 50°C would be represented as 500 and -10°C would be -100. Valid range: Results range from -2000 (-200°C) to +13720 (+1372°C) depending on the thermocouple type.
`ntemp`	The number of entries in the temperature array. This value is checked by the functions to avoid writing past the end of the array. Valid range: If avg is 0, then ntc or greater. If avg is non-zero, then scans * ntc or greater.
Returns	`DerrTCE_NOSETUP` - Setup was not called `DerrTCE_PARAM` - Parameter out of range `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`DaqCvtTCSetup, DaqCvtTCSetupConvert`
Program References	None
Used With	All devices

`daqCvtTCConvert` takes raw A/D readings and converts them to temperature readings in tenths of degrees Celsius (0.1°C). The total number of conversions (scan * chans/scan) must be less than 32K. The Daq* measures thermocouple temperatures by way of a DBK19 or DBK52 that includes a cold-junction compensation circuit (CJC) attached to channel 0. Channel 1 is shorted for performing auto-zero compensation. Channels 2 through 15 accept thermocouples for temperature measurement. Up to 16 expansion cards may be attached to a single Daq* to measure a maximum of 224 (16×14) temperatures. The software supports type J, K, T, E, N28, N14, S, R and B thermocouples.

Two software techniques (calibration and zero compensation) can be used to increase the accuracy of the DBK19 card:

- Software calibration uses gain and offset calibration constants, unique to each card, to compensate for inherent errors on the card.
- Zero compensation is a method by which any offset voltage on the card can be removed at run-time. This is done by measuring a shorted channel at the same gain on the actual input to find the offset, and subtracting this value from the actual reading.

The thermocouple linearization function has a special auto-zero compensation feature that will perform zero compensation on the raw thermocouple data before linearizing when using a DBK19. The auto-zero feature is enabled by default, but can be disabled using the **`daqZeroDbk19`** function. It is not available when using unipolar mode.

The temperature measurement conversion functions are designed for temperature measurement where:

- The cold-junction compensation circuit (CJC) channel (channel 0) reading from the T/C card is immediately followed in the scan sequence by the T/C channel readings, all of which must be from the same type of T/C (including: J, K, T, E, N28, N14, S, R, or B).
- If a DBK19 is used with auto-zeroing enabled, the CJC channel reading described above must be preceded by 2 readings from the shorted channel (channel 1). The first shorted reading must be at the same gain setting as the CJC reading. The other shorted reading must be at the gain of the T/C to be converted.
- If software calibration is used with the DBK19, the calibration constants for the card to be used should be entered into the calibration file.
- The CJC and T/C readings are taken with the optimal gains (as described below).
- All non-thermocouple data conversion, if any, must be done by other means.

The temperature conversion functions take input data from one or more scans from the Daq*. They then examine the CJC and thermocouple readings within that scan and, after optional averaging, convert them to temperatures which are stored as output. For example, see the readings in the table.

Scan	Reading					
	0	1	2	3	4	5
1	V or CJC Zero	V or J Zero	CJC	J1a	J1b	J1c
2	V or CJC Zero	V or J Zero	CJC	J2a	J2b	J2c
3	V or CJC Zero	V or J Zero	CJC	J3a	J3b	J3c
4	V or CJC Zero	V or J Zero	CJC	J4a	J4b	J4c

The first 2 readings of each scan are non-temperature voltage readings to compensate for the CJC circuit and the shorted channel 0. The third reading is from the CJC, and the remaining 3 readings are from 3 type J thermocouples. If the auto-zero feature is disabled, the first 2 readings will be ignored. Otherwise, the first 2 readings will be used to remove offset errors in the CJC and T/C reading. The CJC and T/C readings are used to produce one temperature result for each T/C reading. Thus, the 24 original readings are reduced to 12 temperatures.

The conversion process has 2 steps: setup and conversion. *Setup* describes the characteristics of the temperature measurement, and *Conversion* changes the raw readings into temperatures. All of the functions return error codes as defined in Daqx.h which also includes the function prototypes and the definitions of the thermocouple-type codes.

To measure temperatures, the scan must be set up so the T/C measurements consecutively follow their corresponding CJC measurement (the CJC measurement need not be the first element in the scan). If auto-zeroing is enabled, the CJC measurement must be preceded by both a CJC zero measurement and a T/C zero measurement.

All of the thermocouples converted with a single invocation of the conversion functions must be of the same type: J, K, T, E, N28, N14, S, R, or B. To measure with more than one type of thermocouple, they must be sorted by type within the scan, and each type must be preceded by the related CJC.

The scan is not restricted to thermocouple measurements. The scan may include other types of signals such as voltage, current, or digital input; but conversion of these readings is up to you. The temperature conversion functions cannot handle them.

Gain Codes				
Type	Unipolar Gain Code	Unipolar Gain	Bipolar Gain Code	Bipolar Gain
CJC	`Dbk19UniCJC`	90	`Dbk19BiCJC`	60
J	`Dbk19UniTypeJ`	180	`Dbk19BiTypeJ`	90
K	`Dbk19UniTypeK`	180	`Dbk19BiTypeK`	90
T	`Dbk19UniTypeT`	240	`Dbk19BiTypeT`	180
E	`Dbk19UniTypeE`	90	`Dbk19BiTypeE`	60
N28	`Dbk19UniTypeN28`	240	`Dbk19BiTypeN28`	240
N14	`Dbk19UniTypeN14`	180	`Dbk19BiTypeN14`	90
S	`Dbk19UniTypeS`	240	`Dbk19BiTypeS`	240
R	`Dbk19UniTypeR`	180	`Dbk19BiTypeR`	240
B	`Dbk19UniTypeB`	240	`Dbk19BiTypeB`	240

The temperature measurements must be made with the correct gain settings. The gain settings for the different thermocouple types depend on the channel type and the bipolar/unipolar setting of the Daq* as specified in the table. **Note**: Unipolar operations are not recommended for thermocouple measurement unless the measured temperatures will be greater than the Daq* temperature.

When measuring thermocouples using the gains above, the following temperature ranges apply.

Thermocouple mV Outputs For Temperature Ranges Depending on Ambient Temperature						
T/C Type	Measured Temperature Range @ 0°C ambient		Measured Temperature Range @ 25°C ambient		Measured Temperature Range @ 50°C ambient	
	Temperature °C	0°C Output (mV)	Temperature°C	25°C Output (mV)	Temperature°C	50°C Output (mV)
J	-200 to 760	-7.9 to 42.9	-200 to 760	-9.2 to 41.6	-200 to 760	-11.8 to 39.0
K	-200 to 1372	-5.9 to 54.9	-200 to 1372	-6.9 to 53.9	-200 to 1372	-8.9 to 52.9 (50.0
T	-200 to 400	-5.6 to 20.9	-200 to 400	-6.6 to 19.9	-200 to 400	-8.7 to 17.7
E	-270 to 1000	-9.8 to 76.4	-270 to 1000	-11.3 to 74.9	-270 to 1000	-14.5 to 71.7
N28	-270 to 400	-4.3 to 13.0	-270 to 400	-5.0 to 12.3	-270 to 400	-6.4 to 10.9
N14	0 to 1300	0.0 to 47.5	0 to 1300	-0.7 to 46.8	0 to 1300	-2.0 to 45.5
S	-50 to 1780	-0.2 to 18.8	-50 to 1780	-0.4 to 18.7	-50 to 1780	-0.7 to 18.4
R	-50 to 1780	-0.2 to 21.3	-50 to 1780	-0.4 to 21.1	-50 to 1780	-0.7 to 20.8
B	50 to 1780	0.0 to 13.4	50 to 1780	0.0 to 13.4	50 to 1780	0.0 to 13.4

daqCvtTCSetup

DLL Function	`daqCvtTCSetup(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg);`
C	`daqCvtTCSetup(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg);`
Visual BASIC	`VBdaqCvtTCSetup&(ByVal nscan&, ByVal cjcPosition&, ByVal ntc&, ByVal tcType&, ByVal bipolar&, ByVal avg&)`
Delphi	`daqCvtTCSetup(nscan:DWORD; cjcPosition:DWORD; ntc:DWORD; tcType:TCType; bipolar:longbool; avg:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan of DaqBook/DaqBoard data. The daqCvtTC… functions can convert several consecutive scans worth of data in a single invocation. Valid range: 2 to 512.
`cjcPosition`	The position of the actual cold-junction compensation circuit (CJC) reading within each scan (not the CJC zero reading, if any). The first reading of the scan is position 0, and the last reading is nscan -1. Each scan of temperature data must include a reading of the CJC signal on the expansion board to which the thermocouples are attached. The CJC readings must be taken with the gain in the section *Scan Setup*. Valid range: 0 to nscan-2 with no zero compensation; 2 to nscan-2 with zero compensation.
`ntc`	The number of thermocouple signals that are to be converted to temperature values. The thermocouple signal readings must immediately follow the CJC reading in the scan data. The first thermocouple signal is at scan position cjcPosition+1,; the next is at cjcPosition+2,; and so on. Valid range: 1 to nscan-1-cjcPosition.
`tcType`	The type of thermocouples that generated the measurements. Valid range: One of the pre-defined values, **`Dbk19TCTypeJ, Dbk19TCTypeK, Dbk19TCTypeT, Dbk19TCTypeE, Dbk19TCTypeN28, Dbk19TCTypeN14, Dbk19TCTypeS, Dbk19TCTypeR`** or **`Dbk19TCTypeB.`**
`bipolar`	Must be set true (non-zero) if the readings were acquired with the Daq* set for bipolar operation. Must be set false (zero) for unipolar operation. The required gain settings for the CJC and thermocouple channels change depending on the unipolar/bipolar mode. Valid range: 0 for unipolar or any non-zero value for bipolar.
`avg`	The type of averaging to be performed. Valid range: any unsigned integer. Since the thermocouple voltage may be small compared to the ambient electrical noise, averaging may be necessary to yield a steady temperature output. 0 specifies block averaging in which all of the scans are averaged together to compute a single temperature measurement for each of the ntemp thermocouples. 1 specifies no averaging. Each scan's readings are converted into ntemp measured temperatures for a total of scans*ntemp results. 2 or more specifies moving average of the specified number of scans. Scan readings are averaged with the avg-1 preceding scans' readings before conversion. The first avg-1 scans are averaged with all of the preceding scans because they do not have enough preceding scans. For example, if avg is 3, then the results from the first scan are not averaged at all, the results from the second scan are averaged with the first scan, the results from the third and subsequent scans are averaged with the preceding two scans as shown in the table.
Returns	`DerrTCE_PARAM` - Parameter out of range `DerrTCE_TYPE` - Invalid thermocouple type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqCvtTCConvert, daqCvtTCSetupConvert`
Program References	None
Used With	All devices

`daqCvtTCSetup` sets up parameters for subsequent temperature conversions. The next table shows how averages are computed.

Scan	Readings from Channel		Results from Channel	
	0	1	0	1
1	1A	2A	1A	2A
2	1B	2B	(1A+1B)/2	(2A+2B)/2
3	1C	2C	(1A+1B+1C)/3	(2A+2B+2C)/3
4	1D	2D	(1B+1C+1D)/3	(2B+2C+2D)/3
5	1E	2E	(1C+1D+1E)/3	(2C+2D+2E)/3
6	1F	2F	(1D+1E+1F)/3	(2D+2E+2F)/3

daqCvtTCSetupConvert

DLL Function	`daqCvtTCSetupConvert(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
C	`daqCvtTCSetupConvert(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);`
Visual BASIC	`VBdaqCvtTCSetupConvert&(ByVal nscan&, ByVal cjcPosition&, ByVal ntc&, ByVal tcType&, ByVal bipolar&, ByVal avg&, counts%(), ByVal scans&, temp%(), ByVal ntemp&)`
Delphi	`daqCvtTCSetupConvert(nscan:DWORD; cjcPosition:DWORD; ntc:DWORD; tcType:TCType; bipolar:longbool; avg:DWORD; counts:PWORD; scans:DWORD; temp:PWORD; ntemp:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan. Valid range: 1- 512
`cjcPosition`	The position of the CJC reading within the scan. Valid range: 0 -(nscan-1) 2 -(nscan-1), if auto-zeroing is used with DBK19.
`ntc`	The number of thermocouple readings that immediately follow the CJC reading within the scan. Valid range: 1 -(nscan-cjcposition-1)
`tcType`	The type of thermocouples being measured.
`bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`avg`	The type of averaging to be performed: block, none or moving.
`counts`	The raw data from one or more scans.
`scans`	The number of scans of raw data in counts.
`temp`	The converted temperatures in tenths of a degree C.
`ntemp`	The number of elements provided in the temp array (for error checking).
Returns	`DerrTCE_PARAM` - Parameter out of range `DerrTCE_TYPE` - Invalid thermocouple type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 3-83)
See Also	`DaqCvtTCSetup, daqCvtTCConvert`
Program References	None
Used With	All devices

`daqCvtTCSetupConvert` sets up and converts raw A/D readings into temperature readings.

daqDacSetOutputMode

DLL Function	`daqDacSetOutputMode(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacOutputMode outputMode);`
C	`daqDacSetOutputMode(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacOutputMode outputMode);`
Visual BASIC	`VBdaqDacSetOutputMode&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal outputMode&)`
Delphi	`daqDacSetOutputMode(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; outputMode:DaqDacOutputMode)`
Parameters	
`handle`	Handle to the device to set the DAC waveform output mode
`deviceType`	Specifies the device type
`chan`	Specifies the DAC channel
`outputMode`	Defines the DAC waveform output mode to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWt, daqDacWtMany`
Program References	DACEX1.C, DAQEX.FRM (VB), ADCEX.PAS, DACEX.PAS (Delphi)
Used With	DaqBook100, DaqBook112, DaqBook120, DaqBook200, DaqBook216,DaqBoard100A, DaqBoard 112A DaqBoard200A, DaqBoard216A

`daqDacSetOutputMode` allows you to set the output mode of the DAC operations for the specified DAC channel. The **`outputMode`** parameter indicates the type of waveform output update to be performed on the specified DAC channel. The following DAC modes can be specified:

- **`DdomVoltage`** - Specifies a single voltage output mode. This mode defines the output mode of the specified DAC channel to be updated only when written to explicitly. With this mode, no waveform outputs can be generated. See the **`daqDacWt`** and **`daqDacWtMany`** functions for DAC channel voltage updates.
- **`DdomStaticWave`** - Specifies static waveform output. This mode allows the generation of a non-streamed waveform output to the specified DAC channel. In this mode, the output stream cannot be continuously updated by the application during actual waveform output. Once the output data buffer has been set and the waveform operation has been initiated, the output data buffer remains static. This mode requires the specified waveform to fit within the physical size of the FIFO on the device.
- **`DdomDynamicWave`** - Specifies dynamic waveform generation. This mode allows continual, dynamic updating of the DAC waveform during DAC waveform output. Dynamic waveform generation is not size dependent, and waveform updating can be performed indefinitely. Actual waveform generation updating is performed by continually feeding waveform data to the device using the **`daqDacWaveSetBuffer`** and **`daqDacTransferStart`** routines to continually fill the device's DAC FIFO. The waveform transfer operation to the DAC FIFO can be halted at any time with **`daqDacTransferStop`**; and the waveform output can be disabled at any time with **`daqDacDisarm`**.

Note: The **`DdomDynamicWave`** output mode is not available on any device at this time.

daqDacTransferGetStat

DLL Function	`daqDacTransferGetStat(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PDWORD active, PDWORD retCount);`
C	`daqDacTransferGetStat(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PDWORD active, PDWORD retCount);`
Visual BASIC	`VBdaqDacTransferGetStat&(ByVal handle&, ByVal deviceType&, ByVal chan&, active&, retCount&)`
Delphi	`daqDacTransferGetStat(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; var active:DWORD; var retCount:DWORD)`
Parameters	
`handle`	Handle of the device from which to retrieve current DAC transfer status
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`active`	Bit mask representing flags indicating the current state of the DAC waveform transfer
`retCount`	Total number of DAC waveform samples transferred for the current DAC waveform transfer
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacTransferSetBuffer, daqDacTransferStart, daqDacTransferStop`
Program References	None
Used With	

`daqDacTransferGetStat` allows you to get the current status of a DAC dynamic waveform transfer for the specified DAC channel. This function will return the current status and the total transfer count of the current DAC waveform output.

The active parameter is a bit mask representing various DAC waveform events. The bit masks for each event are defined as follows:

- **`DdafWaveformActive`** - This bit set indicates that a DAC waveform output is currently active.
- **`DdafWaveformTriggered`** - This bit set indicates that the DAC waveform output has been triggered and waveform output is currently taking place.
- **`DdafTransferActive`** - This bit set indicates that a DAC dynamic waveform transfer to the devices DAC FIFO is taking place.

The **`retCount`** parameter indicates the total number of DAC dynamic waveform samples that have been transferred since the start of the transfer. **Note:** DAC output mode must be set to **`DdomDynamicWave`** for this function to be called. See the **`daqDacSetOutputMode`** function.

daqDacTransferStart

DLL Function	`daqDacTransferStart(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
C	`daqDacTransferStart(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
Visual BASIC	`VBdaqDacTransferStart&(ByVal handle&, ByVal deviceType&, ByVal chan&)`
Delphi	`daqDacTransferStart(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD)`
Parameters	
`handle`	Handle to the device for which a DAC waveform transfer is to be initiated
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacTransferSetBuffer, daqDacTransferGetStat, daqDacTransferStop, daqDacWaveDisarm`
Program References	None
Used With	

`daqDacTransferStart` allows you to initiate a DAC dynamic waveform output transfer for the specified DAC channel. The waveform transfer will be performed from the waveform buffer configured using the **`daqDacWaveSetBuffer`** function. The transfer will continue until the entire buffer has been transferred, until the transfer is halted (**`daqDacTransferStop`**); or until the DAC output is disarmed (**`daqDacWaveDisarm`**).

Note: DAC output mode must be set to **`DdomDynamicWave`** for this function to be called. See the **`daqDacSetOutputMode`** function.

daqDacTransferStop

DLL Function	`daqDacTransferStop(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
C	`daqDacTransferStop(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
Visual BASIC	`VBdaqDacTransferStop&(ByVal handle&, ByVal deviceType&, ByVal chan&)`
Delphi	`daqDacTransferStop(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD)`
Parameters	
`handle`	Handle to the device for which the current DAC waveform transfer is to be stopped
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacTransferSetBuffer, daqDacTransferGetStat, daqDacTransferStart, daqDacWaveDisarm`
Program References	None
Used With	

`daqDacTransferStop` allows you to stop a DAC dynamic waveform transfer for the specified DAC channel if one is currently active. This function will terminate the transfer of DAC data; however, it will not halt the waveform output on the DAC channel. DAC output data already sent to the devices DAC FIFO will continue to be output until there is no more data in the FIFO. The transfer may be re-initiated for the same DAC transfer buffer or another buffer by again calling the **`daqDacTransferStart`** function. To terminate the waveform output as well as the transfer, refer to the **`daqDacWaveDisarm`** function.

Note: DAC output mode must be set to **`DdomDynamicWave`** for this function to be called. See the **`daqDacSetOutputMode`** function.

daqDacWaveArm

DLL Function	`daqDacWaveArm(DaqHandleT handle, DaqDacDeviceType deviceType);`
C	`daqDacWaveArm(DaqHandleT handle, DaqDacDeviceType deviceType);`
Visual BASIC	`VBdaqDacWaveArm&(ByVal handle&, ByVal deviceType&)`
Delphi	`daqDacWaveArm(handle:DaqHandleT; deviceType:DaqDacDeviceType)`
Parameters	
`handle`	Handle to the device for which a DAC waveform output is to be armed
`deviceType`	Specifies the DAC type
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveDisarm`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveArm` allows you to arm a DAC waveform output for the specified device. This command enables the DAC waveform output, based upon the current waveform output configuration. Once issued, DAC waveform output will begin on the specified device when the specified DAC waveform output trigger event has occurred. If the trigger event has been configured as **`DdtsImmediate`**, the waveform output will begin immediately.

daqDacWaveDisarm

DLL Function	`daqDacWaveDisarm(DaqHandleT handle, DaqDacDeviceType deviceType);`
C	`daqDacWaveDisarm(DaqHandleT handle, DaqDacDeviceType deviceType);`
Visual BASIC	`VBdaqDacWaveDisarm&(ByVal handle&, ByVal deviceType&)`
Delphi	`daqDacWaveDisarm(handle:DaqHandleT; deviceType:DaqDacDeviceType)`
Parameters	
`handle`	Handle to the device for which a current DAC waveform output is to be disarmed
`deviceType`	Specifies the DAC type
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveArm`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveDisarm` allows you to disarm a DAC waveform output if one is active on the specified device. This function will disable the waveform output on the specified device and terminate any DAC buffer transfers that are currently active. Waveform output will be terminated immediately, regardless of the current state of the waveform output or the state of the device's DAC FIFO.

daqDacWaveGetFreq

DLL Function	`daqDacWaveGetFreq(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PFLOAT freq);`
C	`daqDacWaveGetFreq(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PFLOAT freq);`
Visual BASIC	`VBdaqDacWaveGetFreq&(ByVal handle&, ByVal deviceType&, ByVal chan&, freq!)`
Delphi	`daqDacWaveGetFreq(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; var freq:single)`
Parameters	
`handle`	Handle to the device for which to retrieve the current waveform output frequency
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`freq`	Returns the current DAC waveform output frequency setting
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveArm, daqDacWaveDisarm, daqDacWaveSetFreq`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveGetFreq` gets the current DAC waveform update frequency for the specified device and channel.

daqDacWaveSetBuffer

DLL Function	`daqDacWaveSetBuffer(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PWORD buf, DWORD scanCount, DWORD transferMask);`
C	`daqDacWaveSetBuffer(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, PWORD buf, DWORD scanCount, DWORD transferMask);`
Visual BASIC	`VBdaqDacWaveSetBuffer&(ByVal handle&, ByVal deviceType&, ByVal chan&, buf%(), ByVal scanCount&, ByVal transferMask&)`
Delphi	`daqDacWaveSetBuffer(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; buf:PWORD; scanCount:DWORD; transferMask:DWORD)`
Parameters	
`handle`	Handle to the device for which a DAC waveform transfer buffer is to be configured
`deviceType`	Specifies the DAC type
`Chan`	Specifies the DAC channel
`buf`	Pointer to the user allocated waveform transfer buffer
`ScanCount`	Length of the waveform buffer in output samples
`transferMask`	Configures the buffer transfer mode
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacTransferStart, daqDacWaveTransferStop`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveSetBuffer` allows you to configure the DAC output waveform transfer buffer for the specified device and channel. This function may be used to configure a user-supplied buffer for output to a DAC channel on the specified device.

The supplied buffer must be loaded by the application with the desired output data before invoking the **`daqDacTransferStart`** routine to initiate the DAC waveform transfer.

The **`transferMask`** parameter is used to configure the characteristics of the DAC waveform output transfer. Among other things the transferMask specifies the update mode and the cycle mode of the buffer. The modes can be set as follows:

- **`DdtmCycleOn`** - Specifies the buffer cycle mode. Allows the transfer to continue when the end of the transfer buffer is reached (by wrapping the transfer of DAC data back to the beginning of the buffer). In this mode, the DAC transfer buffer will continue to be wrapped until the transfer/waveform output is halted by the application (**`daqDacTransferStop`**/ **`daqDacDisarm`**). The default setting is **`DdtmCycleOff`**.
- **`DdtmUpdateSingle`** - Specifies the update mode as single sample. The update mode can be set to update for every sample or block of DAC data. The update-on-single-setting allows the DAC transfer buffer to be updated for each sample output to the specified DAC. Compared to the block mode, this setting provides a higher degree of real-time waveform output-buffer updating at the expense of slower aggregate waveform output rates. The default setting is **`DdtmUpdateBlock`**.

daqDacWaveSetClockSource

DLL Function	`daqDacWaveSetClockSource(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacClockSource clockSource);`
C	`daqDacWaveSetClockSource(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacClockSource clockSource);`
Visual BASIC	`VBdaqDacWaveSetClockSource&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal clockSource&)`
Delphi	`daqDacWaveSetClockSource(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; clockSource:DaqDacClockSource)`
Parameters	
`handle`	Handle to the device for which the waveform output clock source is to be set
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`clockSource`	Set the clock to the specified source
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetFreq, daqDacWaveGetFreq`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveSetClockSource` allows you to set the clock source for the DAC waveform update frequency for the specified device and channel. The **`clockSource`** parameter specifies the clock source to use for DAC output. The valid sources for the DAC waveform update clock include:

- **`DdcsGatedDacClock`** - Specifies a clock based upon a gated DAC default timebase (**`DdcsDacClock`**); is gated by TTL input on pin 25 of P1.
- **`Ddcs9513Ctrl`** - Specifies a DAC timebase driven by Counter 1 of the 9513.
- **`DdcsExternalTTL`** - Specifies an external timebase supplied via TTL input pin 21 of P1.
- **`DdcsAdcClock`** - Specifies the current ADC pacer-clock timebase.
- **`DdcsDacClock`** - Specifies the internal DAC default timebase of 10 MHz.

Note: The first 3 of these potential clock sources pass through and are divided by Counter 0 of the 8254 before the update signal reaches the DAC FIFO.

The DAC waveform update frequency is the rate at which samples are sent from the DAC FIFO to a single DAC channel. If the more than one DAC channel waveform output is active, the waveform update frequency for each channel is this rate divided by the total number of active DAC waveform output channels. If, however, all DAC channels are simultaneously outputting the same waveform, then the waveform update frequency for each channel will not be divided by the total number of channels.

daqDacWaveSetDiskFile

DLL Function	`daqDacWaveSetDiskFile(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, LPSTR filename);`
C	`daqDacWaveSetDiskFile(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, LPSTR filename);`
Visual BASIC	`VBdaqDacWaveSetDiskFile&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal filename$)`
Delphi	`daqDacWaveSetDiskFile(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; filename:PChar)`
Parameters	
`handle`	Handle to the device from which to generate the waveform output
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`filename`	String representing the path and filename of the disk file to be output.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetBuffer`
Program References	None
Used With	

`daqDacWaveSetDiskFile` allows you to specify a disk file from which a DAC waveform output will be generated for the specified device and channel.

daqDacWaveSetFreq

DLL Function	`daqDacWaveSetFreq(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, FLOAT freq);`
C	`daqDacWaveSetFreq(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, FLOAT freq);`
Visual BASIC	`VBdaqDacWaveSetFreq&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal freq!)`
Delphi	`daqDacWaveSetFreq(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; freq:single)`
Parameters	
`handle`	Handle to the device for which to set the waveform output update frequency
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`freq`	Sets the DAC waveform output frequency to the specified frequency
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveGetFreq, daqDacWaveSetClockSource`
Program References	None
Used With	

daqDacWaveSetFreq allows you to set the DAC waveform update frequency for the specified device and channel. The frequency is set via the **freq** parameter and is dependent upon the clock source chosen for the selected device. The clock source can be configured by using the **daqDacWaveSetClockSource** function.

daqDacWaveSetMode

DLL Function	`daqDacWaveSetMode(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacWaveformMode mode, DWORD updateCount);`
C	`daqDacWaveSetMode(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacWaveformMode mode, DWORD updateCount);`
Visual BASIC	`VBdaqDacWaveSetMode&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal mode&, ByVal updateCount&)`
Delphi	`daqDacWaveSetMode(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; mode:DaqDacWaveformMode; updateCount:DWORD)`
Parameters	
`handle`	Handle to the device for which to set the DAC waveform output mode
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`mode`	Specifies the desired DAC waveform output mode
`updateCount`	Sets the total sample update count
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetTrig, daqDacWaveSetFreq`
Program References	None
Used With	

daqDacWaveSetMode allows you to set the DAC waveform update mode for the specified device's DAC channel. This function allows the setting of the waveform output mode and the total number of DAC waveform samples to output.

The **mode** parameter defines the state in which the waveform is to be output. The mode values are defined as follows:

- **DdwmNShot** - Continue outputting waveform until **updateCount** number of samples have been output. Upon completion of the specified amount, automatically terminate and disarm the waveform output operation.
- **DdwmNShotRearm** - Continue outputting waveform until **updateCount** number of samples have been output. Upon completion of the specified amount, reset the **updateCount** and re-arm the DAC waveform output, using the previous configuration. Waveform output will then be restarted when the specified trigger event is detected. The automatic re-arming of the waveform output will continue until the waveform output is disarmed via the **daqDacWaveDisarm** function.
- **DdwmInfinite** - Continue outputting waveform indefinitely. Waveform output will continue until the **daqDacWaveDisarm** function is issued. **updateCount** is ignored.

The **updateCount** parameter defines the total number of samples in the waveform to be output.

daqDacWaveSetPredefWave

DLL Function	`daqDacWaveSetPredefWave(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacWaveType waveType, DWORD amplitude, DWORD offset, DWORD dutyCycle, DWORD phaseShift);`
C	`daqDacWaveSetPredefWave(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacWaveType waveType, DWORD amplitude, DWORD offset, DWORD dutyCycle, DWORD phaseShift);`
Visual BASIC	`VBdaqDacWaveSetPredefWave&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal waveType&, ByVal amplitude&, ByVal offset&, ByVal dutyCycle&, ByVal phaseShift&)`
Delphi	`daqDacWaveSetPredefWave(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; waveType:DaqDacWaveType; amplitude:DWORD; offset:DWORD; dutyCycle:DWORD; phaseShift:DWORD)`
Parameters	
`handle`	Handle to the device to setup a pre-defined waveform output
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`filename`	Specifies the optional predefined waveform output filename
`waveType`	Specifies the predefined waveform output type. Three types: 0 for `DdwtSine`, 1 for `DdwtSquare`, 2 for `DdwtTriangle`.
`amplitude`	Sets the peak-to-peak amplitude for which to generate the pre-defined waveform (in D/A counts 0 to 4095)
`offset`	Sets the offset for the pre-defined waveform (voltage level in D/A counts 0 to 4095)
`dutyCycle`	Sets the duty cycle (as a percentage) of the predefined waveform
`phaseShift`	Set the phase shift (in degrees) of the predefined waveform relative to other DAC channel
Returns	`DerrInvDacChan` - The DAC channel number doesn't exist `DerrInvDacParam` - Parameters were out of range `DerrInvPredefWave` - Predefined waveform is not supported `DerrMemAlloc` - Not enough memory was available to build the waveform `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetUserWave`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveSetPredefWave` allows you to specify a pre-defined waveform for DAC waveform output on the specified device channel. **`daqDacWaveSetMode`** is used to set the update rate and cycling mode for this waveform.

daqDacWaveSetTrig

DLL Function	`daqDacWaveSetTrig(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacTriggerSource triggerSource, BOOL rising);`
C	`daqDacWaveSetTrig(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, DaqDacTriggerSource triggerSource, BOOL rising);`
Visual BASIC	`VBdaqDacWaveSetTrig&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal triggerSource&, ByVal rising&)`
Delphi	`daqDacWaveSetTrig(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; triggerSource:DaqDacTriggerSource; rising:longbool)`
Parameters	
`handle`	Handle of the device for which to set DAC waveform triggering
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
`triggerSource`	Specifies the DAC output trigger source
`rising`	Boolean indicating the trigger source edge
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetMode`
Program References	None
Used With	

`daqDacWaveSetTrig` allows you to set the DAC waveform output trigger for the specified DAC channel. This function is used to setup the trigger event to initiate a DAC waveform output for the specified DAC channel.

The **`triggerSource`** parameter specifies the source of the event which will trigger the DAC waveform output. Currently, there are only two valid DAC waveform trigger events:

- **`DdtsImmediate`** - Trigger DAC waveform immediately upon execution of the **`daqDacWaveArm`** function. This trigger source is used to trigger the DAC waveform output immediately upon waveform configuration.
- **`DdtsSoftware`** - Trigger the DAC waveform upon execution of the **`daqDacWaveSoftTrig`** function. This trigger source requires that the **`daqDacWaveArm`** function be issued before the **`daqDacWaveSoftTrig`** function. This trigger source is used to trigger the waveform output from input from the user application.

The **`rising`** flag is currently ignored and is reserved for future use.

daqDacWaveSetUserWave

DLL Function	`daqDacWaveSetUserWave(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
C	`daqDacWaveSetUserWave(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
Visual BASIC	`VBdaqDacWaveSetUserWave&(ByVal handle&, ByVal deviceType&, ByVal chan&)`
Delphi	`daqDacWaveSetUserWave(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD)`
Parameters	
`handle`	Handle to the device for which the user - defined waveform is to be output
`deviceType`	Specifies the DAC device type
`chan`	Specifies the DAC device channel
Returns	`DerrInvDacChan` - The DAC channel number doesn't exist `DerrInvBuf` - A waveform buffer was not specified `DerrMemAlloc` - Not enough memory was available to build the waveform `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetPredefWave`
Program References	None
Used With	

`daqDacWaveSetUserWave` allows you to configure a user-defined buffer for DAC waveform output. Any arbitrary waveform can be built in an array. **`daqDacWaveSetUserWave`** can then be called by specifying a pointer to the beginning of the waveform, the size of the array, and the target DAC channel to send the waveform.

daqDacWaveSoftTrig

DLL Function	`daqDacWaveSoftTrig(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
C	`daqDacWaveSoftTrig(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan);`
Visual BASIC	`VBdaqDacWaveSoftTrig&(ByVal handle&, ByVal deviceType&, ByVal chan&)`
Delphi	`daqDacWaveSoftTrig(handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD)`
Parameters	
`handle`	Handle to the device for which to trigger the DAC waveform output
`deviceType`	Specifies the DAC type
`chan`	Specifies the DAC channel
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWaveSetTrig`
Program References	None
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWaveSoftTrig` allows you to software trigger the DAC waveform output on the specified device channel. The device channel must first have been configured for software triggering with the **`daqDacWaveSetTrig`** function prior to calling this function. DAC waveform trigger source must be **`DdtsSoftware`**.

daqDacWt

DLL Function	`daqDacWt(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, WORD dataVal);`
C	`daqDacWt(DaqHandleT handle, DaqDacDeviceType deviceType, DWORD chan, WORD dataVal);`
Visual BASIC	`VBdaqDacWt&(ByVal handle&, ByVal deviceType&, ByVal chan&, ByVal dataVal%)`
Delphi	`daqDacWt( handle:DaqHandleT; deviceType:DaqDacDeviceType; chan:DWORD; dataVal:WORD )`
Parameters	
`handle`	Handle to the device for which the Daq* channel value is to be updated
`deviceType`	Specifies the DAC type
`chan`	The D/A channel to output to Valid values: 0 - 1
`dataVal`	The value to output to the selected D/A channel Valid values: 0 -4095
Returns	`DerrInvChan` - Invalid channel `DerrInvDacVal` - Invalid data value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWtMany`
Program References	DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS, DACEX.PAS (Delphi)
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWt` outputs a voltage between 0 and 5 V to the specified 12-bit D/A channel. The voltage has a resolution of approximately 1.22 mV (5 V/4095).

Note: **`daqAdcSetTrig`** will configure the D/A channel 1 if an analog trigger is source selected for the A/D converter.

daqDacWtMany

DLL Function	`daqDacWtMany(DaqHandleT handle, DaqDacDeviceType *deviceTypes, PDWORD chans, PWORD dataVals, DWORD count);`
C	`daqDacWtMany(DaqHandleT handle, DaqDacDeviceType *deviceTypes, PDWORD chans, PWORD dataVals, DWORD count);`
Visual BASIC	`VBdaqDacWtMany&(ByVal handle&, deviceTypes&(), chans&(), dataVals%(), ByVal count&)`
Delphi	`daqDacWtMany(handle:DaqHandleT; deviceTypes:DaqDacDeviceTypeP; chans:PDWORD; dataVals:PWORD; count:DWORD)`
Parameters	
`handle`	Handle to the device for which the values of the Daq* channels are to be updated
`deviceTypes`	Specifies the DAC types
`chans`	Specifies the DAC channels
`dataVals`	The value to output to the D/A channel Valid values: `0 -4095`
`count`	
Returns	`DerrInvDacVal` - Invalid data value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDacWt`
Program References	DACEX1.C, DAQEX.FRM (VB)
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWtMany` outputs voltages between 0 and 5 V to all active 12-bit D/A channels. Each voltage has a resolution of approximately 1.22 mV (5 V/4095).

Note: **`daqAdcSetTrig`** will configure the D/A channel 1 if an analog trigger source is selected for the A/D converter.

daqDefaultErrorHandler

DLL Function	`daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);`
C	`daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);`
Visual BASIC	`VBdaqDefaultErrorHandler(ByVal handle&, ByVal errCode&)`
Delphi	`daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device to which the default error handler is to be attached.
`ErrCode`	The error code number of the detected error (see table *API Error Codes* at end of this chapter).
Returns	Nothing (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetLastError, daqProcessError, daqSetDefaultErrorHandler`
Program References	None
Used With	All devices

`daqDefaultErrorHandler` displays an error message and then exits the application program. When the Daq* library is loaded, it invokes the default error handler whenever it encounters an error. The error handler may be changed with **`daqSetErrorHandler`**.

daqFormatError

DLL Function	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
C	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
Visual BASIC	`VBdaqCalSelectInputSignal(ByVal handle&, ByVal input as DaqCalInputT )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; input: DaqCalInputT)`
Parameters	
`daqError`	Daq* Enhanced API error code
`msg`	Pointer to a string to return the error text
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqSetDefaultErrorHandler, daqSetErrorHandler, daqProcessError, daqGetLastError, daqDefaultErrorHandler`
Program References	None
Used With	All devices

daqFormatError returns the text-string equivalent for the specified error condition. The error condition is specified by the **daqError** parameter. The error text will be returned in the character string pointed to by the **msg** parameter. The character string space must have been previously allocated by the application before calling this function. The allocated character string should be, at minimum, 64 bytes in length.

For more information on specific error codes refer to the *API Error Codes* on page 3-83.

daqGetDeviceCount

DLL Function	`daqGetDeviceCount(PDWORD deviceCount);`
C	`daqGetDeviceCount(PDWORD deviceCount);`
Visual BASIC	`VBdaqGetDevice&(deviceCount&)`
Delphi	`daqGetDeviceCount( var deviceCount:DWORD )`
Parameters	
`deviceCount`	Pointer to which the device count is to be returned
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetDeviceList, daqGetDeviceProperties`
Program References	None
Used With	All devices

daqGetDeviceCount returns the number of currently configured devices. This function will return the number of devices currently configured in the system. The devices do not need to be opened for this function to operate properly. If the number returned does not seem appropriate, the device configuration list should be checked via the Daq* Configuration applet located in the Control Panel. Refer to the configuration section in your device's user manual for more details.

daqGetDeviceList

DLL Function	`daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);`
C	`daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);`
Visual BASIC	`VBdaqGetDeviceList(deviceList as DaqDeviceListT, deviceCount&)`
Delphi	`daqGetDeviceList( var deviceList: DaqDeviceListT; var deviceCount: DWORD)`
Parameters	
`deviceList`	Pointer to memory location to which the device list is to be returned
`deviceCount`	Number of devices returned in the device list
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetDeviceCount, daqGetDeviceProperties, daqOpen,`
Program References	None
Used With	All devices

`daqGetDeviceList` returns a list of currently configured devices. This function will return the device names in the **`deviceList`** parameter for the number of devices returned by the **`deviceCount`** parameter. Each **`deviceList`** entry contains a device name consisting of up to 64 characters. The device name can then be used with the **`daqOpen`** function to open the specific device.

If the number returned does not seem appropriate, the device configuration list should be checked via the Daq* Configuration applet located in the Control Panel. Refer to the configuration section in your device's user manual for more details.

daqGetDeviceProperties

DLL Function	`daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);`
C	`daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);`
Visual BASIC	`VBdaqGetDeviceProperties( daqName$, deviceProps as DaqDevicePropsT)`
Delphi	`daqGetDeviceProperties( daqName: string; var deviceProps: DaqDevicePropsT)`
Parameters	
`daqName`	Pointer to a character string representing the name of the device for which to retrieve properties
`deviceCount`	Number of devices returned in the device list
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetDeviceCount, daqGetDeviceList, daqOpen`
Program References	None
Used With	All devices

`daqGetDeviceProperties` returns the properties for the specified device. The device is specified by passing the name of the device in the **`daqName`** parameter. This name should be a valid name of a configured device. The properties for the device are returned in the **`deviceProps`** parameter. **`deviceProps`** is a pointer to user-allocated memory which will hold the device-properties structure. This memory must have been allocated before calling this function.

For detailed device-property structure layout, refer the to *Daq Device Properties Definition* table.

If this function fails, make sure the **`daqName`** parameter references a valid device which is currently configured. This can be checked via the Daq* Configuration applet located in the Control Panel. Refer to the configuration section in your device's user manual for more details.

daqGetDriverVersion

DLL Function	`daqGetDriverVersion(PDWORD version);`
C	`daqGetDriverVersion(PDWORD version);`
Visual BASIC	`VBdaqGetDriverVersion&(version&)`
Delphi	`daqGetDriverVersion(var version:DWORD)`
Parameters	
`version`	Pointer to the version number of the current device driver.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetHardwareInfo`
Program References	ERREX.PAS (Delphi)
Used With	All devices

`daqGetDriverVersion` allows you to get the revision level of the driver currently in use.

daqGetHardwareInfo

DLL Function	`daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);`
C	`daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);`
Visual BASIC	`VBdaqGetHardwareInfo&(ByVal handle&, ByVal whichInfo&, info As Variant)`
Delphi	`daqGetHardwareInfo(handle:DaqHandleT; whichInfo:DaqHardwareInfo; info:pointer)`
Parameters	
`handle`	Handle to the device
`whichInfo`	Specifies what type of device information to retrieve
`* info`	Pointer to the returned device information
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqGetDriverVersion, daqOpen`
Program References	DACEX.PAS, ERREX.PAS (Delphi)
Used With	All devices

`daqGetHardwareInfo` allows you to retrieve specific hardware information for the specified device. The device handle must be a valid device handle that is currently open. To open a device, see the **`daqOpen`** function.

daqGetLastError

DLL Function	`daqGetLastError(DaqHandleT handle, DaqError *errCode);`
C	`daqGetLastError(DaqHandleT handle, DaqError *errCode);`
Visual BASIC	`VBdaqGetLastError&(ByVal handle&, errCode&)`
Delphi	`daqGetLastError(handle:DaqHandleT; var errCode:DaqError): DaqError; stdcall; external DAQX_DLL; procedure daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device
`*errCode`	Returned last error code
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDefaultErrorHandler, daqProcessError, daqSetDefaultErrorHandler`
Program References	None
Used With	All devices

`daqGetLastError` allows you to retrieve the last error condition registered by the driver.

daqIOGet8255Conf

DLL Function	`daqIOGet8255Conf(DaqHandleT handle, BOOL portA, BOOL portB, BOOL portCHigh, BOOL portCLow, PDWORD config);`
C	`daqIOGet8255Conf(DaqHandleT handle, BOOL portA, BOOL portB, BOOL portCHigh, BOOL portCLow, PDWORD config);`
Visual BASIC	`VBdaqIOGet8255Conf&(ByVal handle&, ByVal portA&, ByVal portB&, ByVal portCHigh&, ByVal portCLow&, config&)`
Delphi	`daqIOGet8255Conf(handle:DaqHandleT; portA:longbool; portB:longbool; portCHigh:longbool; portCLow:longbool; var config:DWORD )`
Parameters	
`handle`	Handle to the device
`portA`	8255 port A value
`portB`	8255 port B value
`portCHigh`	8255 port C high nibble value
`portCLow`	8255 port C low nibble value
`config`	8255 current configuration
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqIORead, daqIOReadBit, daqIOWrite, daqIOWriteBit`
Program References	DIGEX1.C, DAQEX.FRM (VB), DIGEX.PAS (Delphi)
Used With	DaqBook100, DaqBook120, DaqBook200, DaqBoard100A, DaqBoard200A

`daqIOGet8255Conf` allows you to set/get the configuration for the specified 8255 device with the specified port configurations. The configuration is returned in the **`config`** parameter and will indicate the current configuration of the 8255. When set to **`TRUE`**, **`portA`**, **`portB`**, **`portCHigh`** and **`portCLow`** flags will configure the respective port as an input port. If the flag is set to **`FALSE`**, the port will be configured as an output.

daqIORead

DLL Function	`daqIORead(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, PDWORD value);`
C	`daqIORead(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, PDWORD value);`
Visual BASIC	`VBdaqIOReadBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, bitValue&)`
Delphi	`daqIOReadBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; var bitValue:longbool)`
Parameters	
`handle`	Handle to the device to perform the IO read
`devType`	IO Device type
`devPort`	IO port selection
`whichDevice`	IO device instance to read from
`whichExpPort`	IO device expansion port to read from
`value`	IO value read
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqIOReadBit, daqIOWrite, daqIOWriteBit`
Program References	DIGEX1.C, DAQEX.FRM (VB), DIGEX.PAS (Delphi)
Used With	All devices

`daqIORead` allows you to read the specified port on the selected device. The read operation will return the current state of the port in the **`value`** parameter. Normally, if the selected port is a byte-wide port, the port state will occupy the low-order byte of the **`value`** parameter. Digital IO channels for the port correspond to each bit within this low-order byte. If the bit is set, it indicates the channel is in a high state. If the bit is not set, the channel is indicated to be in a low state.

daqIOReadBit

DLL Function	`daqIOReadBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, PBOOL bitValue);`
C	`daqIOReadBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, PBOOL bitValue);`
Visual BASIC	`VBdaqIOReadBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, bitValue&)`
Delphi	`daqIOReadBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; var bitValue:longbool)`
Parameters	
`handle`	Handle to the device from which to perform the IO
`devType`	IO Device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO expansion port address
`bitNum`	IO port bit location to read
`bitValue`	IO port bit value (**`TRUE - high, FALSE - low`**)
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	**`daqIORead, daqIOWrite, daqIOWriteBit`**
Program References	DIGEX1.C, DAQEX.FRM (VB), DIGEX.PAS (Delphi)
Used With	All devices

`daqIOReadBit` allows you to read a specified bit on the selected device and port. The read operation will return the current state of the selected bit in the **`bitValue`** parameter. The selected bit (specified by the **`bitNum`** parameter) corresponds to the IO channel on the port which is to be read. The **`bitValue`** will be **`TRUE`** indicating a high state or **`FALSE`** indicating a low state.

daqIOWrite

DLL Function	`daqIOWrite(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD value);`
C	`daqIOWrite(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD value);`
Visual BASIC	`VBdaqIOWriteBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, ByVal bitValue&)`
Delphi	`daqIOWriteBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; bitValue:longbool)`
Parameters	
`handle`	Handle of the device to perform an IO write operation
`devType`	IO device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO device expansion port address
`value`	Value to write
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	**`daqIORead, daqIOWriteBit, daqIOReadBit`**
Program References	DIGEX1.C, DAQEX.FRM (VB), DIGEX.PAS (Delphi)
Used With	All devices

`daqIOWrite` allows you to write to the specified port on the selected device. The write operation will write the settings indicated in the **`value`** parameter to the selected port. The **`value`** written will depend on the width of the selected port. Normally, for byte-wide ports, only the low-order byte of the value parameter will be written. The IO channels for the port correspond to each bit within the value written. If the channel is to be driven to a high state, then the corresponding bit should be set. Likewise, if the channel is to be driven to a low state, then the corresponding bit should not be set.

daqIOWriteBit

DLL Function	`daqIOWriteBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, BOOL bitValue);`
C	`daqIOWriteBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, BOOL bitValue);`
Visual BASIC	`VBdaqIOWriteBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, ByVal bitValue&)`
Delphi	`daqIOWriteBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; bitValue:longbool)`
Parameters	
`handle`	Handle of the device to perform an IO write to
`devType`	IO device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO device expansion port address
`bitNum`	Bit number to write
`bitValue`	Bit value to write (`TRUE - high, FALSE - low`)
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqIOWrite, daqIORead, daqIOReadBit`
Program References	DIGEX1.C, DAQEX.FRM (VB), DIGEX.PAS (Delphi)
Used With	All devices

`daqIOWriteBit` allows you to write a specified bit on the selected device and port. The write operation will write the specified bit value to the bit selected. The selected bit, specified by the **`bitNum`** parameter, corresponds to the channel on the port for the IO to be driven. The **`bitValue`** parameter should be set to **`TRUE`** to drive the channel to a high state or **`FALSE`** indicating a low state.

daqOnline

DLL Function	`daqOnline(DaqHandleT handle, PBOOL online);`
C	`daqOnline(DaqHandleT handle, PBOOL online);`
Visual BASIC	`VBdaqOnline&(ByVal handle&, online&)`
Delphi	`daqOnline(handle: DaqHandleT; var online: longbool)`
Parameters	
`handle`	Handle of the device to test for online
`online`	Boolean indicating whether the device is currently online
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqOpen, daqClose`
Program References	ERREX.PAS (Delphi)
Used With	All devices

`daqOnline` allows you to determine if a device is online. The device **`handle`** must be a valid device handle which has been opened using the **`daqOpen`** function. The **`online`** parameter indicates the current online state of the device (**`TRUE`** - device online; **`FALSE`** - device not online).

daqOpen

DLL Function	`daqOpen(LPSTR daqName);`
C	`daqOpen(LPSTR daqName);`
Visual BASIC	`VBdaqOpen&(ByVal daqName$)`
Delphi	`daqOpen(devName: PChar)`
Parameters	
`daqName`	String representing the name of the device to be opened
Returns	A handle to the specified device (also, refer to *API Error Codes* on page 3-83)
See Also	`daqClose, daqOnline`
Program References	ADCEX1.C, DIGEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS, ADCEX.PAS (Delphi)
Used With	

daqOpen allows you to open an installed Daq* device for operation. The **daqOpen** function will initiate a session for the device name specified by the **daqName** parameter by opening the device, initializing it, and preparing it for further operation. The **daqName** specified must reference a currently configured device. See *Daq* Configuration* utility (in the *Installation* chapters) for more details on configuring devices and assigning device names.

daqOpen should be performed prior to any other operation performed on the device. This function will return a device handle that is used by other functions to reference the device. Once the device has been opened, the device handle should be used to perform subsequent operations on the device.

Most functions in this manual require a device handle in order to perform their operation. When the device session is complete, **daqClose** may be called with the device handle to close the device session.

daqProcessError

DLL Function	`daqProcessError(DaqHandleT handle, DaqError errCode);`
C	`daqProcessError(DaqHandleT handle, DaqError errCode);`
Visual BASIC	`VBdaqProcessError&(ByVal handle&, ByVal errCode&)`
Delphi	`daqProcessError(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device for which the specified error is to be processed.
`errCode`	Specifies the device error code to process
Returns	Refer to *API Error Codes* on page 3-83
See Also	`daqSetDefaultErrorHandler, daqGetLastError, daqDefaultErrorHandler`
Program References	None
Used With	All devices

daqProcessError allows an application to initiate an error for processing by the driver. This command can be used when it is desirable for the application to initiate processing for a device-defined error.

daqReadCalFile

DLL Function	`daqReadCalFile(DaqHandleT handle, LPSTR calfile);`
C	`daqReadCalFile(DaqHandleT handle, LPSTR calfile);`
Visual BASIC	`VBdaqReadCalFile&(ByVal handle&, ByVal calfile$)`
Delphi	`daqReadCalFile(handle:DaqHandleT; calfile:PChar)`
Parameters	
`handle`	Handle to the device for which to associate the calibration file.
`calfile`	The file name with optional path information of the calibration file. If calfile is NULL or empty (“”), the default calibration file DAQBOOK.CAL will be read.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83) `DerrInvCalfile` - Error occurred while opening or reading calibration file
See Also	`daqCalSetup, daqCalConvert, daqCalSetupConvert`
Program References	None
Used With	

`daqReadCalFile` is the initialization function for reading in the calibration constants from the calibration text file. This function (usually called once at the beginning of a program) will read all the calibration constants from the specified file. The **`calfile`** parameter specifies the path\filename of the calibration file to read.

If calibration constants for a specific channel number and gain setting are not contained in the file, ideal calibration constants will be used—essentially performing no calibration for that channel. If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and a non-zero error code will be returned by the **`daqReadCalFile`** function.

daqSetDefaultErrorHandler

DLL Function	`daqSetDefaultErrorHandler(DaqErrorHandlerFPT handler);`
C	`daqSetDefaultErrorHandler(DaqErrorHandlerFPT handler);`
Visual BASIC	`VBdaqSetDefaultErrorHandler&(ByVal handler&)`
Delphi	`daqSetDefaultErrorHandler(handler:DaqErrorHandlerFPT)`
Parameters	
`handler`	Pointer to a user-defined error handler function.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqDefaultErrorHandler, daqGetLastError, daqProcessError, daqSetErrorHandler`
Program References	ERREX.PAS (Delphi)
Used With	All devices

`daqSetDefaultErrorHandler` allows you to set the driver to use the default error handler specified for all devices.

daqSetErrorHandler

DLL Function	`daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);`
C	`daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);`
Visual BASIC	`VBdaqSetErrorHandler&(ByVal handle&, ByVal handler&)`
Delphi	`daqSetErrorHandler(handle:DaqHandleT; handler:DaqErrorHandlerFPT)`
Parameters	
`handle`	Handle to the device to which to attach the specified error handler
`handler`	Pointer to a user defined error handler function.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqSetDefaultErrorHandler, daqDefaultErrorHandler, daqGetLastError, daqProcessError`
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS (Delphi)
Used With	

`daqSetErrorHandler` specifies the routine to call when an error occurs in any command. The default routine displays a message and then terminates the program. If this is not desirable, use this command to specify your own routine to be called when errors occur. If you want no action to occur when a command error is detected, use this command with a null (0) parameter. The default error routine is **`daqDefaultHandler`**.

daqSetOption

DLL Function	`daqSetOption(DaqHandleT handle, DWORD chan, DWORD flags, DaqOptionType optionType, FLOAT optionValue);`
C	`daqSetOption(DaqHandleT handle, DWORD chan, DWORD flags, DaqOptionType optionType, FLOAT optionValue);`
Visual BASIC	`VBdaqSetOption&(ByVal handle&, ByVal chan&, ByVal flags&, ByVal optionType&, ByVal optionValue!)`
Delphi	`daqSetOption(Handle:DaqHandleT; chan:DWORD; flags:DWORD; optionType:DaqOptionType; optionValue:FLOAT)`
Parameters	
`handle`	The handle to the device for which to set the option
`chan`	The channel number on the device for which the option is to be set
`flags`	Flags specifying the options to use.
`optionType`	Specifies the type of option.
`optionValue`	The value of the option to set
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-44)
See Also	`daqAdcExpSetChanOption,`
Program References	None
Used With	All devices

`daqSetOption` allows the setting of options for a device's channel/signal path configuration.

- The **chan** parameter specifies which channel the option applies to.
- The **optionType** specifies the type of option to apply to the channel.
- The **optionValue** parameter specifies the value of the option.
- The **flags** parameter specifies how the option is to be applied.

For more information on the options and their valid settings, refer to the *Option Value and Option Type* tables.

daqSetTimeout

DLL Function	`daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);`
C	`daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);`
Visual BASIC	`VBdaqSetTimeout&(ByVal handle&, ByVal mSecTimeout&)`
Delphi	`daqSetTimeout(handle:DaqHandleT; mSecTimeout:DWORD)`
Parameters	
`handle`	Handle to the device for which the event time-out is to be set
`mSecTimeout`	Specifies time-out (ms) for events
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqWaitForEvent, daqWaitForEvents`
Program References	None
Used With	All devices

`daqSetTimeout` allows you to set the time-out for waiting on a single event or multiple events. This function can be used in conjunction with the **`daqWaitForEvent`** and **`daqWaitForEvents`** functions to specify a maximum amount of time to wait for the event(s) to be satisfied.

The **`mSecTimeout`** parameter specifies the maximum amount of time (in milliseconds) to wait for the event(s) to occur. If the event(s) do not occur within the specified time-out, the **`daqWaitForEvent`** and/or **`daqWaitForEvents`** will return.

daqTest

DLL Function	`daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);`
C	`daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);`
Visual BASIC	`VBdaqTest&(ByVal handle&, ByVal command&, ByVal count&, cmdAvailable&, result&)`
Delphi	[not supported]
Parameters	
`handle`	Handle to the device for which the test is to be performed
`command`	Specifies the type of test to be run
`count`	Optional parameter which specifies the length of the test
`cmdAvailable`	Return Boolean indicating the availability of the test for the device
`result`	Pointer to the test result field
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqOpen`
Program References	None
Used With	All devices

`daqTest` allows you to test a Daq* device for specific functionality. Test types vary, and test results are based on the type of test requested. Tests can only be performed on valid, opened Daq* devices. If there are problems with the test, be sure to check the device for proper configuration and that the device is powered-on and properly connected.

The **`command`** parameter specifies the test to run. There are two main types of tests: resource and performance.

Resource tests are pass/fail and are useful in determining if the device has the appropriate resources to function efficiently. If one or more of the resource tests fail, the Daq Configuration utility (found in the operating system's Control Panel) may be used to change the resource settings related to the problem. Valid resource test types are defined as follows:

- **`DtsBaseAddressValid`** - This test will indicate if there is a problem communicating with the device at its currently specified base address. A non-zero in the **`result`** parameter will indicate a failed condition.
- **`DtsInterruptLevelValid`** - This test will indicate if there is a problem with performing acquisitions using interrupts. A non-zero in the **`result`** parameter will indicate a failed condition. If this is the case, the interrupts may not be properly configured (if the device is a DaqBook, the LPT interrupts may not be enabled on the system) or an interrupt conflict exists with another device.
- **`DtsDmaChannelValid`** - (DaqBoard only) This test will indicate if there is a problem with performing acquisitions through DMA transfers with the currently configured DMA channel for the device. A non-zero in the **`result`** parameter will indicate a failed condition. If this is the case, DMA may not be enabled for the device or a conflict may exist with another device.

Performance tests measure the speed at which certain operations can be performed on the device. In general, the performance test results indicate the maximum rate at which the operation can be performed on the device. The valid performance test types are defined as follows:

- **`DtsAdcFifoInputSpeed`** - This test will determine the maximum rate at which analog input can be acquired and transferred to system memory. Analog input performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsDacFifoOutputSpeed`** - (DaqBoard only) This test will determine the maximum rate at which analog output data can be read from system memory and transferred to the device's DAC FIFO. Analog output performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsIOInputSpeed`** - This test will determine the maximum rate at which digital input can be read from the device's DIO port and transferred to system memory. Digital input performance results will be returned in the **`result`** parameter with units of bytes/second.
- **`DtsIOOutputSpeed`** - This test will determine the maximum rate at which digital output can be read from system memory and output to the device's DIO port. Digital output performance results will be returned in the **`result`** parameter with units of bytes/second.

The **`cmdAvailable`** parameter is a pointer to a Boolean value that indicates whether or not the specified test is available for the device.

The **`count`** parameter can be used to indicate the duration or length of the test. For instance, a resource test will be run **`count`** times; and if any one iteration of the test fails, it will indicate an overall failure of the test. For a performance test, the **`count`** parameter will indicate the number of times to run the test, and the test result will be an average of all the tests performed.

daqWaitForEvent

DLL Function	`daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);`
C	`daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);`
Visual BASIC	`VBdaqWaitForEvent&(ByVal handle&, ByVal daqEvent&)`
Delphi	`daqWaitForEvent(handle:DaqHandleT; daqEvent:DaqTransferEvent)`
Parameters	
`handle`	Handle of the device for which to wait of the specified event
`daqEvent`	Specifies the event to wait on
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqWaitForEvents, daqSetTimout`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqWaitForEvent` allows you to wait on a specific Daq* event to occur on the specified device. This function will not return until the specified event has occurred or the wait has timed out—whichever comes first. The event time-out can be set with the **`daqSetTimout`** function. See the *Transfer Event Definitions* table for event definitions.

daqWaitForEvents

DLL Function	`daqWaitForEvents(DaqHandleT *handles, DaqTransferEvent *daqEvents, DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);`
C	`daqWaitForEvents(DaqHandleT *handles, DaqTransferEvent *daqEvents, DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);`
Visual BASIC	`VBdaqWaitForEvents&(handles&(), daqEvents&(), ByVal eventCount&, eventSet&(), ByVal waitMode&)`
Delphi	`daqWaitForEvents(handles:DaqHandlePT; daqEvents:DaqTransferEventP; eventCount:DWORD; eventSet:PLONGBOOL; waitMode:DaqWaitMode)`
Parameters	
`*handles`	Pointer to an array of handles which represent the list of device on which to wait for the events
`*daqEvents`	Pointer to an array of events which represents the list of events to wait on
`eventCount`	Number of defined events to wait on
`*eventSet`	Pointer to an array of Booleans indicating if the events have been satisfied.
`waitMode`	Specifies the mode for the wait
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqWaitForEvent, daqSetTimeout`
Program References	None
Used With	All devices

`daqWaitForEvents` allows you to wait on specific Daq* events to occur on the specified devices. This function will wait on the specified events and will return based upon the criteria selected with the **`waitMode`** parameter. A time-out for all events can be specified using the **`daqSetTimeout`** command.

Events to wait on are specified by passing an array of event definitions in the **`events`** parameter. The number of events is specified with the **`eventCount`** parameter. See the *Transfer Event Definitions* table for **`events`** parameter definitions. Also see the *Transfer Event Wait Mode Definitions* table for **`waitMode`** parameter definitions.

daqZeroConvert

DLL Function	`daqZeroConvert(PWORD counts, DWORD scans);`
C	`daqZeroConvert(PWORD counts, DWORD scans);`
Visual BASIC	`VBdaqZeroConvert&(counts%(), ByVal scans&)`
Delphi	`daqZeroConvert(counts:PWORD; scans:DWORD)`
Parameters	
`counts`	The raw data from one or more scans.
`scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqZeroSetup, daqZeroSetupConvert, daqZeroDbk19`
Program References	None
Used With	All devices

`daqZeroConvert` compensates one or more scans according to the previously called **`daqZeroSetup`** function. This function will modify the array of data passed to it.

daqZeroDBK19

DLL Function	`daqZeroDbk19(BOOL zero);`
C	`daqZeroDbk19(BOOL zero);`
Visual BASIC	`VBdaqZeroDbk19&(ByVal zero&)`
Delphi	`daqZeroDbk19(zero:longbool)`
Parameters	
`zero`	If non-zero will enable auto zero compensation in the `daqCvtTC`... functions
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqZeroSetup, daqZeroConvert, daqZeroSetupConvert, daqCvtTCSetup, daqCvtTCConvert, daqcvtTcSetupConvert`
Program References	None
Used With	All devices

`daqZeroDBK19` will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the DBK19. When enabled, the thermocouple conversion functions will require a CJC zero reading and a TC zero reading to precede the actual CJC and TC reading. This can easily be done by configuring the scan group to read:

- channel 1 using the DBK19 CJC gain code (CJC zero)
- channel 1 using the gain code of the connected TC (TC zero)
- channel 0 using the DBK19 CJC gain code (CJC)
- and finally, the thermocouple channels using the gain code of the connected thermocouples.

Note: the offset of the real CJC value should be specified (not the offset of the CJC zero) when calling the thermocouple linearization setup functions.

daqZeroSetup

DLL Function	`daqZeroSetup(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings);`
C	`daqZeroSetup(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings);`
Visual BASIC	`VBdaqZeroSetup&(ByVal nscan&, ByVal zeroPos&, ByVal readingsPos&, ByVal nReadings&)`
Delphi	`daqZeroSetup(nscan:DWORD; zeroPos:DWORD; readingsPos:DWORD; nreadings:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan.
`zeroPos`	The position of the zero reading within the scan
`readingsPos`	The position of the readings to be zeroed within the scan.
`nReadings`	The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqZeroConvert, daqZeroSetupConvert, daqZeroDbk19`
Program References	None
Used With	All devices

`daqZeroSetup` configures the location of the shorted channel and the channels to be zeroed within a scan, the size of the scan, and the number of readings to zero. However, this function does not do the actual conversion. A non-zero return value indicates an invalid parameter error.

daqZeroSetupConvert

DLL Function	`daqZeroSetupConvert(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings, PWORD counts, DWORD scans);`
C	`daqZeroSetupConvert(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings, PWORD counts, DWORD scans);`
Visual BASIC	`VBdaqZeroSetupConvert&(ByVal nscan&, ByVal zeroPos&, ByVal readingsPos&, ByVal nReadings&, counts%(), ByVal scans&)`
Delphi	`daqZeroSetupConvert(nscan:DWORD; zeroPos:DWORD; readingsPos:DWORD; nreadings:DWORD; counts:PWORD; scans:DWORD)`
Parameters	
`nscan`	The number of readings in a single scan.
`zeroPos`	The position of the zero reading within the scan
`readingsPos`	The position of the readings to be zeroed within the scan.
`nReadings`	The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading.
`counts`	The raw data from one or more scans.
`scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 3-83)
See Also	`daqZeroSetup, daqZeroConvert, daqZeroDbk19`
Program References	None
Used With	All devices

`daqZeroSetupConvert` performs both the setup and convert steps with one call. This is useful when the zero compensation needs to be performed multiple times because data was read from channels at different gains or from different boards.

API Reference Tables

These tables provide information for programming with the Daq* Application Programming Interface. Information includes channel identification and error codes, as well as valid parameter values and descriptions. The tables are organized as follows:

API Parameter Reference Tables		
Table Title	**Sub-Title/Parameter/Description**	**Page**
A/D Channel Descriptions	Identifies Daq* local and expansion channel numbering scheme	3-74
Daq Device Property Definitions - `daqGetDeviceProperties`	Identifies the format (`DWORD`, `STRING`, or `FLOAT`) for property parameters	3-74
Digital I/O Port Connection	Identifies Daq* local and expansion digital port numbering scheme	3-75
Event-Handling Definitions	Transfer Event Definitions - `DaqTransferEvent` Transfer Event Wait Mode Definitions - `DaqWaitMode`	3-76
Hardware Information Definitions	Hardware Information Selector Definitions - `DaqHardwareInfo` Hardware Version Definitions - `DaqHardwareVersion`	3-76
DBK Card Definitions	DBK Card Expansion Bank Definitions - `DaqAdcExpType` Dbk Card Option Value Definitions - `DaqChanOptionValue` Dbk Card Option Type Selector Definitions - `DaqChanOptionType`	3-76
ADC Gain Definitions	Identifies gain codes for Daq* base unit and several DBKs	3-77
ADC Trigger Source Definitions	`DaqAdcTriggerSource` `DaqEnhTrigSensT`	3-78
ADC Miscellaneous Definitions	ADC Flag Definitions - `DaqAdcFlag` Frequency vs Period - `DaqAdcRateMode` ADC Acquisition Mode Definitions - `DaqAdcAcqMode` ADC Transfer Mask Definitions - `DaqAdcTransferMask` ADC Clock Source Definitions - `DaqAdcClockSource` ADC File Open Mode Definitions - `DaqAdcOpenMode` ADC Acquisition/Transfer Active Flag Definitions - `DaqAdcActiveFlag` ADC Acquisition State - `DaqAdcAcqState` ADC BufferTransfer Mask- `DaqAdcBufferXferMask`	3-78
DAC Definitions	DAC Device Type Definitions - `DaqDacDeviceType` DAC Output Mode Definitions - `DaqDacOutputMode` DAC Trigger Source Definitions - `DaqDacTriggerSource` DAC Clock Source Definitions - `DaqDacClockSource` DAC Waveform Mode Definitions - `DaqDacWaveformMode` DAC Predefined Waveform Type Definitions - `DaqDacWaveType` DAC Transfer Mask Definitions - `DaqDacTransferMask` DAC Waveform/Transfer Active Flag Definitions - `DaqDacActiveFlag`	3-79
Data Conversion Definitions	Software Calibration Type Definitions - `DcalType` RTD Type Definitions - `RtdType` Thermocouple Type Definitions - `TCType`	3-79
WBK Card Definitions	WBK Option Values - `DaqChanOptionValue` WBK Channel Options - `DaqAdcExpType` WBK Module Option-Types - `DaqOptionType`	3-80
General I/O Definitions	I/O Device Type Definitions - `DaqIODeviceType` I/O Operation Code Definitions - `DaqIOOperationCode` I/O Operation Code Definitions - `DaqIOExpansionPort` DAC Transfer Mask Definitions - `DaqIOTransferMask` I/O Operation Code Definitions - `DaqIOEventCode` DAC Transfer Active Flag Definitions - `DaqIOActiveFlag` I/O Port Type Definitions - `DaqIODevicePort`	3-81
9513 Counter/Timer Definitions	Time-of-Day Definitions - `Daq9513TimeOfDay` Count Source Definitions - `Daq9513CountSource` Output Control Definitions - `Daq9513OutputControl` Gating Control Definitions - `Daq9513GatingControl` Multiple Counter Command Definitions - `Daq9513MultCtrCommand`	3-82
DaqTest Command Definitions	`DaqTestCommand`	3-82
Calibration Input Signal Sources	`DaqCalInputT` `DaqCalTableTypeT`	3-82
API Error Codes	Identifies API errors by number and description	3-83

A/D Channel Descriptions

A/D Channel	Source
0 to 15	Local channels 0 to 15
16 to 31	Channels 0 to 15 of A/D expansion card 0
32 to 47	Channels 0 to 15 of A/D expansion card 1
48 to 63	Channels 0 to 15 of A/D expansion card 2
64 to 79	Channels 0 to 15 of A/D expansion card 3
80 to 95	Channels 0 to 15 of A/D expansion card 4
96 to 111	Channels 0 to 15 of A/D expansion card 5
112 to 127	Channels 0 to 15 of A/D expansion card 6
128 to 143	Channels 0 to 15 of A/D expansion card 7
144 to 159	Channels 0 to 15 of A/D expansion card 8
160 to 175	Channels 0 to 15 of A/D expansion card 9
176 to 191	Channels 0 to 15 of A/D expansion card 10
192 to 207	Channels 0 to 15 of A/D expansion card 11
208 to 223	Channels 0 to 15 of A/D expansion card 12
224 to 239	Channels 0 to 15 of A/D expansion card 13
240 to 255	Channels 0 to 15 of A/D expansion card 14
256 to 271	Channels 0 to 15 of A/D expansion card 15
272	High-speed digital I/O (DaqBook/100, DaqBook/200, DaqBoard/100A or DaqBoard/200A)
Note: In differential mode, only (sub)channels 0 to 7 are valid.	

Daq Device Property Definitions - `daqGetDeviceProperties`

Property	Description	Format
`deviceType`	Main Chassis Device Type Definition	`DWORD`
`basePortAddress`	Port Address (ISA Addr, LPT Port, etc)	`DWORD`
`dmaChannel`	DMA Channel (if applicable)	`DWORD`
`protocol`	Interface Protocol	`DWORD`
`alias`	Device Alias Name	`STRING`
`maxAdChannels`	Maximum A/D channels (with full expansion)	`DWORD`
`maxDaChannels`	Maximum D/A channels (with full expansion)	`DWORD`
`maxDigInputBits`	Maximum Dig. Inputs (with full expansion)	`DWORD`
`maxDigOutputBits`	Maximum Dig. Outputs (with full expansion)	`DWORD`
`maxCtrChannels`	Maximum Counter/Timers (with full expansion)	`DWORD`
`mainUnitAdChannels`	Maximum Main Unit A/D channels (no expansion)	`DWORD`
`mainUnitDaChannels`	Maximum Main Unit D/A channels (no expansion)	`DWORD`
`mainUnitDigInputBits`	Maximum Main Unit Digital Inputs (no expansion)	`DWORD`
`mainUnitDigOutputBits`	Maximum Main Unit Digital Outputs (no expansion)	`DWORD`
`mainUnitCtrChannels`	Maximum Main Unit Counter/Timer channels (no exp.)	`DWORD`
`adFifoSize`	A/D on-board FIFO Size	`DWORD`
`daFifoSize`	D/A on-board FIFO Size	`DWORD`
`adResolution`	Maximum A/D Converter Resolution	`DWORD`
`daResolution`	Maximum D/A Converter Resolution	`DWORD`
`adMinFreq`	Minimum A/D Conversion Scan Frequency (Hz)	`FLOAT`
`adMaxFreq`	Maximum A/D Conversion Scan Frequency (Hz)	`FLOAT`
`daMinFreq`	Minimum D/A Output Update Frequency (Hz)	`FLOAT`
`daMaxFreq`	Maximum D/A Output Update Frequency (Hz)	`FLOAT`

Digital I/O Port Connection

Base Unit		
Description	**Value**	**Address Select Jumper Location**
`Ddp4BitIO`	`83h`	Connector P1
`DdpLocalA`	`10h`	Connector P2 Port A
`DdpLocalB`	`11h`	Connector P2 Port B
`DdpLocalC`	`12h`	Connector P2 Port C
`DdpLocalCHigh`	`B2h`	Connector P2 Port C High Nibble
`DdpLocalCLow`	`92h`	Connector P2 Port C Low Nibble

Expansion Unit Address A		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21**
`DdpExp0A`	`60h`	Dig Exp Chan 0 Port A / (P2 A)
`DdpExp0B`	`61h`	Dig Exp Chan 0 Port B / (P2 A)
`DdpExp0C`	`62h`	Dig Exp Chan 0 Port C / (P2 A)
`DdpExp0High`	`E2h`	Dig Exp Chan 0 Port C High Nibble / (P2 A)
`DdpExp0Low`	`C2h`	Dig Exp Chan 0 Port C Low Nibble / (P2 A)
`DdpExp1A`	`64h`	Dig Exp Chan 1 Port A / (P3 A)
`DdpExp1B`	`65h`	Dig Exp Chan 1 Port B / (P3 A)
`DdpExp1C`	`66h`	Dig Exp Chan 1 Port C / (P3 A)
`DdpExp1CHigh`	`E6h`	Dig Exp Chan 1 Port C High Nibble / (P3 A)
`DdpExp1Low`	`C6h`	Dig Exp Chan 1 Port C Low Nibble / (P3 A)

Expansion Unit Address B		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
`DdpExp2A`	`68h`	Dig Exp Chan 2 Port A / (P2 B)
`DdpExp2B`	`69h`	Dig Exp Chan 2 Port B / (P2 B)
`DdpExp2C`	`6Ah`	Dig Exp Chan 2 Port C / (P2 B)
`DdpExp2CHigh`	`EAh`	Dig Exp Chan 2 Port C High Nibble / (P2 B)
`DdpExp2Low`	`CAh`	Dig Exp Chan 2 Port C Low Nibble / (P2 B)
`DdpExp3A`	`6Ch`	Dig Exp Chan 3 Port A / (P3 B)
`DdpExp3B`	`6Dh`	Dig Exp Chan 3 Port B / (P3 B)
`DdpExp3C`	`6Eh`	Dig Exp Chan 3 Port C / (P3 B)
`DdpExp3CHigh`	`EEh`	Dig Exp Chan 3 Port C High Nibble / (P3 B)
`DdpExp3Low`	`CEh`	Dig Exp Chan 3 Port C Low Nibble / (P3 B)

Expansion Unit Address C		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
`DdpExp4A`	`70h`	Dig Exp Chan 4 Port A / (P2 C)
`DdpExp4B`	`71h`	Dig Exp Chan 4 Port B / (P2 C)
`DdpExp4C`	`72h`	Dig Exp Chan 4 Port C / (P2 C)
`DdpExp4CHigh`	`F2h`	Dig Exp Chan 4 Port C High Nibble / (P2 C)
`DdpExp4Low`	`D2h`	Dig Exp Chan 4 Port C Low Nibble / (P2 C)
`DdpExp5A`	`74h`	Dig Exp Chan 5 Port A / (P3 C)
`DdpExp5B`	`75h`	Dig Exp Chan 5 Port B / (P3 C)
`DdpExp5C`	`76h`	Dig Exp Chan 5 Port C / (P3 C)
`DdpExp5CHigh`	`F6h`	Dig Exp Chan 5 Port C High Nibble / (P3 C)
`DdpExp5Low`	`D6h`	Dig Exp Chan 5 Port C Low Nibble / (P3 C)

Expansion Unit Address D		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
`DdpExp6A`	`78h`	Dig Exp Chan 6 Port A / (P2 D)
`DdpExp6B`	`79h`	Dig Exp Chan 6 Port B / (P2 D)
`DdpExp6C`	`7Ah`	Dig Exp Chan 6 Port C / (P2 D)
`DdpExp6CHigh`	`FAh`	Dig Exp Chan 6 Port C High Nibble / (P2 D)
`DdpExp6Low`	`DAh`	Dig Exp Chan 6 Port C Low Nibble / (P2 D)
`DdpExp7A`	`7Ch`	Dig Exp Chan 7 Port A / (P3 D)
`DdpExp7B`	`7Dh`	Dig Exp Chan 7 Port B / (P3 D)
`DdpExp7C`	`7Eh`	Dig Exp Chan 7 Port C / (P3 D)
`DdpExp7CHigh`	`FEh`	Dig Exp Chan 7 Port C High Nibble / (P3 D)
`DdpExp7Low`	`DEh`	Dig Exp Chan 7 Port C Low Nibble / (P3 D)

Event-Handling Definitions

Transfer Event Definitions - `daqTransferEvent`	
DteAdcData	0
DteAdcDone	1
DteDacData	2
DteDacDone	3
DteIOData	4
DteIODone	5

Transfer Event Wait Mode Definitions - `daqWaitMode`	
DwmNoWait	0
DwmWaitForAny	1
DwmWaitForAll	2

Hardware Information Definitions

Hardware Information Selector Definitions - `daqHardwareInfo`

Definition	Value
DhiHardwareVersion	0
DhiProtocol	1
DhiAdcBits	2
DhiADmin	3
DhiADmax	4

Hardware Version Definitions - `daqHardwareVersion`

Definition	Value
DaqBook100	0
DaqBook112	1
DaqBook120	2
DaqBook200	3
DaqBook216	4
DaqBoard100	5
DaqBoard112	6
DaqBoard200	7
DaqBoard216	8
Daq112	9
Daq216	10
WaveBook512	11
WaveBook516	12
TempBook66	13

DBK Card Definitions

DBK Card Expansion Bank Definitions - `daqAdcExpType`		
DaetNotDefined	0	Bank is unknown or undefine the bank
DaetDbk50	1	Dbk50 option
DaetDbk5	2	Dbk5 option
DaetDbk2	3	Dbk2 option
DaetDbk4	4	Dbk4 option
DaetDbk7	5	Dbk7 option

DBK Card Option Type Selector Definitions - `daqChanOptionType`		
DcotDbk4MaxFreq	0	
DcotDbk4SetBaseline	1	
DcotDbk4Excitation	2	
DcotDbk4Clock	3	
DcotDbk4Gain	4	internally used by `daqAdcSetScan`
DcotDbk7Slope	0	
DcotDbk7DebounceTime	1	
DcotDbk7MinFreq	2	
DcotDbk7MaxFreq	3	
DcotDbk50Gain	0	internally used by `daqAdcSetScan`

DBK Card Option Value Definitions - `daqChanOptionValue`	
Dbk4 cutoff frequencies for `DcotMaxFreq` option type	
DcovDbk4Freq18000Hz	0
DcovDbk4Freq9000Hz	1
DcovDbk4Freq4500Hz	2
DcovDbk4Freq2250Hz	3
DcovDbk4Freq1125Hz	4
DcovDbk4Freq563Hz	5
DcovDbk4Freq281Hz	6
DcovDbk4Freq141Hz	7
Dbk4 set baseline for `DcotSetBaseline` option type	
DcovDbk4BaselineNever	0
DcovDbk4BaselineOneShot	1
Dbk7 debounce times for `DcotDebounceTime` option type	
DcovDbk7DebounceNone	0
DcovDbk7Debounce600us	1
DcovDbk7Debounce2500us	2
DcovDbk7Debounce10ms	3

ADC Gain Definitions

Base Unit	
DgainX1	0
DgainX2	1
DgainX4	2
DgainX8	3

DBK4-Filter	
Dbk4FilterX1	0
Dbk4FilterX10	1
Dbk4FilterX100	2
Dbk4FilterX1000	3
Dbk4FilterX2	4
Dbk4FilterX20	5
Dbk4FilterX200	6
Dbk4FilterX2000	7
Dbk4FilterX4	8
Dbk4FilterX40	9
Dbk4FilterX400	10
Dbk4FilterX4000	11
Dbk4FilterX8	12
Dbk4FilterX80	13
Dbk4FilterX800	14
Dbk4FilterX8000	15
DBK4-Bypass	
Dbk4BypassX1_583	0
Dbk4BypassX15_83	1
Dbk4BypassX158_3	2
Dbk4BypassX1583	3
Dbk4BypassX3_166	4
Dbk4BypassX31_66	5
Dbk4BypassX316_6	6
Dbk4BypassX3166	7
Dbk4BypassX6_332	8
Dbk4BypassX63_32	9
Dbk4BypassX633_2	10
Dbk4BypassX6332	11
Dbk4BypassX12_664	12
Dbk4BypassX126_64	13
Dbk4BypassX1266_4	14
Dbk4BypassX12664	15

DBK7	
Dbk7X1	0

DBK8	
Dbk8X1	0

DBK9	
Dbk9VoltageA	0
Dbk9VoltageB	1
Dbk9VoltageC	2
Dbk9VoltageD	3

DBK12	
Dbk12X1	0
Dbk12X2	1
Dbk12X4	2
Dbk12X8	3
Dbk12X16	7
Dbk12X32	11
Dbk12X64	15

DBK13	
Dbk13X1	0
Dbk13X10	1
Dbk13X100	2
Dbk13X1000	3
Dbk13X2	4
Dbk13X20	5
Dbk13X200	6
Dbk13X2000	7
Dbk13X4	8
Dbk13X40	9
Dbk13X400	10
Dbk13X4000	11
Dbk13X8	12
Dbk13X80	13
Dbk13X800	14
Dbk13X8000	15

DBK14 Bipolar	
Dbk14BiCJC	Dbk13X2
Dbk14BiTypeJ	Dbk13X100
Dbk14BiTypeK	Dbk13X100
Dbk14BiTypeT	Dbk13X200
Dbk14BiTypeE	Dbk13X40
Dbk14BiTypeN28	Dbk13X400
Dbk14BiTypeN14	Dbk13X100
Dbk14BiTypeS	Dbk13X200
Dbk14BiTypeR	Dbk13X200
Dbk14BiTypeB	Dbk13X400
DBK14 Unipolar	
Dbk14UniCJC	Dbk13X4
Dbk14UniTypeJ	Dbk13X200
Dbk14UniTypeK	Dbk13X200
Dbk14UniTypeT	Dbk13X400
Dbk14UniTypeE	Dbk13X100
Dbk14UniTypeN28	Dbk13X800
Dbk14UniTypeN14	Dbk13X200
Dbk14UniTypeS	Dbk13X400
Dbk14UniTypeR	Dbk13X400
Dbk14UniTypeB	Dbk13X800

DBK15 Bipolar	
Dbk15BiX1	0
Dbk15BiX2	1
DBK15 Unipolar	
Dbk15UniX1	2
Dbk15UniX2	3

DBK16	
Dbk16ReadBridge	0
Dbk16SetOffset	1
Dbk16SetScalingGain	2
Dbk16SetInputGain	3

DBK18	
Dbk18X1	0

DBK19 Bipolar	
Dbk19BiCJC	0
Dbk19BiTypeJ	1
Dbk19BiTypeK	1
Dbk19BiTypeT	2
Dbk19BiTypeE	0
Dbk19BiTypeN28	3
Dbk19BiTypeN14	1
Dbk19BiTypeS	3
Dbk19BiTypeR	2
Dbk19BiTypeB	3
DBK19 Unipolar	
Dbk19UniCJC	1
Dbk19UniTypeJ	2
Dbk19UniTypeK	2
Dbk19UniTypeT	3
Dbk19UniTypeE	1
Dbk19UniTypeN28	3
Dbk19UniTypeN14	2
Dbk19UniTypeS	3
Dbk19UniTypeR	3
Dbk19UniTypeB	3

DBK42	
Dbk42X1	0

DBK43/43A	
Dbk43ReadBridge	0
Dbk43SetOffset	1
Dbk43SetScalingGain	2
Dbk43SetInputGain	3

DBK44	
Dbk44X1	0

DBK50	
Dbk50Range0	0
Dbk50Range10	1
Dbk50Range100	2
Dbk50Range300	3

DBK51	
Dbk51Range	0
Dbk51Range100mV	1
Dbk51Range1	2
Dbk51Range10	3

DBK52 Bipolar	
Dbk52BiCJC	Dbk19BiCJC
Dbk52BiTypeJ	Dbk19BiTypeJ
Dbk52BiTypeK	Dbk19BiTypeK
Dbk52BiTypeT	Dbk19BiTypeT
Dbk52BiTypeE	Dbk19BiTypeE
Dbk52BiTypeN28	Dbk19BiTypeN28
Dbk52BiTypeN14	Dbk19BiTypeN14
Dbk52BiTypeS	Dbk19BiTypeS
Dbk52BiTypeR	Dbk19BiTypeR
Dbk52BiTypeB	Dbk19BiTypeB
DBK52 Unipolar	
Dbk52UniCJC	Dbk19UniCJC
Dbk52UniTypeJ	Dbk19UniTypeJ
Dbk52UniTypeK	Dbk19UniTypeK
Dbk52UniTypeT	Dbk19UniTypeT
Dbk52UniTypeE	Dbk19UniTypeE
Dbk52UniTypeN28	Dbk19UniTypeN28
Dbk52UniTypeN14	Dbk19UniTypeN14
Dbk52UniTypeS	Dbk19UniTypeS
Dbk52UniTypeR	Dbk19UniTypeR
Dbk52UniTypeB	Dbk19UniTypeB

ADC Trigger Source Definitions

daqAdcTriggerSource		DaqEnhTrigSensT	
DatsImmediate	0	DetsRisingEdge	0
DatsSoftware	1	DetsFallingEdge	1
DatsAdcClock	2	DetsAboveLevel	2
DatsGatedAdcClock	3	DetsBelowLevel	3
DatsExternalTTL	4	DetsAfterRisingEdge	4
DatsHardwareAnalog	5	DetsAfterFallingEdge	5
DatsSoftwareAnalog	6	DetsAfterAboveLevel	6
DatsEnhancedTrig	7	DetsAfterBelowLevel	7

ADC Miscellaneous Definitions

ADC Flag Definitions - daqAdcFlag

Analog/High Speed Digital Flag		Unsigned/Signed ADC Data Flag		SSH Hold/Sample Flag - For Internal Use Only	
DafAnalog	00h	DafUnsigned	00h	DafSSHSample	00h
DafHighSpeedDigita	01h	DafSigned	04h	DafSSHHold	10h
Unipolar/Bipolar Flag		**Single Ended/Differential Flag**		**Clear or shift the least significant nibble - typically used with 12-bit ADCs**	
DafUnipolar	00h	DafSingleEnded	00h	DafIgnoreLSNibble	00h
DafBipolar	02h	DafDifferential	08h	DafClearLSNibble	20h
				DafShiftLSNibble	40h

Frequency vs Period - daqAdcRateMode	
DarmPeriod	0
DarmFrequency	1

ADC Acquisition Mode Definitions - daqAdcAcqMode	
DaamNShot	0
DaamNShotRearm	1
DaamInfinitePost	2
DaamPrePost	3

ADC Transfer Mask Definitions - daqAdcTransferMask	
DatmCycleOff	00h
DatmCycleOn	01h
DatmUpdateBlock	00h
DatmUpdateSingle	02h
DatmWait	00h
DatmReturn	04h
DatmUserBuf	00h
DatmDriverBuf	08h

ADC Clock Source Definitions -daqAdcClockSource	
DacsAdcClock	0
DacsGatedAdcClock	1
DacsTriggerSource	2

ADC File Open Mode Definitions - daqAdcOpenMode	
DaomAppendFile	0
DaomWriteFile	1
DaomCreateFile	2

ADC Acquisition/Transfer Active Flag Definitions - daqAdcActiveFlag	
DaafAcqActive	01h
DaafAcqTriggered	02h
DaafTransferActive	04h

ADC Acquisition State - daqAdcAcqState	
DaasPreTrig	0
DaasPostTrig	1

ADC Buffer Transfer Mask - daqAdcBufferXferMask	
DabtmOldest	1
DabtmNewest	2
DabtmWait	3
DabtmReturn	4

DAC Definitions

DAC Device Type Definitions - `daqDacDeviceType`	
DddtLocal	0
DddtDbk	1

DAC Output Mode Definitions - `daqDacOutputMode`	
DdomVoltage	0
DdomStaticWave	1
DdomDynamicWave	2

DAC Trigger Source Definitions - `daqDacTriggerSource`	
DdtsImmediate	0
DdtsSoftware	1

DAC Clock Source Definitions - `daqDacClockSource`	
DdcsDacClock	0
DdcsGatedDacClock	1
DdcsAdcClock	2
DdcsExternalTTL	3
Ddcs9513Ctr1	4

DAC Waveform Mode Definitions - `daqDacWaveformMode`	
DdwmNShot	0
DdwmNShotRearm	1
DdwmInfinite	2

DAC Predefined Waveform Type Definitions - `daqDacWaveType`	
DdwtSine	0
DdwtSquare	1
DdwtTriangle	2

DAC Transfer Mask Definitions - `daqDacTransferMask`	
DdtmCycleOff	00h
DdtmCycleOn	01h
DdtmUpdateBlock	00h
DdtmUpdateSingle	02h

DAC Waveform/Transfer Active Flag Definitions - `daqDacActiveFlag`	
DdafWaveformActive	01h
DdafWaveformTriggered	02h
DdafTransferActive	04h
DdafUnderrun	08h

Data Conversion Definitions

Software Calibration Type Definitions - `DcalType`		
DcalTypeDefault	0	
DcalTypeCJC	1	channel to be calibrated is a real CJC reading - not a CJC zero reading
DcalDbk4Bypass	2	channel to be calibrated using the methods and structures for a Dbk4 with the filters bypassed (set by jumper on the card)
DcalDbk4Filter	3	channel to be calibrated using the methods and structures for a Dbk4 with the cutoff filters enabled

RTD Type Definitions - `RtdType`		
Dbk9RtdType100	0	RTD 100 ohm Platinum alpha = .00385
Dbk9RtdType500	1	RTD 500 ohm Platinum alpha = .00385
Dbk9RtdType1K	2	RTD 1000 ohm Platinum alpha = .00385

Thermocouple Type Definitions - `TCType`						
DBK14		**DBK19**		**DBK52**		
Dbk14TCTypeJ	0	Dbk19TCTypeJ	9	Dbk52TCTypeJ	9	(Dbk19TCTypeJ)
Dbk14TCTypeK	1	Dbk19TCTypeK	10	Dbk52TCTypeK	10	(Dbk19TCTypeK)
Dbk14TCTypeT	2	Dbk19TCTypeT	11	Dbk52TCTypeT	11	(Dbk19TCTypeT)
Dbk14TCTypeE	3	Dbk19TCTypeE	12	Dbk52TCTypeE	12	(Dbk19TCTypeE)
Dbk14TCTypeN28	4	Dbk19TCTypeN28	13	Dbk52TCTypeN28	13	(Dbk19TCTypeN28)
Dbk14TCTypeN14	5	Dbk19TCTypeN14	14	Dbk52TCTypeN14	14	(Dbk19TCTypeN14)
Dbk14TCTypeS	6	Dbk19TCTypeS	15	Dbk52TCTypeS	15	(Dbk19TCTypeS)
Dbk14TCTypeR	7	Dbk19TCTypeR	16	Dbk52TCTypeR	16	(Dbk19TCTypeR)
Dbk14TCTypeB	8	Dbk19TCTypeB	17	Dbk52TCTypeB	17	(Dbk19TCTypeB)

WBK Card Definitions

WBK Option Values - DaqChanOptionValue	
WBK12 Filter-Type - `WcotWbk12FilterType`	
`DcovWbk12FilterElliptic`	0
`DcovWbk12FilterLinear`	1
WBK12 Filter-Mode - `WcotWbk12FilterMode`	
`DcovWbk12FilterBypass`	0
`DcovWbk12FilterOn`	1
WBK12 Anti-Aliasing Filter-Mode-`WcotWbk12PreFilterMode`	
`DcovWbk12PreFilterDefault`	0
`DcovWbk12PreFilterOff`	1
WBK13 Filter-Type - `WcotWbk13FilterType`	
`DcovWbk13FilterElliptic`	0
`DcovWbk13FilterLinear`	1
WBK13 Filter-Mode - `WcotWbk13FilterMode`	
`DcovWbk13FilterBypass`	0
`DcovWbk13FilterOn`	1
WBK13 Anti-Aliasing Filter-Mode- `WcotWbk13PreFilterMode`	
`DcovWbk13PreFilterDefault`	0
`DcovWbk13PreFilterOff`	1
WBK14 Current-Source - `WcotWbk14CurrentSrc`	
`DcovWbk14CurrentSrcOff`	0
`DcovWbk14CurrentSrc2mA`	1
`DcovWbk14CurrentSrc4mA`	2
WBK14 High-Pass Filter - `WcotWbk14HighPassCutOff`	
`DcovWbk14HighPass0_1Hz`	0
`DcovWbk14HighPass10Hz`	1
WBK14 Low-Pass Filter-Mode - `WcotWbk14LowPassMode`	
`DcovWbk14LowPassBypass`	0
`DcovWbk14LowPassOn`	1
WBK-14 Low-Pass Filter-Mode - `WcotWbk14PreFilterMode`	
`DcovWbk14PreFilterDefault`	0
`DcovWbk14PreFilterOff`	1
WBK14 Excitation-Source Waveform - `WmotWbk14ExcSrcWaveform`	
`DmovWbk14ExcSrcRandom`	0
`DmovWbk14ExcSrcSine`	1

WBK Channel Options - DaqAdcExpType	
`DoctWbk11`	6
`DoctWbk12`	7
`DoctWbk13`	8
`DmctWbk512`	9
`DmctWbk10`	10
`DmctWbk14`	11
`DmctWbk15`	12
`DmctResponseDac`	*13
*Response DAC on WaveBook	

WBK Module Option-Types - `DaqOptionType`	
`DcotWbk12FilterCutOff`	0
`DcotWbk12FilterType`	1
`DcotWbk12FilterMode`	2
`DcotWbk12PreFilterMode`	3
`DcotWbk13FilterCutOff`	0
`DcotWbk13FilterType`	1
`DcotWbk13FilterMode`	2
`DcotWbk13PreFilterMode`	3
`DcotWbk14LowPassMode`	0
`DcotWbk14LowPassCutOff`	1
`DcotWbk14HighPassCutOff`	2
`DcotWbk14CurrentSrc`	3
`DcotWbk14PreFilterMode`	4
`DmotWbk14ExcSrcWaveform`	5
`DmotWbk14ExcSrcFreq`	6
`DmotWbk14ExcSrcAmplitude`	7
`DmotWbk14ExcSrcOffset`	8

General I/O Definitions

I/O Device Type Definitions - `daqIODeviceType`	
`DiodtLocalBitIO`	0
`DiodtLocal8255`	1
`DiodtLocal9513`	2
`DiodtExp8255`	3 Dbk20, Dbk21
`DiodtDbk23`	4
`DiodtDbk24`	5
`DiodtDbk25`	6
`DiodtExp9513`	7 Not available

I/O Operation Code Definitions - `daqIOOperationCode`	
`DioocReadByte`	0
`DioocWriteByte`	1
`DioocReadWord`	2
`DioocWriteWord`	3
`DioocReadDWord`	4
`DioocWriteDWord`	5

I/O Operation Code Definitions - `daqIOExpansionPort`	
`DioepP1`	0
`DioepP2`	1
`DioepP3`	2

DAC Transfer Mask Definitions - `daqIOTransferMask`	
`DiotmCycleOff`	0
`DiotmCycleOn`	1

I/O Operation Code Definitions - `daqIOEventCode`	
`DioecP1IR`	0
`DioecP2IR`	1
`DioecP3IR`	2

DAC Transfer Active Flag Definitions - `daqIOActiveFlag`	
`DioafDone`	0
`DioafArmed`	1
`DioafTriggered`	2

I/O Port Type Definitions - `daqIODevicePort`

Local 9513, Expansion 9513	
`Diodp9513Command`	0
`Diodp9513Data`	1
`Diodp9513MasterMode`	2
`Diodp9513Alarm1`	3
`Diodp9513Alarm2`	4
`Diodp9513Status`	5
`Diodp9513Mode1`	6
`Diodp9513Mode2`	7
`Diodp9513Mode3`	8
`Diodp9513Mode4`	9
`Diodp9513Mode5`	10
`Diodp9513Load1`	11
`Diodp9513Load2`	12
`Diodp9513Load3`	13
`Diodp9513Load4`	14
`Diodp9513Load5`	15
`Diodp9513Hold1`	16
`Diodp9513Hold2`	17
`Diodp9513Hold3`	18
`Diodp9513Hold4`	19
`Diodp9513Hold5`	20
`Diodp9513Hold1HC`	21 *
`Diodp9513Hold2HC`	22 *
`Diodp9513Hold3HC`	23 *
`Diodp9513Hold4HC`	24 *
`Diodp9513Hold5HC`	25 *
* Hold register when in hold cycle mode	

Local Bit I/O	
`DiodpBitIO`	0

Local 8255, Dbk20, Dbk21	
`Diodp8255A`	0
`Diodp8255B`	1
`Diodp8255C`	2
`Diodp8255IR`	3
`Diodp8255CHigh`	4
`Diodp8255CLow`	5

DBK23	
`DiodpDbk23A`	0
`DiodpDbk23B`	1
`DiodpDbk23C`	2
`DiodpDbk23Unused`	3

DBK24	
`DiodpDbk24A`	0
`DiodpDbk24B`	1
`DiodpDbk24C`	2
`DiodpDbk24Unused`	3

DBK25	
`DiodpDbk25`	0

9513 Counter/Timer Definitions

Time-of-Day Definitions - `daq9513TimeOfDay`	
DtodDisabled	0
DtodDivideBy5	1
DtodDivideBy6	2
DtodDivideBy10	3

Count Source Definitions - `daq9513CountSource`	
DcsTcnM1*	0
DcsSrc1	1
DcsSrc2	2
DcsSrc3	3
DcsSrc4	4
DcsSrc5	5
DcsGate1	6
DcsGate2	7
DcsGate3	8
DcsGate4	9
DcsGate5**	10
DcsF1**	11
DcsF2**	12
DcsF3**	13
DcsF4**	14
DcsF5**	15

*invalid with `daq9513SetMasterMode` or `daqCtrRdFreq`
**invalid with `daq9513RdFreq`

Output Control Definitions - `daq9513OutputControl`	
DocInactiveLow	0
DocHighTermCntPulse	1
DocTCToggled	2
DocInactiveHighImp	3
DocLowTermCntPulse	4

Gating Control Definitions - `daq9513GatingControl`	
DgcNoGating	0
DgcHighTCNM1	1
DgcHighLevelGateNP1	2
DgcHighLevelGateNM1	3
DgcHighLevelGateN	4
DgcLowLevelGateN	5
DgcHighEdgeGateN	6
DgcLowEdgeGateN	7

Multiple Counter Command Definitions - `daq9513MultCtrCommand`	
DmccArm	0
DmccLoad	1
DmccLoadArm	2
DmccDisarmSave	3
DmccSave	4
DmccDisarm	5

`daqTest` Command Definitions

DaqTestCommand	
DtstBaseAddressValid	0
DtstInterruptLevelValid	1
DtstDmaChannelValid	2
DtstAdcFifoInputSpeed	3
DtstDacFifoOutputSpeed	4
DtstIOInputSpeed	5
DtstIOOutputSpeed	6

Calibration Input Signal Sources

DaqCalInputT		
DciNormal	0	External signal from device input connector(s)
DciCalGround	1	Internal calibration ground signal
DciCal5V	2	Internal 5 V calibration signal
DciCal500mV	3	Internal 500 mV calibration signal
DaqCalTableTypeT		
DcttFactory	0	Factory calibration constants
DcttUser	1	User-defined calibration constants

API Error Codes

Error Name	Code # hex - dec	Description
DerrNoError	00h - 0	No error
DerrBadChannel	01h - 1	Specified LPT channel was out-of-range
DerrNotOnLine	02h - 2	Requested device is not online
DerrNoDaqbook	03h - 3	DaqBook is not on the requested channel
DerrBadAddress	04h - 4	Bad function address
DerrFIFOFull	05h - 5	FIFO Full detected, possible data corruption
DerrBadDma	06h - 6	Bad or illegal DMA channel or mode specified for device
DerrBadInterrupt	07h - 7	Bad or illegal INTERRUPT level specified for device
DerrDmaBusy	08h - 8	DMA is currently being used
DerrInvChan	10h - 16	Invalid analog input channel
DerrInvCount	11h - 17	Invalid count parameter
DerrInvTrigSource	12h - 18	Invalid trigger source parameter
DerrInvLevel	13h - 19	Invalid trigger level parameter
DerrInvGain	14h - 20	Invalid channel gain parameter
DerrInvDacVal	15h - 21	Invalid DAC output parameter
DerrInvExpCard	16h - 22	Invalid expansion card parameter
DerrInvPort	17h - 23	Invalid port parameter
DerrInvChip	18h - 24	Invalid chip parameter
DerrInvDigVal	19h - 25	Invalid digital output parameter
DerrInvBitNum	1Ah - 26	Invalid bit number parameter
DerrInvClock	1Bh - 27	Invalid clock parameter
DerrInvTod	1Ch - 28	Invalid time-of-day parameter
DerrInvCtrNum	1Dh - 29	Invalid counter number
DerrInvCntSource	1Eh - 30	Invalid counter source parameter
DerrInvCtrCmd	1Fh - 31	Invalid counter command parameter
DerrInvGateCtrl	20h - 32	Invalid gate control parameter
DerrInvOutputCtrl	21h - 33	Invalid output control parameter
DerrInvInterval	22h - 34	Invalid interval parameter
DerrTypeConflict	23h - 35	An integer was passed to a function requiring a character
DerrMultBackXfer	24h - 36	A second background transfer was requested
DerrInvDiv	25h - 37	Invalid Fout divisor
Temperature Conversion Errors		
DerrTCE_TYPE	26h - 38	TC type out-of-range
DerrTCE_TRANGE	27h - 39	Temperature out-of-CJC-range
DerrTCE_VRANGE	28h - 40	Voltage out-of-TC-range
DerrTCE_PARAM	29h - 41	Unspecified parameter value error
DerrTCE_NOSETUP	2Ah - 42	dacTCConvert called before dacTCSetup
DaqBook		
DerrNotCapable	2Bh - 43	Device is incapable of function
Background		
DerrOverrun	2Ch - 44	A buffer overrun occurred
Zero and Cal Conversion Errors		
DerrZCInvParam	2Dh - 45	Unspecified parameter value error
DerrZCNoSetup	2Eh - 46	dac…Convert called before dac…Setup
DerrInvCalFile	2Fh - 47	Cannot open the specified cal file
Environmental Errors		
DerrMemLock	30h - 48	Cannot lock allocated memory from operating system
DerrMemHandle	31h - 49	Cannot get a memory handle from operating system
Pre-trigger acquisition Errors		
DerrNoPreTActive	32h - 50	No pre-trigger configured
Daq FIFO Errors (DaqBoard only)		
DerrInvDacChan	33h - 51	DAC channel does not exist
DerrInvDacParam	34h - 52	DAC parameter is invalid
DerrInvBuf	35h - 53	Buffer points to NULL or buffer size is zero
DerrMemAlloc	36h - 54	Could not allocate the needed memory
DerrUpdateRate	37h - 55	Could not achieve the specified update rate
DerrInvDacWave	38h - 56	Could not start waveforms because of missing or invalid parameters
DerrInvBackDac	39h - 57	Could not start waveforms with background transfers
DerrInvPredWave	3Ah - 58	Predefined waveform not supported
RTD Conversion Errors		
DerrRtdValue	3Bh - 59	rtdValue out-of-range
DerrRtdNoSetup	3Ch - 60	rtdConvert called before rtdSetup

Error Name	Code # hex - dec	Description
`DerrRtdArraySize`	`3Dh - 61`	Temperature array not large enough
`DerrRtdParam`	`3Eh - 62`	Incorrect RTD parameter
`DerrInvBankType`	`3Fh - 63`	Invalid bank-type specified
`DerrBankBoundary`	`40h - 64`	Simultaneous writes to DBK cards in different banks not allowed
`DerrInvFreq`	`41h - 65`	Invalid scan frequency specified
`DerrNoDaq`	`42h - 66`	No Daq112B/216B installed
`DerrInvOptionType`	`43h - 67`	Invalid option-type parameter
`DerrInvOptionValue`	`44h - 68`	Invalid option-value parameter
New *API Error Codes*		
`DerrTooManyHandles`	`60h - 96`	No more handles available to open
`DerrInvLockMask`	`61h - 97`	Only a part of the resource is already locked, must be all or none
`DerrAlreadyLocked`	`62h - 98`	All or part of the resource was locked by another application
`DerrAcqArmed`	`63h - 99`	Operation not available while an acquisition is armed
`DerrParamConflict`	`64h - 100`	Each parameter is valid, but the combination is invalid
`DerrInvMode`	`65h - 101`	Invalid acquisition/wait/dac mode
`DerrInvOpenMode`	`66h - 102`	Invalid file-open mode
`DerrFileOpenError`	`67h - 103`	Unable to open file
`DerrFileWriteError`	`68h - 104`	Unable to write file
`DerrFileReadError`	`69h - 105`	Unable to read file
`DerrInvClockSource`	`6Ah - 106`	Invalid acquisition mode
`DerrInvEvent`	`6Bh - 107`	Invalid transfer event
`DerrTimeout`	`6Ch - 108`	Time-out on wait
`DerrInitFailure`	`6Dh - 109`	Unexpected result occurred while initializing the hardware
`DerrBufTooSmall`	`6Eh - 110`	Unexpected result occurred while initializing the hardware
`DerrInvType`	`6Fh - 111`	Invalid acquisition/wait/dac mode
`DerrArraySize`	`70h - 112`	Used as a catch all for arrays not large enough
`DerrBadAlias`	`71h - 113`	Invalid alias names for Vxd lookup
`DerrInvCommand`	`72h - 114`	Invalid command
`DerrInvHandle`	`73h - 115`	Invalid handle
`DerrNoTransferActive`	`74h - 116`	Transfer not active
`DerrNoAcqActive`	`75h - 117`	Acquisition not active
`DerrInvOpstr`	`76h - 118`	Invalid operation string used for enhanced triggering
`DerrDspCommFailure`	`77h - 119`	Device with DSP failed communication
`DerrEepromCommFailure`	`78h - 120`	Device with EEPROM failed communication
`DerrInvEnhTrig`	`79h - 121`	Device using enhanced trigger detected invalid trigger type
`DerrInvCalConstant`	`7Ah - 122`	User calibration constant out of range
`DerrInvErrorCode`	`7Bh - 123`	Invalid error code
`DerrInvAdcRange`	`7Ch - 124`	Invalid analog input voltage range parameter
`DerrInvCalTableType`	`7Dh - 125`	Invalid calibration table type
`DerrInvCalInput`	`7Eh - 126`	Invalid calibration input signal selection
`DerrInvRawDataFormat`	`7Fh - 127`	Invalid raw-data format selection
`DerrNotImplemented`	`80h - 128`	Feature/function not implemented yet
`DerrInvDioDeviceType`	`81h - 129`	Invalid digital I/O device type
`DerrInvPostDataFormat`	`82h - 130`	Invalid post-processing data format selection

Standard API Programming Models 4

Overview

By using the Application Programming Interface (API) with Daq* systems, you can create custom software to satisfy your data acquisition requirements. Chapter 5 (*Daq* Command Reference Standard API*) explains the related API functions in detail. This chapter shows how to combine API functions to perform typical tasks using the standard API (enhanced models are in chapter 3). When you understand how the API commands work together and with the hardware, you are ready to program for optimum data acquisition. To help you get this programmer's perspective, this chapter is divided into 3 parts:

- **Data Acquisition Environment** outlines related concepts and defines Daq* capabilities the programmer must work with (the API, hardware features, and signal management).
- **Programming Models** explains the sequence and type of operations necessary for data acquisition. These models provide the software building blocks to develop more complex and specialized programs. The description for each model has a flowchart and program excerpt to show how the API functions work.
- **Summary Guide of Selected API Functions** is an easy-to-read table that describes when to use the basic API functions.

Data Acquisition Environment

In order to write effective data acquisition software, programmers must understand:

- Software tools (the API documented in this manual and the programming language—you may need to consult documentation for your language).
- Hardware capabilities and constraints.
- General concepts of data acquisition and signal management.

Application Programming Interface (API)

The API includes all the software functions needed for building a data acquisition system with the hardware described in the user's manual. Chapter 5 supplies the details how each function is used (parameters, hardware applicability, etc). (The *Visual Basic VBX Support* chapter explains special features available in a VBX environment.) In addition, you may need to consult your language and computer documentation.

Standard vs Enhanced API

Major differences between the standard and enhanced APIs were described in the introductory chapter. Language support varies as follows:

- standard API accommodates C, QuickBASIC, Visual Basic, and Turbo Pascal 7
- enhanced API accommodates C, Visual Basic, and Delphi.

Note: Codes for standard and enhanced APIs are NOT compatible; hence, a separate chapter of programming models for each (this chapter is for the standard API models; chapter 2 is for the enhanced API models).

Hardware Capabilities and Constraints

To program the system effectively, you must understand your Daq* and DBK hardware capabilities. Obviously you cannot program the hardware to perform beyond its design and specifications, but you also want to take full advantage of the system's power and features. In the hardware *User s Manual*, you may need to refer to sections that describe your hardware's capability. In addition, you may need to consult your computer documentation. In some cases, you may need to verify the hardware setup, use of channels, and signal conditioning options. Some hardware devices have jumpers and DIP switches that must match the programming (or reprogramming as the system evolves).

Signal Environment

This guide refers to several data acquisition concepts. Such concepts important for programmers are listed here and explained in the chapter *Signal Management and Troubleshooting Tips* in the *User s Manual*. You must apply these concepts as needed in your situation. Some of these concepts include:

- **Channel Identification**. Refer to *Signal Management* and the related reference table in chapter 5.
- **Scan Rates and Sequencing**. With multiple scans, the time between scans becomes a parameter. This time can be a constant or can be dependent upon a trigger. Refer to *Signal Management*.
- **Counter/Timer Operation**. Refer to *Signal Management* and DaqCtr* functions in chapter 5.
- **Triggering Options**. Triggering starts the A/D conversion. The trigger can be an external analog or TTL trigger, or a program controlled software trigger. Refer to *Signal Management* and the trigger functions in chapter 5.
- **Foreground/Background**. Foreground transfer routines require the entire transfer to occur before returning control to the application program. Background routines start the A/D acquisition and return control to the application program before the transfer occurs. Data is transferred while the application program is running. Data will be transferred to the user memory buffer during program execution in 1 sample or 256 sample blocks depending on the configuration. The programmer must determine what tasks can proceed in the background while other tasks perform in the foreground and how often the status of the background operations should be checked.
- **Tagged Data**. 12-bit Daq*s return data in a 16-bit format: the upper 12 bits contain the A/D readings, and the lower 4 bits contain channel information. Channel tagging can be enabled/disabled using the daqAdcSetTag command. Tagged data can be converted to an array of A/D readings and an array of channel numbers using daqAdcConvertTagged. The DaqBook and DaqBoard can use channel tagging, but the Daq PCMCIA cannot. Refer to daqAdcSetTag in chapter 5.

Parameters in the various A/D routines include: number of channels; number of scans; start of conversion triggering; timing between scans; and mode of data transfer. Up to 512 A/D channels can be sampled in a single scan. These channels can be consecutive or non-consecutive with the same or different gains. The scan sequence makes no distinction between local and expansion channels.

Basic Models

This section outlines basic programming steps commonly used for data acquisition. Consider the models as building blocks that can be put together in different ways or modified as needed. As a general tutorial, these examples use QuickBASIC since most programmers know BASIC and can translate to other languages as needed.

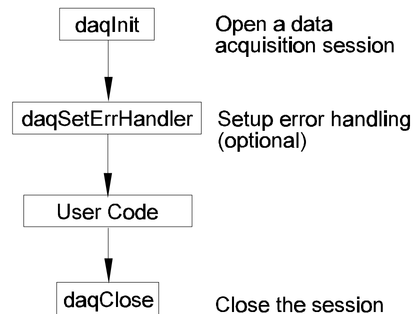
The standard API programming models discussed in this chapter include:

Model Type	Model Name	Page
Configuration	Initialization and Error Handling	4-4
Acquisition	Foreground Acquisition with User-Level Commands Foreground Acquisition with Low-Level Commands Foreground Acquisition, High-Speed Digital Input Background Acquisition, Multi-Channel, Multi-Scan Background Acquisition, Direct-To-Disk In Cycle Mode	4-5 4-7 4-8 4-9 4-11
Analog Output	Analog Output Generating DAC FIFO Waveforms with User-Level Commands (DaqBoard Only) Generating DAC FIFO Waveforms with Hardware-Level Commands (DaqBoard Only)	4-13 4-14 4-16
Use of P3 s Counter/Timer	Background Counter Acquisition Using Interrupts Variable Rate, Variable Duty-Cycle Square-Wave Output	4-18 4-20
Use of 8255 Chip	Single Square-Wave Output Digital I/O on P2	4-22 4-23
Temperature Measurements	Temperature Measurements Using Single TC Type on Single DBK19 Card Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards Temperature Measurements Using Multiple RTDs on a Single DBK9 Card	4-24 4-32 4-35
Calibration	Using DBK Card Calibration Files	4-37
Zero Compensation	Zero Compensation	4-40
Conversion	Linear Conversions	4-42

Initialization and Error Handling

This program (INITEX1.BAS) demonstrates how to initialize the Daq* and use various methods of error handling. Similar code exists in all the example programs but are only detailed here. Functions used include:

- QBdaqInit%(lptPort%, intr%)
- QBdaqSetErrHandler%(errHandler%)
- QBdaqClose%



Every program begins with an INCLUDE directive which defines constants and declarations used in the program. (daqbook.bi for QuickBASIC; DaqBook.bas for Visual Basic; ifcode.int for Turbo Pascal; DaqBook.h for C and DLL).

```
'$INCLUDE: 'daqbook.bi'
CLS
PRINT "INIT1.BAS": PRINT
Ret% = QBdaqInit%(LPT1%, 7)
Ret% = QBdaqClose%
```

If there was a problem initializing, BASIC would return an "Illegal Function Call" error. Disable the Daq* from reporting errors to BASIC.

```
Ret% = QBdaqSetErrHandler%(0)
```

If there is a Daq* error, the program will continue. The function's return value (an error number or 0 if no error) can help you debug a program.

```
IF (QBdaqInit%(LPT1%,)  0) THEN
 PRINT "Cannot initialize DaqBook!"
```

Daq* functions return daqErrno%.

```
 PRINT "daqErrno% : "; HEX$(daqErrno%)
END IF
```

The next statement defines an error handling routine that frees us from checking the return value of every Daq* function call. Although not necessary, this sample program transfers program control to a user-defined routine when an error is detected. Without a Daq* error handler, QuickBASIC will receive and handle the error, post it on the screen and terminate the program. QuickBASIC provides an integer variable (ERR) that contains the most recent error code. This variable can be used to detect the error source and take the appropriate action. The function QBdaqSetErrHandler tells QuickBASIC to assign ERR to a specific value when a Daq*error is encountered. The following line tells QuickBASIC to set ERR to 100 when a Daq*error is encountered. (Other languages work similarly; refer to specific language documentation as needed.)

```
Ret% = QBdaqSetErrHandler%(100)
```

The ON ERROR GOTO command in QuickBASIC allows a user-defined error handler to be provided, rather than the standard error handler that QuickBASIC uses automatically. The program uses ON ERROR GOTO to transfer program control to the label ErrorHandler if an error is encountered.

```
ON ERROR GOTO ErrorHandler
```

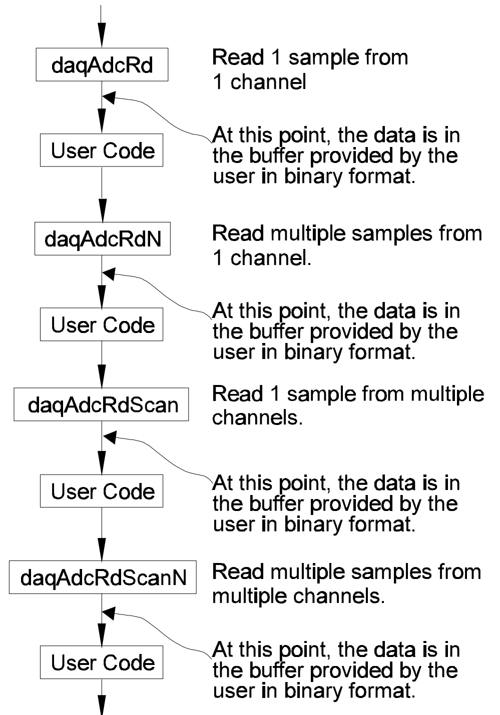
Daq* errors will send the program into the error handling routine. The body of the program goes here.

```
Ret% = QBdaqInit%(LPT1%, 7)
END
```

This is the error handler. Program control is sent here on error.

```
ErrorHandler:
PRINT "ERROR! Program aborted"
PRINT "BASIC Error :"; ERR
IF ERR = 100 THEN PRINT "DaqBook Error : "; HEX$(daqErrno%)
END
```

Foreground Acquisition with User-Level Commands



The program ADCEX1.BAS shows the use of several high-level analog input routines. These commands are easier to use than low-level commands but less flexible in scan configuration. This example demonstrates the use of the Daq*'s 4 highest ADC functions and channel tagging. Functions used include:

- QBdaqAdcRd%(chan%, sample%, gain%)
- QBdaqAdcRdN%(chan%, Buf%(), count%, trigger%, level%, freq!, gain%)
- QBdaqAdcRdScan%(startChan%, endChan%, Buf%(), gain%)
- QBdaqAdcRdScanN%(startChan%, endChan%, Buf%(), count%, trigger%, level%, freq!, gain%)
- QBdaqAdcSetTag%(Tag%)
- QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), count%)

This program will initialize the Daq* hardware, then take readings from the analog input channels in the base unit (not the expansion cards). For transporting data in and out of the Daq* driver, arrays are dimensioned.

```
DIM sample%(1), buf%(80),
 taggedData%(80), tags%(80), ret%
```

Although not required, this example disables channel tagging. When analog input channels in the base unit are accessed, the upper 4 bits of the 16-bit value are a channel tag; unless disabled by the following function.

```
ret% = QBdaqAdcSetTag%(0)
```

The next line requests 1 reading from 1 channel with a gain of ×1. The variable DgainX1% is actually a defined constant from DAQBOOK.BI, included at the beginning of this program.

```
ret% = QBdaqAdcRd%(0, sample%(0), DgainX1%)
PRINT USING "&   ####"; "Result of AdcRd:"; sample%(0): PRINT
```

The next line requests 10 readings from channel 0 at a gain of ×1, using the pacer clock at 1 kHz.

```
ret% = QBdaqAdcRdN%(0, buf%(), 10, DtsPacerClock%, 0, 1000!, DgainX1%)
PRINT "Results of AdcRdN: ";
FOR x = 0 TO 9
  PRINT USING "####  "; buf%(x);
NEXT x
```

With channel tags enabled, the program will then collect one sample of channels 0 through 7 using the QBdaqAdcRdScan function.

```
ret% = QBdaqAdcSetTag%(1)
ret% = QBdaqAdcRdScan%(0, 7, taggedData%(), DgainX1%)
```

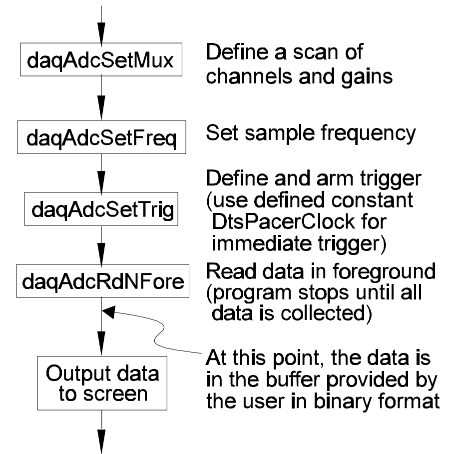
After the data has been collected and placed in a QuickBASIC array, the QBdaqAdcConvertTagged function can be used to separate the channel data from the tag data. After the function call, the data is in the buf% array and the tags are in the tags% array.

```
ret% = QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), 8)
PRINT "Results of AdcRdscan:"
FOR x = 0 TO 7
 PRINT USING "& #  &  ####"; "Channel:"; tags%(x); "Data:"; buf%(x)
NEXT x: PRINT
```

Using the QBdaqAdcRdScan function, the program will now take 10 readings from channels 0 though 7. After the data has been collected, the data is then separated from the tags.

```
ret% = QBdaqAdcRdScanN(0, 7, taggedData%(), 10, DtsPacerClock%, 0, 1000!,
 DgainX1%)
ret% = QBdaqAdcConvertTagged%(taggedData%(), buf%(),  tags%(), 80)
PRINT "Results of AdcRdscanN:"
FOR x = 0 TO 7
 PRINT USING "& #  &  "; "Channel:"; tags%(x); "Data:";
 FOR y = 0 TO 9
   PRINT USING "####  "; buf%((y * 8) + x);
 NEXT y: PRINT
NEXT x
```

Foreground Acquisition with Low-Level Commands



This program (ADCEX2.BAS) sets up an acquisition that collects scans in the foreground. After the channels and frequency have been configured, a foreground acquisition function is called. At this point, program execution is suspended until all the data is gathered. This example implements daqAdcRdNFore to get 10 samples from channels 0 through 7, triggered by the pacer clock with a 1000 Hz sampling frequency and unity gain. Functions used include:

- QBdaqAdcSetMux%(startChan%, endChan%, gain%)
- QBdaqAdcSetFreq%(freq!)
- QBdaqAdcSetTrig%(source%, level%, ctr0Mode%, pacerMode%)
- QBdaqAdcRdNFore%(Buf%(), count%)
- QBdaqAdcSetTag%(Tag%)
- QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), count%)

This program will initialize the Daq* hardware, then take readings from the analog input channels in the base unit (not the expansion cards). The functions used in this program are of a lower level than those used in ADCEX1.BAS and provide more flexibility.

```
DIM buf%(80), taggedData%(80), tags%(80), ret%
```

To begin the setup, the base unit multiplexer is set to scan through channels 0 through 7 with a gain of ×1.

```
ret% = QBdaqAdcSetMux%(0, 7, DgainX1%)
```

Next, set the internal sample rate to 1 kHz.

```
ret% = QBdaqAdcSetFreq%(1000!)
```

The acquisition begins on a trigger. The next line defines the trigger event to be the pacer clock, which will start the acquisition immediately. The variable DtsPacerClock% is a constant defined in DAQBOOK.BI. The one-shot parameter is set to continuous. Since the trigger source is not an analog input channel, the Level argument is not relevant. The ctr0Mode is also not relevant since the trigger is not DtsTTLRise or DtsTTLFall.

```
ret% = QBdaqAdcSetTrig%(DtsPacerClock%, 0, 0, 0, 1)
```

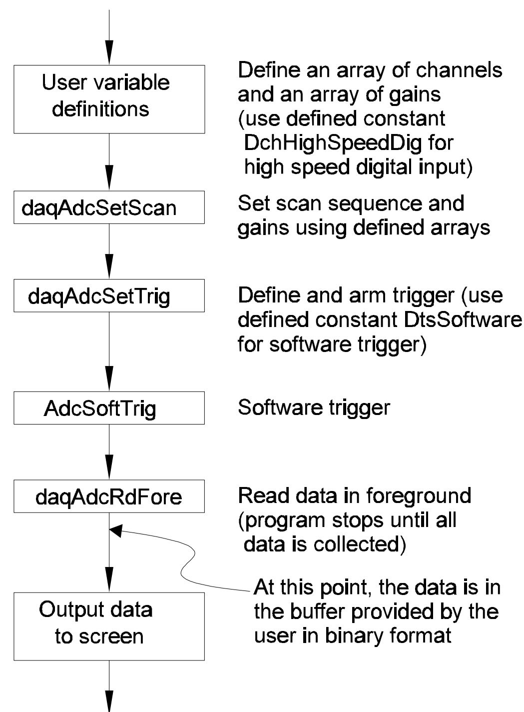
After setting up and arming the acquisition, the data is immediately ready to be collected. Had the trigger source been an external TTL signal or analog input, the data would only be ready after the trigger had been satisfied.

```
ret% = QBdaqAdcRdNFore%(taggedData%(), 10)
```

After the data has been collected and placed in a QuickBASIC array, the QBdaqAdcConvertTagged function can be used to separate the channel data from the tag data. After the function call, the data is in the buf% array and the tags are in the tags% array.

```
ret% = QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), 80)
PRINT "Results of AdcRdNFore:": PRINT
FOR x = 0 TO 7
  PRINT USING "& #  &  "; "Channel:"; tags%(x); "Data:";
    FOR y = 0 TO 9
            PRINT USING "####  "; buf%((y * 8) + x);
    NEXT y
  PRINT
NEXT x
ret% = QBdaqClose%
END
```

Foreground Acquisition, High-Speed Digital Input



This program (ADCEX3.BAS) reads a single value from the high-speed digital input (from the DaqBook/100/200 or DaqBoard/100A/200A) using a software trigger and foreground acquisition. After the channel and trigger are configured, the software trigger is executed and foreground acquisition starts. At this point, program execution is suspended until all the data is gathered. Functions used include:

- QBdaqAdcSetScan%(chans%(), gains%(), count)
- QBdaqAdcSetTrig%(source%, level%, ctr0Mode%, pacerMode%)
- QBdaqAdcSoftTrig%
- QBdaqAdcRdFore%(sample%)

```
DIM chans%(1), gains%(1), buf%(10),
  ret%
```

The QBdaqAdcSetScan function loads the scan sequencer with a list of channels and associated gains. The function call requires 2 arrays: an array of channels and an array of associated gains. This example will access only the high-speed digital input port, so only one entry is required in the array. The last parameter in the function call is the number of elements in the array, which in our case is 1. The variables DchHighSpeedDig% and DgainX1% are both constants found in DAQBOOK.BI.

```
chans%(0) = DchHighSpeedDig%
gains%(0) = DgainX1%
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 1)
```

The pacer clock is set using the QBdaqAdcSetClk function. Its 2 arguments adjust 2 internal counters which affect the scanning speed of the analog inputs. Assuming the internal clock jumper is in the default position of 1 MHz, the scan rate will be equal to 1 MHz/(argument1*argument2). In this case, 1,000,000/(10*10) leads to a scan rate of 10 kHz.

```
ret% = QBdaqAdcSetClk%(10, 10)
```

After setting up the sequencer, the trigger must be configured. This example uses a software trigger to start the acquisition. The function QBdaqAdcSoftTrig serves as the trigger. The variable DtsSoftware% is a defined constant in DAQBOOK.BI. The one-shot parameter is set to continuous. The Level parameter is irrelevant when using the software trigger. The ctr0mode argument is also not relevant since the trigger is not DtsTTLRise or DtsTTLFall.

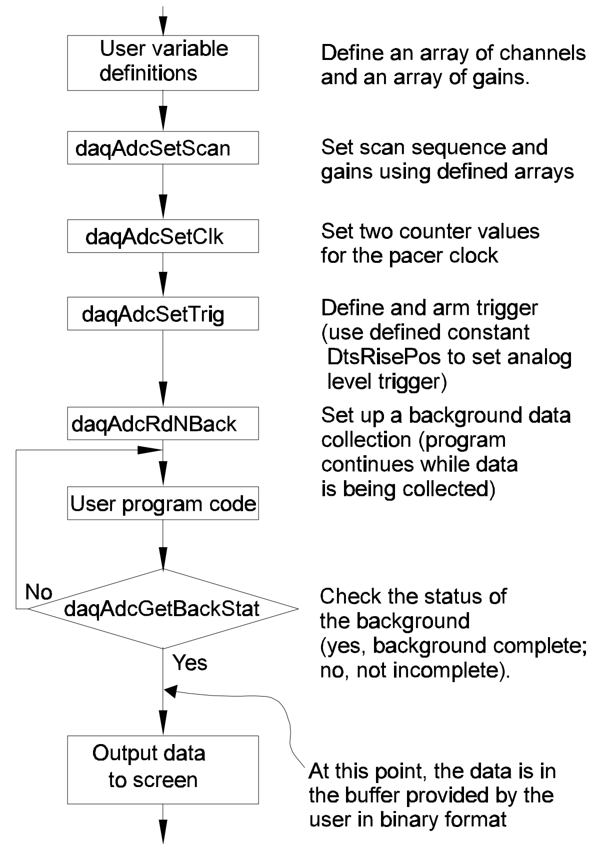
```
ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 1)
ret% = QBdaqAdcSoftTrig%
```

After the software trigger, the data can be collected.

```
ret% = QBdaqAdcRdNFore%(buf%(),10)
FOR x =0 TO 9
  PRINT USING "##/\ \" ; x + 1 ; "&H" ; HEX$(buf%(x))
NEXT
```

Background Acquisition, Multi-Channel, Multi-Scan

This program (ADCEX4.BAS) sets up an acquisition that collects scans in the background. After the acquisition is configured and armed, the program continues in the foreground while data is being collected in the background. The foreground program can poll the background acquisition to determine its status.



This example reads multiple channels and scans in the background mode to get 10 samples from channels 0 through 7 (triggered by an analog level) with a 1 Hz sampling frequency and unity gain. After the acquisition has been started, data is transferred to the user buffer as it is being collected (the user program continues to run in the foreground). Functions used include:

- QBdaqAdcSetScan%(chans%(), gains%(), count)
- QBdaqAdcSetClk%(ctr1%, ctr2%)
- QBdaqAdcSetTrig%(source%, level%, ctr0Mode%, pacerMode%)
- QBdaqAdcRdNBack%(Buf%(), count%, cycle%, armNotEmpty%)
- QBdaqAdcGetBackStat%(active%, count%)
- QBdaqAdcSetTag%(Tag%)
- QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), count%)

```
DIM taggedData%(80), buf%(80), tags%(80), active%, count%
```

The QBdaqAdcSetScan function loads the scan sequencer with a list of channels and associated gains. The function call requires two arrays; an array of channels, and an array of associated gains. This example will load the sequencer with channels 0 through 7, all with a gain of ×1. The last parameter in the function call is the number of elements in the array, which in our case is 8. The variable DgainX1% is a constant found in DAQBOOK.BI.

```
FOR x% = 0 TO 7
  chans%(x%) = x%
  gains%(x%) = DgainX1%
NEXT x%
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 8)
```

Although not required, this example enables channel tagging. When analog input channels in the base unit are accessed, the upper 4 bits of the 16-bit value are a channel tag. The following function call enables the channel tags.

```
ret% = QBdaqAdcSetTag%(1)
```

The pacer clock is set using the QBdaqAdcSetClk function. Its 2 arguments adjust 2 internal counters which affect the scanning speed of the analog inputs. Assuming the internal clock jumper is in the default position of 1 MHz, the scan rate will be equal to 1 MHz/(argument1*argument2). In this case, 1,000,000/(1,000*1,000) = 1 Hz.

```
ret% = QBdaqAdcSetClk%(1000, 1000)
```

The trigger is then set up using the QBdaqAdcSetTrig function. An analog level trigger is used to trigger the acquisition. The trigger channel is always the first one in the scan sequence, in this case channel 0. A level of 10 counts is selected.

```
ret% = QBdaqAdcSetTrig%(DtsAnalogRisePos%, 0, 10, 0, 1)
```

After setting up and arming the trigger, the function QBdaqAdcRdNBack can be called to collect the data in the background. This function will wait until the trigger is satisfied before attempting to collect the data. The following line collects 10 scans, placing them in the array taggedData%. This example shows the Cycle flag as OFF which will stop background operation after 10 scans have been collected. An Update Size parameter of 1 indicates the user buffer will be updated after every sample. A parameter of 0 indicates the user buffer will be updated after 256 samples.

```
ret% = QBdaqAdcRdNBack%(taggedData%(), 10, 0, 1)
```

At this point, the Daq* is armed and, depending on the state of the trigger, possibly collecting data. The program, however, proceeds to the next line. In our case, we enter into a polling loop to check the status of the background operation.

```
DO
 ret% = QBdaqAdcGetBackStat%(active%, count%)
 LOCATE 3, 1
 PRINT "Transfer in progress.  "; count%; " samples acquired"
LOOP WHILE active% <>0
PRINT "Acquisition complete : "; count%; " samples acquired": PRINT
```

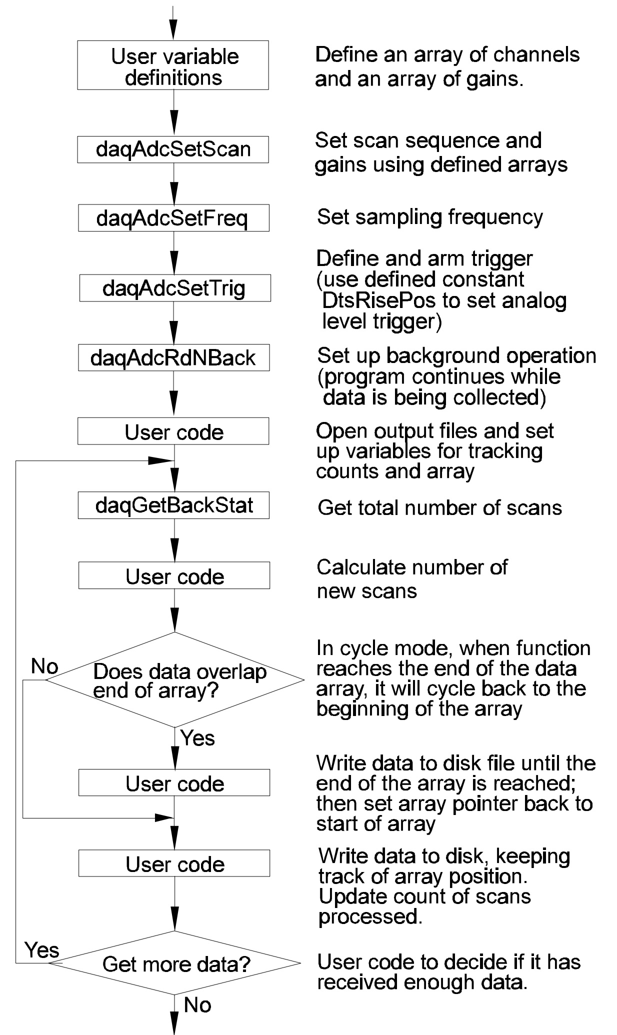
After the data has been collected and placed in a QuickBASIC array, the QBdaqAdcConvertTagged function can be used to separate the channel data from the tag data. After the function call, the data is in the buf% array and the tags are in the tags% array.

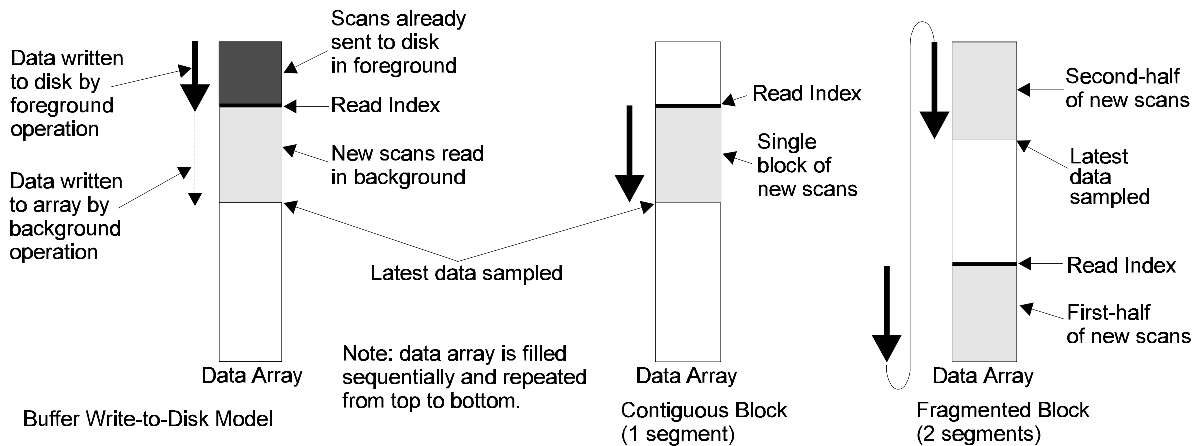
```
ret% = QBdaqAdcConvertTagged(taggedData%(), buf%(), tags%(), 80)
PRINT "Data acquired : ": PRINT
FOR x = 0 TO 7
     PRINT USING "& #  &  "; "Channel:"; tags%(x); "Data:";
     FOR y = 0 TO 9
           PRINT USING "####  "; buf%((y * 8) + x);
     NEXT y
     PRINT
NEXT x
ret% = QBdaqClose%
END
```

Background Acquisition, Direct-To-Disk In Cycle Mode

This program continuously reads data in the background and periodically writes data to disk in the foreground. In cycle mode, this data transfer can contiune indefinitely. When the background transfer reaches the end of the data array, it will reset its array pointer back to the beginning of the array and continue writing data to it. Thus, the allocated buffer can be used repeatedly like a FIFO buffer.

While reading and writing data, the program must track two variables. The first is the number of scans already processed and written to disk (ScansProcessed) versus the number of scans actually read by the Daq* hardware. The difference between this number and the count returned by daqADCGetBackStat is the number of new scans to be processed. The second item tracked is the array position. The program must write data to the disk until it reaches the end of the data array and then set the read index back to the beginning. As the background operation is filling dataArray, the foreground operation CollectDataTimer will empty dataArray to disk. The foreground emptying of dataArray will always lag background filling, but both will loop back to the beginning of dataQrray when the end is reched. Either the entire block of data can be written to disk, or the data needs to be broken up into two smaller blocks to be written to disk (see figure).

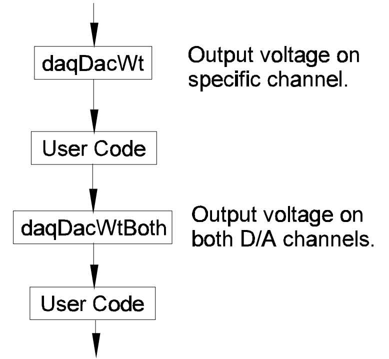




Use of a Data Array in Cycle Mode

```
DIM buffer1%(8000), buf%(8000), active%, count&, scansprocessed&
DIM newscans&, arrayposition&, j&
bufsize% = 1000

' Define arrays of channels and gains : 0-7 , unity gain
FOR x% = 0 TO 7
  chans%(x%) = x%
  gains%(x%) = DgainX1%
NEXT x%

' Load scan sequence FIFO
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 8)

' Set Sampling Frequency
ret% = QBdaqAdcSetFreq%(3000)

' Define and arm trigger
ret% = QBdaqAdcSetTrig%(DtsPacerClock%, 0, 0, 0, 0)

' Read data in the background
ret% = QBdaqAdcRdNBack%(buffer1%(), bufsize%, 1, 0)

' Write data to disk
OPEN "c:dasqdata.bin" FOR OUTPUT AS #1
scansprocessed& = 0
arrayposition& = 0

DO
  ret% = QBdaqAdcGetBackStat%(active%, count&)
  LOCATE 3, 1
  newscans& = count& - scansprocessed&
  PRINT "Number of scans acquired:"; count&
  PRINT "Number of scans saved to disk:"; scansprocessed&

  'Write scans to end of array.
  IF ((newscans& * 8) + arrayposition&) > (bufsize% * 8) THEN
   newscans& = newscans& - ((bufsize% * 8 - arrayposition&) / 8)
   FOR arrayposition& = arrayposition& TO (bufsize% * 8)
          PRINT #1, buffer1%(arrayposition&)
   NEXT arrayposition&
   arrayposition& = 0
  END IF

  'Write scans to disk
  numloops& = newscans& * 8
  FOR j& = 0 TO numloops&
   PRINT #1, buffer1%(arrayposition&)
   arrayposition& = arrayposition& + 1
  NEXT j&
  scansprocessed& = count&

LOOP WHILE scansprocessed& < 5000

PRINT "Acquisition complete.": PRINT
CLOSE #1
```

Analog Output



The program DACEX1.BAS shows how to output analog voltages on analog output channels 0 and 1. These commands only have to be issued one time unless explicity changed. The output voltages will be sustained. This example demonstrates the use of the two digital-to-analog converters (values used assume bipolar mode). Functions used include:

- QBdaqDacWt%(chan%, dataVal%)
- QBdaqDacWtBoth%(chan1Val%, chan2Val%)

Assuming the voltage reference to be connected to the internal, default of 5 V, the next function will set channel 0 to an output voltage of 5 V. Since the internal digital-to-analog converter has 12-bit resolution, 4095 represents full-scale. Channel 1 is equal to 0.

```
ret% = QBdaqDacWt%(0, 4095)
```

Prompt the user to hit a key to continue.

```
PRINT "5 VDC on channel 1."
PRINT "hit any key to continue..."
WHILE INKEY$ = "": WEND: PRINT
```

The QBdaqDacWtBoth writes to both analog outputs simultaneously. The next line sets channel 0 to 5 V and channel 1 to 2.5 V. At full-scale, a digital value of 4095 corresponds to 5 V; a digital value of 2048 corresponds to ½ of 5 V.

```
ret% = QBdaqDacWtBoth%(4095, 2048)
```

Prompt the user to hit a key to continue.

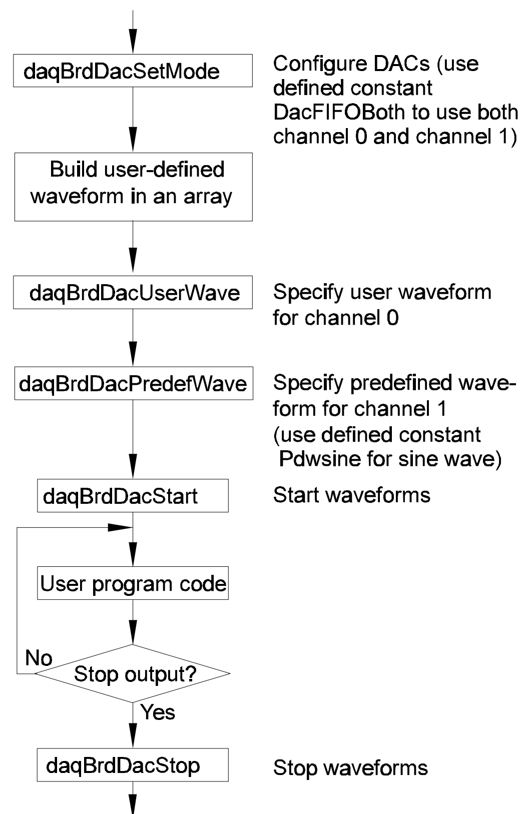
```
PRINT "5 VDC on channel 1, 2.5 VDC on channel 2."
PRINT "hit any key to continue..."
WHILE INKEY$ = "": WEND: PRINT
```

The next line sets both outputs to 0 V.

```
ret% = QBdaqDacWtBoth%(0,0)
```

Generating DAC FIFO Waveforms with User-Level Commands (DaqBoard Only)

This program (DACEX2.BAS) demonstrates the use of the DAC FIFO to generate waveforms with user-level commands. The DAC is configured for output on both channels, and the user waveform is constructed. Output begins after the waveform is assigned to a channel. At this point, the program continues while the waveforms are generated.



The user-level command set does not require an in-depth knowledge of the FIFO hardware. (DACEX3.BAS demonstrates the use of the low-level commands to directly manipulate the FIFO hardware.) Functions used include:

- QBdaqBrdDacSetMode%(updateRate!, mode%, cycle%)
- QBdaqBrdDacPredefWave%(DAC%, samples%, waveType%, amplitude%, offset%, dutyCycle%, phaseShift%)
- QBdaqBrdDacUserWave%(DAC%, buf%(), samples%)
- QBdaqBrdDacStart%
- QBdaqBrdDacStop%

Commands from the hardware-level and user-level command sets should not be used together in the same program. **Note:** This example uses the high-speed DAC FIFO to generate waveforms and can only be used with the DaqBoard product line.

```
DIM waveBuf%(512)
```

The next command sets the update rate (first parameter) to 10 µs per sample; enables waveform output on both DACs (second parameter); and sets the waveforms to cycle continuously.

```
ret% = QBdaqBrdDacSetMode%(10, DacFIFOBoth%, 1)
```

The next series of statements build the waveform that will be output on DAC channel 0. The waveform will be a ramp from the low-voltage level up to the high-voltage level for the first 128 samples (half of the waveform). It will then drop back down to the low-voltage level for the next quarter of the waveform and then rise to a level midway between the low and high references for the last quarter of the waveform. **Note**: The voltage references depend on the configuration of the hardware (refer to hardware sections of the manual as needed).

```
point% = 0
FOR x = 0 to 127
        waveBuf%(x) = point%
        point% = point% + &H20
NEXT x
FOR x = 128 to 191
        waveBuf%(x) = 0
NEXT x
FOR x = 192 to 255
        waveBuf%(x) = &H800
NEXT x
```

The next line assigns the 256 sample waveform we have just built to DAC channel 0.

```
ret% =  QBdaqBrdDacUserWave%(0, waveBuf%(), 256)
```

The next line assigns a 256-sample sine wave to DAC channel 1. The sine wave will have a peak-to-peak amplitude equal to the full-scale output of the DAC and will be centered around the half-scale point. The waveform will have a 50% duty-cycle and a phase shift that lags DAC channel 0 by 90 degrees. This waveform is built for you by the driver.

```
ret% = QBdaqBrdDacPredefWave%(1, 256, PdwSine%, &Hfff, &H800, 50 , 90)
```

The next line starts all the waveforms that have been set up by the QBdaqBrdDacSetMode, QBdaqBrdDacPredefWave, and QBdaqBrdDacUserWave commands.

```
ret% = QBdaqBrdDacStart%
```

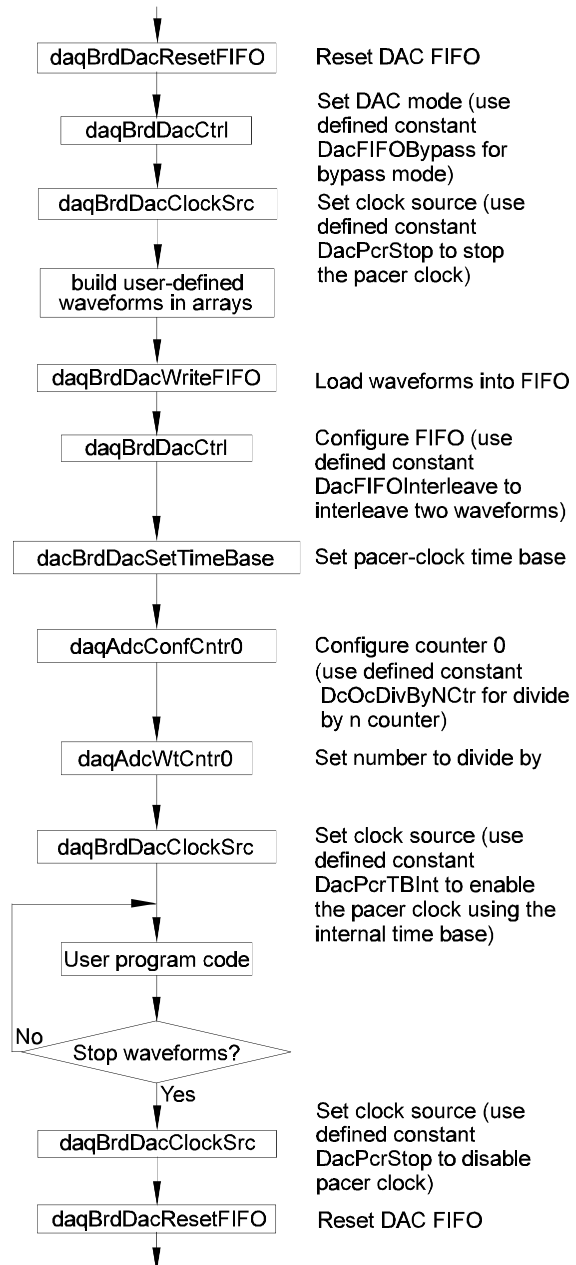
After the user is prompted to stop the waveforms by pressing a key on the computer, the next line stops all the waveforms that have been set up by the QBdaqBrdDacSetMode, QBdaqBrdDacPredefWave, and QBdaqBrdDacUserWave commands and started by the QBdaqBrdDacStart command.

```
PRINT "The wafeforms are being outputted on DACs 0 and 1."
PRINT "Press a key to stop the waveforms and end the program."
WHILE INKEY$="":WEND:PRINT
ret% = QBdaqBrdDacStop%
```

Generating DAC FIFO Waveforms with Hardware-Level Commands (DaqBoard Only)

This program (DACEX3.BAS) demonstrates the use of the DAC FIFO to generate waveforms with hardware-level commands. The programmer must understand the 8255 integrated circuit. After configuration and output enabled, the program continues while the waveforms are generated. Functions used include:

- QBdaqBrdDacResetFIFO%
- QBdaqBrdDacCtrl%(mode%, retransmit%)
- QBdaqBrdDacClockSrc%(source%)
- QBdaqBrdDacSetTimeBase%(frequency %)
- QBdaqAdcConfCntr0%(config%)
- QBdaqAdcWtCntr0%(cntr0%)
- QBdaqBrdDacWriteFIFO%(samples%, storage%())



Note: This example uses the high-speed DAC FIFO to generate waveforms and can only be used with the DaqBoard product line.

We first define a constant for the waveform's size and then dimension an array to build a waveform in.

```
CONST WavePoints  = 512
DIM i%, point%,
 waveBuf%(WavePoints)
```

The first steps are to reset the FIFO. This clears any previous samples from the FIFO.

```
Ret% = QBdaqBrdDacResetFIFO%
```

Set the DAC FIFO to the DaqBook compatible (or FIFO bypass) mode. Note: This step is not required, it is only shown here as a demonstration of how to put the DACs in the DaqBook compatible mode.

```
Ret% =
 QBdaqBrdDacCtrl%(DacFIFOBypass%
 , 0)
```

And turn off the DAC FIFO's pacer clock.

```
Ret% = QBdaqBrdDacClockSrc%(DacPcrStop%)
```

The next series of statements build the waveforms that will be output on DAC channels 0 and 1. The waveform on channel 1 will ramp from the low-voltage reference level to the high-voltage reference level. The waveform on channel 0 will ramp from the high-voltage reference level down to the low-voltage reference level. The waveform samples are interleaved in the buffer with channel 1 samples in the first buffer and channel 0 samples in the second buffer. Note: The voltage references depend on the hardware configuration. (See sections of the manual on your hardware configuration for more information.)

```
FOR x = 0 to WavePoints% - 1 STEP 2
        waveBuf%(x) = point%
        waveBuf%(x+1) = &Hfff - point%
        point% = point% + &H10
NEXT x
```

The next line loads the buffer we have built into the DAC FIFO hardware.

```
Ret% = QBdaqBrdDacWriteFIFO%(WavePoints%, waveBuf%(0) )
```

The next line specifies that the waveform samples are interleaved in the FIFO and that the samples should be re-transmitted when the end of the FIFO is reached.

```
Ret% = QBdaqBrdDacCtrl%(DacFIFOInterleave%, 1)
```

Next we will set the main time base for the DAC pacer clock to 5 MHz; and then, we will configure counter 0 of the 8254 (which the DAC pacer clock passes through) to be a divide by 10 counter. This will give us a update rate of 500 kHz per sample. Since the samples are interleaved in the FIFO, each waveform will be updated at a rate of 250 kHz.

```
Ret% = QBdaqBrdDacSetTimeBase%(TB5Mhz%)
Ret% = QBdaqAdcConfCntr0%(Dc0cDivByNCtr%)
Ret% = QBdaqAdcWtCntr0%(10)
```

To start the waveforms, we will turn on the DAC pacer clock and tell the hardware to use the internal time base for this clock. After the user is prompted to stop the waveforms by pressing a key on the computer, the next line stops all the waveforms that have been set up by stopping the clock pulses to the DAC FIFO and clearing the FIFO with the QBdaqBrdDacResetFIFO command. Issuing either command is enough to stop the waveform generation.

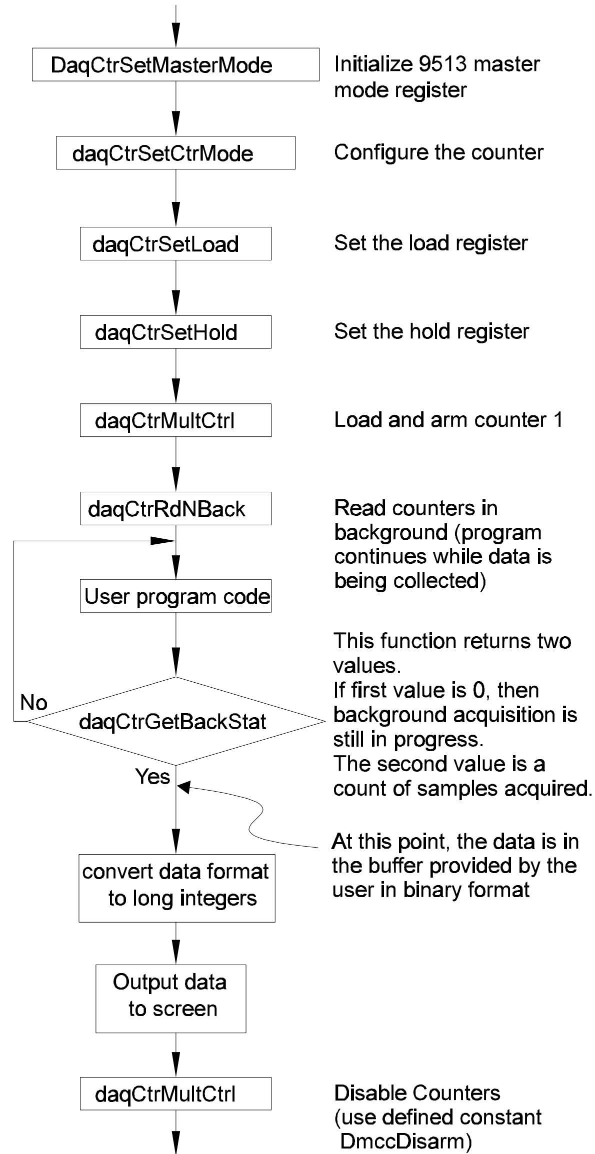
```
PRINT "The wafeforms are being outputted on DACs 0 and 1."
PRINT "Press a key to stop the waveforms and end the program."
WHILE INKEY$="":WEND:PRINT
Ret% = QBdaqBrdDacClockSrc%(DacPcrStop%)
Ret% = QBdaqBrdDacResetFIFO%
```

Background Counter Acquisition Using Interrupts

This program (CTREX2.BAS) sets up a counting acquisition that counts events in the background. First, a signal is generated on the fout pin of P3 and must be physically connected to the interrupt input. After configuring and arming the counter for background acquisition, the program continues in the foreground. The foreground program can poll the background acquisition to determine its status. A 10 Hz square wave will be placed on the oscillator output. Use it to trigger the External Interrupt. **Note**: these counters are only available on the DaqBook/100/200 and DaqBoard/100A/200A. Functions used include:

- QBdaqCtrSetCtrMode%(ctrNum%, gateCtrl%, cntEdge%, cntSource%, specGate%, reload%, cntRepeat%, cntType%, cntDir%, outputCtl%)
- QBdaqCtrSetLoad%(ctrNum%, ctrVal%)
- QBdaqCtrSetHold%(ctrNum%, ctrVal%)
- QBdaqCtrMultCtrl%(ctrCommand%, ctr1%, ctr2%, ctr3%, ctr4%, ctr5%)
- QBdaqCtrRdNlctrValck%(ctr1Buf%(), ctr2Buf%(), ctr3Buf%(), ctr4Buf%(), ctr5Buf%(), count%, startIP0%, cycle%)
- QBdaqCtrGetBackStat%(active%, count%)

```
DIM active%, count%, &, ret%
DIM ctr1Buf%(1000),
 ctr2Buf%(10), ctr3Buf%(10),
 ctr4Buf%(10), ctr5Buf%(10)
```



The QBdaqCtrSetMasterMode function will be used to generate a 10 Hz pulse train on the fout signal located on connector P3. The F5 internal clock of 100 Hz is used as the source for fout, and the divisor is 10. Fout has to be physically connected to the interrupt input with IR Enable enabled (see figure).

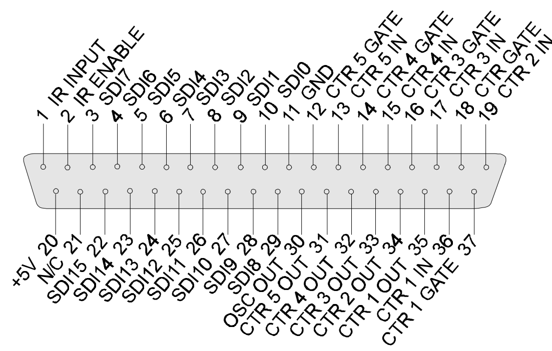
```
ret% = QBdaqCtrSetMasterMode%(10,
 DcsF5%, 0, 0, DtodDisabled%)
```

The QBdaqCtrSetCtrMode function is used to configure counter 1 as an up-counter with a source of F3, the internal 10 kHz clock.

```
ret% = QBdaqCtrSetCtrMode%(1,
 DgcNoGating%, 1, DcsF3%, 0, 0,
 1, 0, 1, DocTCToggled%)
```

When the counter is triggered, it will load itself with the contents of the load register, set to 0 by the QBdaqCtrSetLoad.

```
ret% = QBdaqCtrSetLoad%(1, 0)
```



P3 Pin Numbers and Signal Labels

The Hold Register is reset to zero.

```
ret% = QBdaqCtrSetHold%(1, 0)
```

The next function loads and arms counter 1, which will start it at zero and prepare it to start counting.

```
ret% = QBdaqCtrMultCtrl%(DmccLoadArm%, 1, 0, 0, 0, 0)
```

Read the counters in the background using interrupts count: 10, startIP0: immediate(0), cycle: no(0). QBdaqCtrRdNBack will set up a read of the values of the specified counters in the background and place the 16-bit count values in the supplied arrays. The count argument specifies how many values to store in the arrays. Counter 1 counts at 10 kHz. Readings (samples) are taken every 10 Hz, so the counter is incremented by 1000 each time it is read.

```
ret% = QBdaqCtrRdNBack%(ctr1Buf%(), ctr2Buf%(), ctr3Buf%(), ctr4Buf%(),
  ctr5Buf%(), 10, 0, 0)
```

After the background read command is called, the program continues on the next line of execution. In our program, we immediately go into a polling loop to check the status of the background operation.

```
DO
  ret% = QBdaqCtrGetBackStat%(active%, count%)
  LOCATE 6, 1
  PRINT "Transfer in progress.  "; count%; " samples acquired"
LOOP WHILE active% <>0
PRINT "Acquisition complete.": PRINT
```

QuickBASIC's integer values are signed, ranging from -32766 to +32767. The values returned from the 16-bit counters are not signed, ranging from 0 to 65,535. The integer values must be converted to long integers to be properly interpreted.

```
FOR x = 0 TO 9
  longBuf&(x) = ctr1Buf%(x)
  If longBuf&(x) < 0 Then longBuf&(x) = longBuf&(x) + 65536
NEXT
PRINT "data: "
FOR x = 0 TO 9
PRINT USING "##  ######"; x + 1; longBuf&(x)
NEXT
```

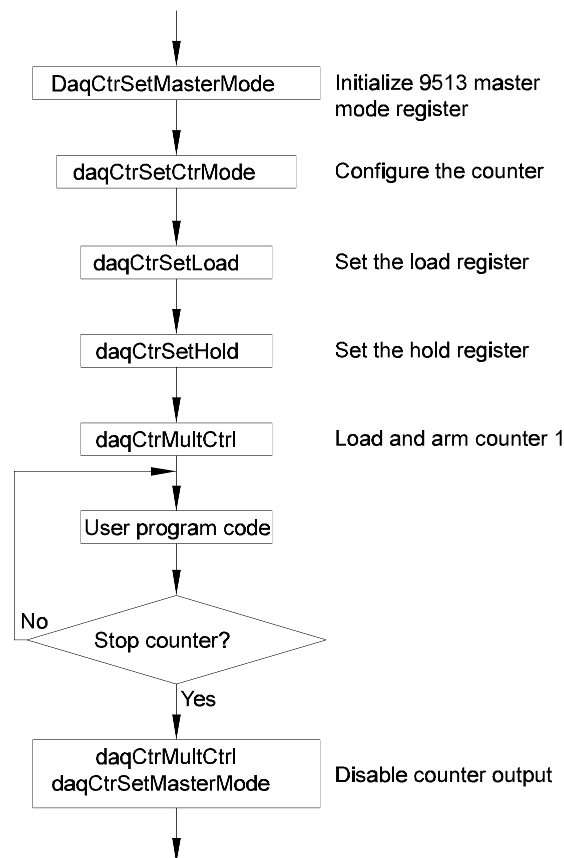
Next, stop and disarm all of the counters.

```
ret% = QBdaqCtrMultCtrl%(DmccDisarm%, 1, 0, 0, 0, 0)
```

Variable Rate, Variable Duty-Cycle Square-Wave Output

This program (CTREX1.BAS) demonstrates the use of the counter/timer section of a DaqBook/100/200 or DaqBoard/100A/200A with the P3 port. After configuring the counter and setting the load and hold registers, the counter is armed. At this point, program execution continues while the counter outputs the signal. This example generates a variable rate, variable duty-cycle square wave. Functions used include:

- QBdaqCtrSetMasterMode%(foutDiv%, foutSource%, comp1%, comp2%, tod%)
- QBdaqCtrSetCtrMode%(ctrNum%, gateCtrl%, cntEdge%, cntSource%, specGate%, reload%, cntRepeat%, cntType%, cntDir%, outputCtl%)
- QBdaqCtrSetHold%(ctrNum%, ctrVal%)
- QBdaqCtrSetLoad%(ctrNum%, ctrVal%)
- QBdaqCtrMultCtrl%(ctrCommand%, ctr1%, ctr2%, ctr3%, ctr4%, ctr5%)



Initialize the 9513 master mode register fout divider: 10, fout source: DcsF2 (100 kHz), compare1: no, compare 2: no, time of day disabled. This will place a 10 kHz pulse on the oscillator output. The QBdaqCtrSetMasterMode function will initialize the counter/timer section and configure several of its parameters. This is a system-wide function which affects all 5 counter timers. **Note**: for a complete understanding of counter/timer operation, read the data book on the 9513 chip supplied by AMD. Aside from initializing the counter/timer section, this application does not use most of the capabilities of the QBdaqCtrSetMasterMode function. The first two arguments in this function select a clock source for the fout signal found on connector P3, then select a divider for that signal. F2 in this application is a fixed, internal frequency source of 100 kHz. Our example divides this fixed frequency by 10 yielding a signal on fout of 10 kHz.

```
ret% = QBdaqCtrSetMasterMode%(10, DcsF2%, 0, 0, DtodDisabled%)
```

The QBdaqCtrSetCtrMode function configures an individual counter in the 9513. The first argument specifies the counter to be configured, the second specifies the internal operation of the gate control. Our application does not use the gate, so it is disabled. The fixed 100 kHz internal clock (F1) is used as the source. By setting the reload parameter to 1, the counter will use the ’load’ register and the ’hold’ register to generate the pulse train. When the counter is armed, the ’load’ register value is loaded then decremented on every edge of the F1 clock. The output signal will be high during this phase. When the terminal count is reached, the ’hold’ register is loaded then decremented on every edge of the F1 clock. The output signal is low during this phase. If the reload argument is set to 0, only the ’load’ register is used, always yielding a 50% duty-cycle pulse train. The cntRepeat argument specifies whether the pulse train should execute once or repeat continuously. The counter interprets the load and load register as either binary or BCD depending on the value of the cntType argument. The cntDir specifies whether the internal counter should count up or down to reach the terminal count. A value of 5 counted down has the same effect as a value of 65,530 counted up.

```
ret% = QBdaqCtrSetCtrMode%(1, DgcNoGating%, 1, DcsF1%, 0, 1, 1, 0, 0,
  DocTCToggled%)
```

Set the load register to 75 and the hold register to 25. This produces a high duty-cycle of 75% and (with 100 total counts to count down) a frequency of 10 kHz.

```
ret% = QBdaqCtrSetLoad%(1, 75)
```

```
ret% = QBdaqCtrSetHold%(1, 25)
```

The QBdaqCtrMultCtrl function will arm counter 1.

```
ret% = QBdaqCtrMulCtrl% (DmccDisarm%, 1, 0, 0, 0, 0)
ret% = QBdaqCtrSetMasterMode% (0, 0, 0, 0, DtodDisabled%)
Print "Outputs disabled."
```

Continue the pulse train until a key is pressed.

```
PRINT "A 10 kHz 25% duty-cycle square wave is on the counter 1 output."
PRINT "press any key to halt counter 1 output."
WHILE INKEY$ = "": WEND
```

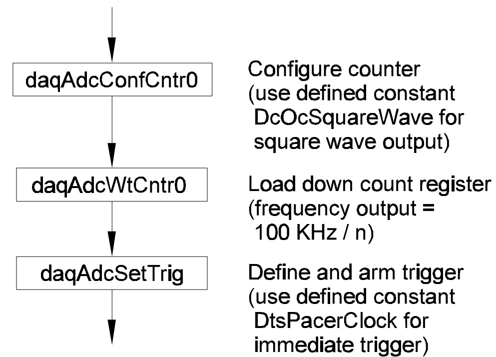
QBdaqCtrMultCtrl will stop the pulse train.

```
ret% = QBdaqCtrMultCtrl%(DmccDisarm%, 1, 0, 0, 0, 0)
```

Single Square-Wave Output

This program (ADCEX5.BAS) demonstrates the use of the 8254's counter 0 (accessible via the DaqBook/DaqBoard P1 connector—not available via Daq PCMCIA). After configuring the control register and loading the down-count register of the counter, the trigger is defined and armed. At this point, program execution continues while the counter outputs the signal. This program will initialize the Daq* hardware, then generate a 50 kHz pulse train on the counter 0 signal of P1. Functions used include:

- QbdaqAdcConfCntr0%(config%)
- QbdaqAdcWtCntr0%(cntr0%)
- QBdaqAdcSetTrig%(source%, oneShot%, level%, ctr0Mode%, pacerMode%)



To operate the counter, it must first be configured by the QBdaqAdcConfCntr function. In this program, the counter is configured to generate a square wave.

```
ret% = QBdaqAdcConfCntr0%(Dc0cSquareWave%)
```

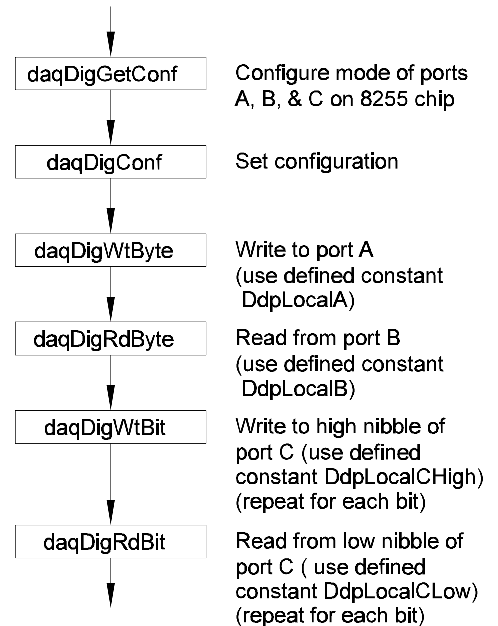
This counter contains a down-counter which effectively divides the counter source by the loaded count. The next line loads the count-down register with a 2, which divides the source count (100 kHz) by 2 to equal a pulse train of 50 kHz.

```
ret% = QBdaqAdcWtCntr0%(2)
```

To start the pulse train, it must be armed and triggered by the QBdaqAdcSetTrig function. This function also arms an analog input acquisition, if configured, which will be synchronized with the start of this pulse train. With a trigger source of the internal pacer clock, the pulse train will start immediately.

```
ret% = QBdaqAdcSetTrig%(DtsPacerClock%, 0, 0, 1, 0)
```

Digital I/O on P2



This program (DIGEX1.BAS) demonstrates the functions controlling digital I/O on connector P2 of the DaqBook/100/200 and DaqBoard/100A/200A. First, the 3 digital ports on the 8255 are configured as input, output (or both in the case of port C); then, appropriate I/O commands are issued. Functions used include:

- QBdaqDigConf%(port%, config%)
- QBdaqDigWtByte%(port%, byteVal%)
- QBdaqDigRdByte%(port%, byteVal%)
- QBdaqDigWtBit%(port%, bitNum%, bitVal%)
- QBdaqDigRdBit%(port%, bitNum%, bitVal%)

```
DIM byteVal%, bitVal%
ret% = QBdaqDigGetConf% (0,1,0,1,
  config%)
```

The function QBdaqDigGetConf returns the appropriate configuration value to use inQBdaqDigConfig. The QBdaqDigConf function tells the Daq* whether the digital I/O is located in the base unit, or on an expansion card. The second argument is the byte to be sent to the 8255's control register.

```
ret% = QBdaqDigConf%(DdcLocal%, configure)
```

Write hex 55 to port A on the Daq*'s base unit.

```
ret% = QBdaqDigWtByte%(DdpLocalA%, &H55)
```

Read port B and put the value into the variable byteVal%.

```
ret% = QBdaqDigRdByte%(DdpLocalB%, byteVal%)
PRINT "The value on digital port B : &H"; HEX$(byteVal%): PRINT
```

The following lines write to individual bits on the base unit's port C.

```
ret% = QBdaqDigWtBit%(DdpLocalCHigh%, 0, 1)
ret% = QBdaqDigWtBit%(DdpLocalCHigh%, 1, 0
ret% = QBdaqDigWtBit%(DdpLocalCHigh%, 2, 1)
ret% = QBdaqDigWtBit%(DdpLocalCHigh%, 3, 0)
PRINT "The high nibble of digital port C set to : 0101": PRINT
```

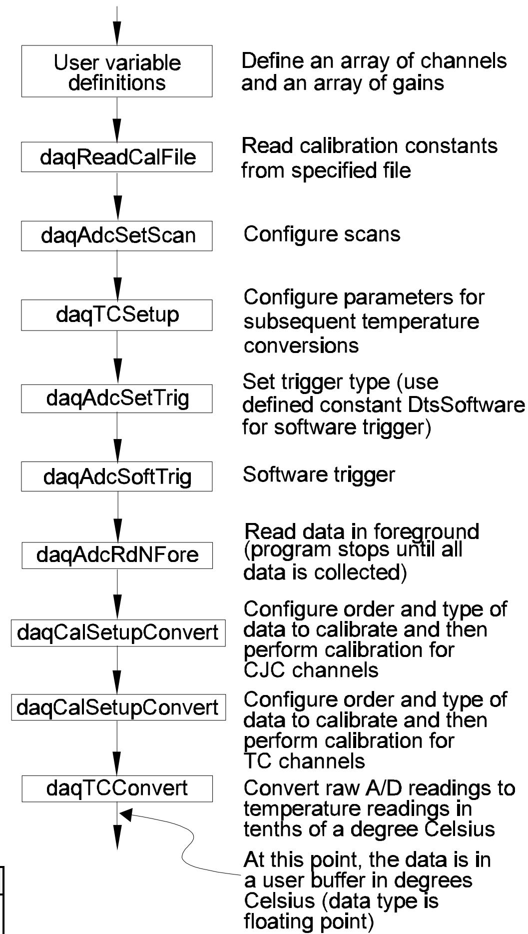
The next lines read the low nibble of port C on the base unit.

```
FOR x% = 0 TO 3
  ret% = QBdaqDigRdBit%(DdpLocalCLow%, x%, bitVal%)
PRINT "The value on bit "; x%; " of digital port C : &H"; HEX$(bitVal%)
NEXT x%
```

Temperature Measurements Using Single TC Type on a Single DBK19 Card

The 4 examples follow the same command sequence except for their arguments or program code for data output:

- Example 1 demonstrates repeated measurements of TC inputs.
- Example 2 demonstrates block averaging of the same TC inputs as example one. This example performs each reading 5 times and averages them together.
- Example 3 uses the same data as example 2, but rather than averaging the 5 scans, it outputs each of them to the screen.
- Example 4 gather the same data as the previous examples but applies a moving average to that data.



DBK19 Example 1: Type J Thermocouples

In this example, we wish to repeatedly measure the temperatures sensed by 2 type J thermocouples attached to channels 18 and 19 through a DBK19 card. The DBK19 CJC signal is always the first signal on the card and the shorted channel (used for zero compensation) is always the second signal on the card. In this case, they are on channels 16 and 17. First we list the configuration (see table).

Card	Channel	Channel Type
DBK19	16 17 18 19	CJC Shorted (zero) Type J Type J
Local	0 1-15	Used for DBK19 Free for other uses

Now we must specify the scan, the sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the temperature channels; the scan must first include the CJC zero, thermocouple zero and CJC, and then the temperature channels (see table). The thermocouples need not be scanned in any particular order. We might have specified channel 18 before channel 17, but keeping things in order will make the calibration easier.

Scan Position	Channel Type	Channel
0	CJC Zero	17
1	Type J Zero	17
2	CJC	16
3	Type J	18
4	Type J	19

For each scan position, we must specify the PGA gain code. Assuming the Daq* is configured for bipolar operation (to allow measurement of temperatures below the temperature at the DBK19 card), we choose the gain codes from the table and add them to the scan description.

Scan Position	Channel Type	Channel	Gain Code
0	CJC Zero	17	Dbk19BiCJC
1	Type J Zero	17	Dbk19BiTypeJ
2	CJC	16	Dbk19BiCJC
3	Type J	18	Dbk19BiTypeJ
4	Type J	19	Dbk19BiTypeJ

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input					
	Readings				
Measurement	0	1	2	3	4
1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	CJC Zero	Type J Zero	CJC	Type J	Type J
3	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...
10	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

Now we can configure the Daq* with this information:

```
DIM chans%(5), gains%(5), buf%(5), temp%(2)

' read calibration file
ret% = QBdaqReadCalFile%("daqbook.cal")

' Set array of channels and gains
chans%(0) = 17
gains%(0) = Dbk19BiCJC%
chans%(1) = 17
gains%(1) = Dbk19BiTypeJ%
chans%(2) = 16
gains%(2) = Dbk19BiCJC%
chans%(3) = 18
gains%(3) = Dbk19BiTypeJ%
chans%(4) = 19
gains%(4) = Dbk19BiTypeJ%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 5)

' Temperature measurements require 16-bit data
ret% = QBdaqAdcSetTag%(1)

' Configure for the conversion to temperatures
ret% = QBdaqTCSetup%(5, 2, 2, Dbk19TCTypeJ%, 1, 1)

FOR i = 1 TO 10
 ' Define and arm trigger :
 ret% = QBdaqAdcSetTrig%(DtsSoftware%, 1, 0, 0, 0)

 ' Trigger
 ret% = QBdaqAdcSoftTrig%

 ' Read the data
 ret% = QBdaqAdcRdNFore%( buf%(), 1)

 ' Calibrate the CJC -1 channel starting at position 2
 ret% = QBdaqCalSetupConvert%(5, 2, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
 1, buf%(), 1)

 ' Calibrate the TCs -2 channel starting at position 3
 ret% = QBdaqCalSetupConvert%(5, 3, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
 18, 1, 1, buf%(), 1)
```

```
  ' Convert 'scans' scans of counts to two temperatures
  ret% = QBdaqTCConvert%( buf%(), 1, temp%(), 2)

  'Display the temperatures
  PRINT "Channel 18: "; .01 * temp%(0); "   Channel 19: "; .01 * temp%(1)
NEXT i

FUNCTION IntToUint (IntVal AS INTEGER)
  'Converts 16-bit signed integer to unsigned integer.
  IF 0 <= IntVal THEN
    IntToUint = IntVal     'Return positive values with no change
  ELSE
    IntToUint = 65535 + CLNG(IntVal) + 1 'Convert negative values to
                                          'positive
  END IF
END FUNCTION
```

DBK19 Example 2: Block Averaged TC readings

In this example, we want to acquire the same information as in example 1, except we wish to use the Daq*'s high speed to reduce the noise by taking each reading 5 times and averaging them together.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

Assuming we are using the same thermocouples connected in the same way, the scan configuration is like example 1:

```
DIM chans%(5), gains%(5), buf%(25), temp%(2)

' read calibration file
ret% = QBdaqReadCalFile%("daqbook.cal")

' Set array of channels and gains
chans%(0) = 17
gains%(0) = Dbk19BiCJC%
chans%(1) = 17
gains%(1) = Dbk19BiTypeJ%
chans%(2) = 16
gains%(2) = Dbk19BiCJC%
chans%(3) = 18
gains%(3) = Dbk19BiTypeJ%
chans%(4) = 19
gains%(4) = Dbk19BiTypeJ%
```

```
' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 5)

' Temperature measurements require 16-bit data
ret% = QBdaqAdcSetTag%(1)

' Configure for the conversion to temperatures
ret% = QBdaqTCSetup%(5, 2, 2, Dbk19TCTypeJ%, 1, 0)

FOR i = 1 TO 10
  ' Define and arm trigger :
  ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)

  ' Trigger
  ret% = QBdaqAdcSoftTrig%

  ' Read the data
  ret% = QBdaqAdcRdNFore%(buf%(), 5)

  ' Calibrate the CJC -1 channel starting at position 2
  ret% = QBdaqCalSetupConvert%(5, 2, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
  1, counts%(), 5)

  ' Calibrate the TCs -2 channel starting at position 3
  ret% = QBdaqCalSetupConvert%(5, 3, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
  18, 1, 1, buf%(), 5)

  ' Convert 'scans' scans of counts to two temperatures
  ret% = QBdaqTCConvert%(buf%(), 5, temp%(), 2)

  'Display the temperatures
  PRINT "Channel 18: "; .01 * temp%(0); "   Channel 19: "; .01 * temp%(1)
NEXT i

' Close DaqBook/100 and end program
ret% = QBdaqClose%
END

ErrorHandler:
PRINT "ERROR! Program aborted"
PRINT "BASIC Error :"; ERR
IF ERR = 100 THEN PRINT "DaqBook/100 Error : "; HEX$(daqErrno%)
END

FUNCTION IntToUint (IntVal AS INTEGER)
  'Converts 16-bit signed integer to unsigned integer.
  IF 0 <= IntVal THEN
    IntToUint = IntVal   'Return positive values with no change
  ELSE
    IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                             'positive
  END IF
END FUNCTION
```

DBK19 Example 3: Multiple Sequential Measurement

In this example, we wish to collect the same data as in example 2; but instead of averaging the groups of 5 consecutive scans, we want to convert each scan's measurements into individual temperature values.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert			
		Results	
Measurement	Scan	0	1
1	1	Temp °C	Temp °C
1	2	Temp °C	Temp °C
1	3	Temp °C	Temp °C
1	4	Temp °C	Temp °C
1	5	Temp °C	Temp °C
2	1	Temp °C	Temp °C
2	2	Temp °C	Temp °C
...	...	...	...
10	5	Temp °C	Temp °C

The scan setup is the same as in examples 1 and 2 and is omitted here for brevity. We again configure for the conversion to temperatures, this time (as in example 1) specifying no averaging:

```
DIM chans%(5), gains%(5), buf%(25), temp%(5, 2)
CLS
PRINT "DBK19_3": PRINT

'Set error handler and initialize DaqBook/100
ret% = QBdaqSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBdaqInit%(LPT1%, 7)

' read calibration file
ret% = QBdaqReadCalFile%("daqbook.cal")

' Set array of channels and gains
chans%(0) = 17
gains%(0) = Dbk19BiCJC%
chans%(1) = 17
gains%(1) = Dbk19BiTypeJ%
chans%(2) = 16
gains%(2) = Dbk19BiCJC%
chans%(3) = 18
gains%(3) = Dbk19BiTypeJ%
chans%(4) = 19
gains%(4) = Dbk19BiTypeJ%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 5)

' Temperature measurements require 16-bit data
ret% = QBdaqAdcSetTag%(1)
```

```
' Configure for the conversion to temperatures
ret% = QBdaqTCSetup%(5, 2, 2, Dbk19TCTypeJ%, 1, 1)

FOR i = 1 TO 10
 ' Define and arm trigger :
 ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)

 ' Trigger
 ret% = QBdaqAdcSoftTrig%

 ' Read the data
 ret% = QBdaqAdcRdNFore%(buf%(), 5)

 ' Calibrate the CJC -1 channel starting at position 2
 ret% = QBdaqCalSetupConvert%(5, 2, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
 1, buf%(), 5)

 ' Calibrate the TCs -2 channel starting at position 3
 ret% = QBdaqCalSetupConvert%(5, 3, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
 18, 1, 1, buf%(), 5)

 ' Convert 'scans' scans of counts to two temperatures
 ret% = QBdaqTCConvert%(buf%(), 5, temp%(), 10)

 FOR j = 1 TO 5
   'Display the temperatures
   PRINT "Channel 18: "; .01 * temp%(j, 0); "   Channel 19: "; .01 *
 temp%(j, 1)
 NEXT j
NEXT i

FUNCTION IntToUint (IntVal AS INTEGER)
   'Converts 16-bit signed integer to unsigned integer.
 IF 0 <= IntVal THEN
   IntToUint = IntVal    'Return positive values with no change
 ELSE
   IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                             'positive
 END IF
END FUNCTION
```

DBK19 Example 4: Moving Averaged Measurements

In this example, we wish to collect the same data as in example 3; but to reduce noise, we will use a moving average to average consecutive triplets of scans.

The following tables show the raw data input and the resulting temperature data output for this sample program.

Raw Data Input						
		Readings				
Measurement	Scan	0	1	2	3	4
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J
...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J

Results After daqTCConvert		
	Results	
Measurement	0	1
1	Temp °C	Temp °C
2	Temp °C	Temp °C
3	Temp °C	Temp °C
...	...	...
10	Temp °C	Temp °C

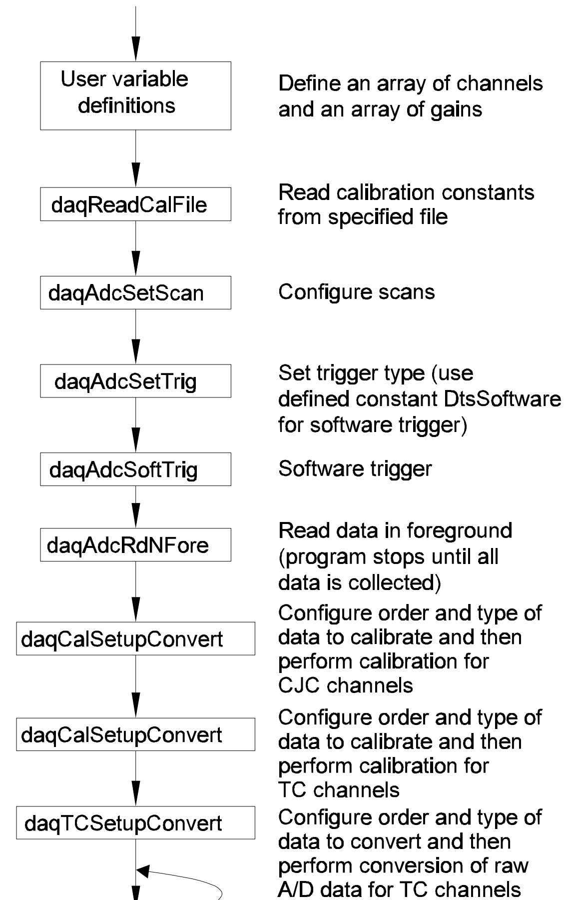
The scan setup is the same as in the previous examples and is omitted here for brevity. We again configure for the conversion to temperatures, this time (as in example 1) specifying moving averaging of 3 scans.

```
DECLARE FUNCTION IntToUint! (IntVal AS INTEGER)
' DBK19_4.BAS
' In this example we will collect the same data as example three but we
' will use a moving average to average consecutive triplets of scans.
'$INCLUDE: 'daqbook.bi'
DIM chans%(5), gains%(5), buf%(25), temp%(5, 2)
CLS
PRINT "DBK19_4": PRINT

'Set error handler and initialize DaqBook/100
ret% = QBdaqSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBdaqInit%(LPT1%, 7)

' read calibration file
ret% = QBdaqReadCalFile%("daqbook.cal")

' Set array of channels and gains
chans%(0) = 17
gains%(0) = Dbk19BiCJC%
chans%(1) = 17
gains%(1) = Dbk19BiTypeJ%
chans%(2) = 16
gains%(2) = Dbk19BiCJC%
chans%(3) = 18
gains%(3) = Dbk19BiTypeJ%
chans%(4) = 19
gains%(4) = Dbk19BiTypeJ%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 5)
```

```
' Temperature measurements require 16-bit data
ret% = QBdaqAdcSetTag%(1)

' Configure for the conversion to temperatures
ret% = QBdaqTCSetup%(5, 2, 2, Dbk19TCTypeJ%, 1, 3)

FOR i = 1 TO 10
 ' Define and arm trigger :
 ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)

 ' Trigger
 ret% = QBdaqAdcSoftTrig%

 ' Read the data
 ret% = QBdaqAdcRdNFore%(buf%(), 5)

 ' Calibrate the CJC -1 channel starting at position 2
 ret% = QBdaqCalSetupConvert%(5, 2, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
 1, buf%(), 5)

 ' Calibrate the TCs -2 channel starting at position 3
 ret% = QBdaqCalSetupConvert%(5, 3, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
 18, 1, 1, buf%(), 5)

 ' Convert 'scans' scans of buf to two temperatures
 ret% = QBdaqTCConvert%(buf%(), 5, temp%(), 10)

 FOR j = 1 TO 5
   'Display the temperatures
   PRINT "Channel 18: "; .01 * temp%(j, 0); "   Channel 19: "; .01 *
 temp%(j, 1)
 NEXT j
NEXT i

FUNCTION IntToUint (IntVal AS INTEGER)
 'Converts 16-bit signed integer to unsigned integer.
 IF 0 <= IntVal THEN
   IntToUint = IntVal  'Return positive values with no change
 ELSE
   IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                            'positive
 END IF
END FUNCTION
```

Temperature Measurements Using Multiple TC Types on Multiple DBK19 Cards

This program demonstrates temperature acquisitions using multiple TC types and multiple DBK19 cards. The two commands daqTCSetup and daqTCConvert have been combined into the one daqTCSetupConvert command. The sequence of the last 3 blocks on the flow chart must be used multiple times, once for each card, and if there is multiple TC types on a card, once for each TC type on that card.



In this example, we wish to repeatedly measure the temperatures sensed by 2 Type J and 2 Type K thermocouples attached through 1 DBK19 card and 2 more Type J thermocouples attached through another DBK19. The DBK19 CJC signal is always the first signal on the card, and the shorted channel (used for zero compensation) is always the second channel on the card. First we list the configuration:

Now we must specify the scan, the sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the temperature channels; and so, the scan must first include the CJC and then immediately the temperature channels (see table).

Card	Channel	Channel Type
DBK19	16	CJC
	17	Shorted (zero)
	18	Type J
	19	Type J
	20	Type K
	21	Type K
DBK19	32	CJC
	33	Shorted (zero)
	34	Type J
	35	Type J
Local	0-1	Used for DBK19
	2-15	Free for other uses

Scan Position	Channel Type	Channel	Gain Code
0	CJC Zero	17	Dbk19BiCJC
1	Type J Zero	17	Dbk19BiTypeJ
2	CJC	16	Dbk19BiCJC
3	Type J	18	Dbk19BiTypeJ
4	Type J	19	Dbk19BiTypeJ
5	CJC Zero	17	Dbk19BiCJC
6	Type K	17	Dbk19BiTypeK
7	ZeroCJC	16	Dbk19BiCJC
8	Type K	20	Dbk19BiTypeK
9	Type K	21	Dbk19BiTypeK
10	CJC Zero	33	Dbk19BiCJC
11	Type J Zero	33	Dbk19BiTypeJ
12	CJC	32	Dbk19BiCJC
13	Type J	34	Dbk19BiTypeJ
14	Type J	35	Dbk19BiTypeJ

The thermocouples are separated in the scan by type. The readings from each type are consecutive and immediately preceded by their CJC Zero, thermocouple zero, and CJC readings for calculation reference. It is not appropriate to consolidate the 4 Type J thermocouples because they are connected through 2 different DBK19s. Each DBK19 has its own CJC and offset errors as a reference for thermocouples attached to that DBK19.

For each scan position, we must specify the PGA gain. Assuming the Daq* is configured for bipolar operation (to allow measurement of temperatures below the temperature at the DBK19 cards), we choose the gain codes from the table above and add them to the scan description.

The following tables show the raw data input and the resulting temperature data output for this sample program.**Raw Data Input**

Measure-ment	Scan	Readings 0	1	2	3	4	5	6	7	8	9	10	(1-3)	14
1	1	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	2	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	3	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	4	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
1	5	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
2	1	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
2	2	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
10	5	CJC Zero	Type J Zero	CJC	Type J	Type J	CJC Zero	Type K Zero	CJC	Type K	Type K	CJC Zero	...	Type J

Results After daqTCConvert

Measurement	Results 0	1	2	3	4	5
1	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C
2	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C
...	...	...	...	...	...	...
10	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C	Temp °C

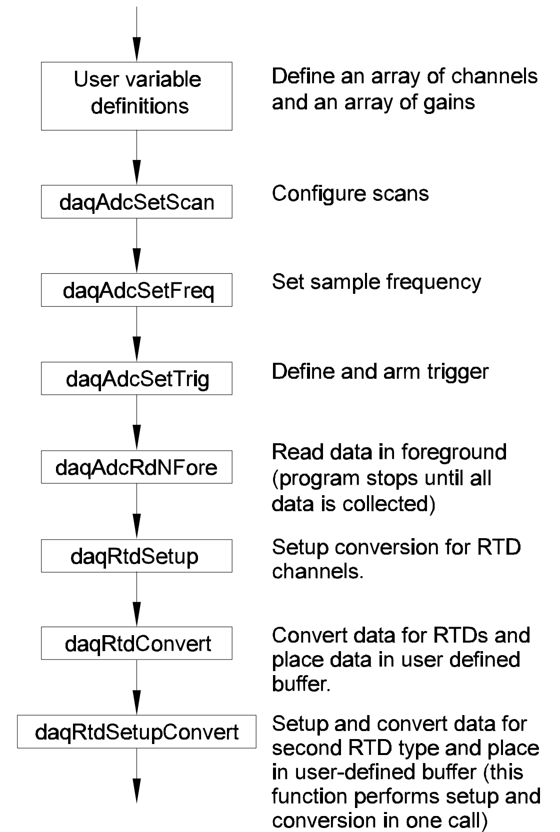
Now we can configure the DaqBook/DaqBoard with this information:

```
DIM chans%(15), gains%(15), buf%(75), temp1%(2), temp2%(2), temp3%(2)
' read calibration file
ret% = QBdaqReadCalFile%("daqbook.cal")
' Set array of channels and gains
chans%(0) = 17: gains%(0) = Dbk19BiCJC%
chans%(1) = 17: gains%(1) = Dbk19BiTypeJ%
chans%(2) = 16: gains%(2) = Dbk19BiCJC%
chans%(3) = 18: gains%(3) = Dbk19BiTypeJ%
chans%(4) = 19: gains%(4) = Dbk19BiTypeJ%
chans%(5) = 17: gains%(5) = Dbk19BiCJC%
chans%(6) = 17: gains%(6) = Dbk19BiTypeK%
chans%(7) = 16: gains%(7) = Dbk19BiCJC%
chans%(8) = 20: gains%(8) = Dbk19BiTypeK%
chans%(9) = 21: gains%(9) = Dbk19BiTypeK%
chans%(10) = 33: gains%(10) = Dbk19BiCJC%
chans%(11) = 33: gains%(11) = Dbk19BiTypeJ%
chans%(12) = 32: gains%(12) = Dbk19BiCJC%
chans%(13) = 34: gains%(13) = Dbk19BiTypeJ%
chans%(14) = 35: gains%(14) = Dbk19BiTypeJ%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 15)
' Temperature measurements require 16-bit data
ret% = QBdaqAdcSetTag%(1)
FOR i = 1 TO 10
 ' Define and arm trigger :
 ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)
 ' Trigger
 ret% = QBdaqAdcSoftTrig%

 ' Read the data
 ret% = QBdaqAdcRdNFore%(buf%(), 5)

 ' Calibrate the CJC -1 channel starting at position 2
```

```
  ret% = QBdaqCalSetupConvert%(15, 2, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
  1, buf%(), 15)

  ' Calibrate the TCs -2 channel starting at position 3
  ret% = QBdaqCalSetupConvert%(15, 3, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
  18, 1, 1, buf%(), 15)

  ' Calibrate the TCs -1 channel starting at position 7
  ret% = QBdaqCalSetupConvert%(15, 7, 1, DcalTypeCJC%, Dbk19BiCJC%, 16, 1,
  1, buf%(), 15)

  ' Calibrate the TCs -2 channel starting at position 8
  ret% = QBdaqCalSetupConvert%(15, 8, 2, DcalTypeDefault%, Dbk19BiTypeK%,
  20, 1, 1, buf%(), 15)

  ' Calibrate the CJC -1 channel starting at position 12
  ret% = QBdaqCalSetupConvert%(15, 12, 1, DcalTypeCJC%, Dbk19BiCJC%, 32,
  1, 1, buf%(), 15)

  ' Calibrate the TCs -2 channel starting at position 13
  ret% = QBdaqCalSetupConvert%(15, 13, 2, DcalTypeDefault%, Dbk19BiTypeJ%,
  34, 1, 1, buf%(), 15)

  ' Convert 'scans' scans of counts to two temperatures
  ret% = QBdaqTCSetupConvert%(15, 2, 2, Dbk19TCTypeJ%, 1, 0, buf%(), 15,
  temp1%(), 2)

  'Display the temperatures
  PRINT "Channel 18: "; .01 * temp1%(0); "   Channel 19: "; .01 *
  temp1%(1)

  ' Convert 'scans' scans of counts to two temperatures
  ret% = QBdaqTCSetupConvert%(15, 3, 2, Dbk19TCTypeK%, 1, 0, buf%(), 15,
  temp2%(), 2)

  'Display the temperatures
  PRINT "Channel 20: "; .01 * temp2%(0); "   Channel 21: "; .01 *
  temp2%(1)

  ' Convert 'scans' scans of counts to two temperatures
  ret% = QBdaqTCSetupConvert%(15, 6, 2, Dbk19TCTypeJ%, 1, 0, buf%(), 15,
  temp3%(), 2)

  'Display the temperatures
  PRINT "Channel 34: "; .01 * temp3%(0); "   Channel 35: "; .01 *
  temp3%(1)
NEXT i

FUNCTION IntToUint (IntVal AS INTEGER)
  'Converts 16-bit signed integer to unsigned integer.
  IF 0 <= IntVal THEN
    IntToUint = IntVal 'Return positive values with no change
  ELSE
    IntToUint = 65535 + CLNG(IntVal) + 1
            'Convert negative values to positive
  END IF
END FUNCTION
```

Temperature Measurements Using Multiple RTDs on a Single DBK9 Card

This program demonstrates temperature acquisitions using multiple RTD types and a single DBK9 card. After this program configures and arms the DBK card, it begins acquiring data in the foreground acquisition mode. At this point, program execution is suspended until all the data is gathered. The program demonstrates the conversion of data as both a two-step process and a single-step process. Note the conversion routines need to be called for each type of RTD in the scan. The temperature at the RTD is derived from 4 voltage values.



In this example, we wish to acquire some temperature readings from 3 RTDs. There are two 100-ohm RTDs attached to channels 16 and 17 of the DBK9 and one 1000-ohm RTD attached to channel 18. The configuration looks like this:

First we must specify the scan sequence of channel numbers and gains that are to be gathered as one burst of readings. In this example, we are only interested in the RTD channels. The scan must include the 4 voltage readings in the correct order for each channel (see table).

Note that the RTDs need not be scanned in any particular order but the 4 readings for each RTD must be placed in the scan sequentially. We might have specified channel 17 before channel 16. It is best to group all the RTD reading groups of the same value together because this makes using the temperature conversion functions easier.

Now we can configure the Daq* with this information. First we will define some constants that will make the program easier to modify.

Card	Channel	Channel Type
DBK9	16 17 18	100 ohm RTD 100 ohm RTD 1000 ohm RTD
Local	0 1-15	Used for DBK9 Free for other uses

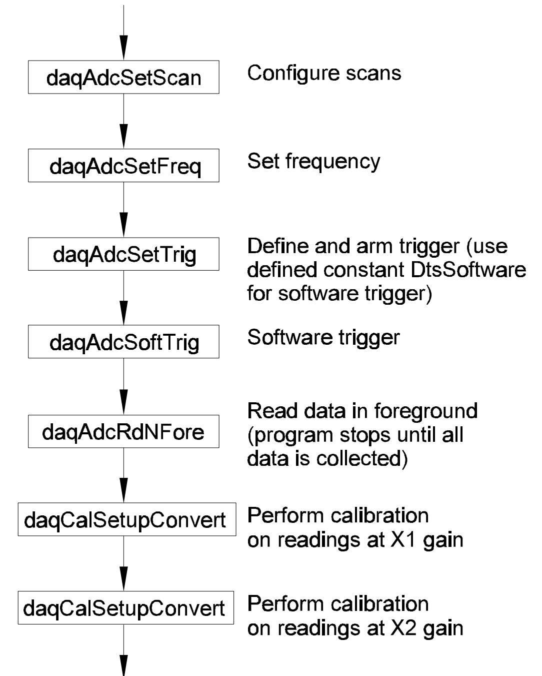
Scan Position	Channel Number	*Channel Gain
0	16	Dbk9VoltageA
1	16	Dbk9VoltageB
2	16	Dbk9VoltageD
3	16	Dbk9VoltageD
4	17	Dbk9VoltageA
5	17	Dbk9VoltageB
6	17	Dbk9VoltageD
7	17	Dbk9VoltageD
8	18	Dbk9VoltageA
9	18	Dbk9VoltageB
10	18	Dbk9VoltageD
11	18	Dbk9VoltageD

* These are not actual gains. They are used to select voltages A-D for each RTD channel.

```
RdsPerRtd = 4
NRtds = 3
FirstRtdChanNo = 16
Nscans% = 10
ReadingsPerScan% = NRtds * RdsPerRtd
BufSize = Nscans% * ReadingsPerScan%
VaOffset = 0
VbOffset = 1
VcOffset = 2
VdOffset = 3

DIM chans%(ReadingsPerScan%), gains%(ReadingsPerScan%), buf%(BufSize),
  temp1%(Nscans% * 2), temp2%(Nscans%)
```

```
' Set array of channels and gains
FOR RTD% = 0 TO NRtds - 1
  FOR j% = 0 TO RdsPerRtd
   chans%(RTD% * RdsPerRtd + j) = RTD% + FirstRtdChanNo
  NEXT j%
  gains%(RTD% * RdsPerRtd + VaOffset) = Dbk9VoltageA%
  gains%(RTD% * RdsPerRtd + VbOffset) = Dbk9VoltageB%
  gains%(RTD% * RdsPerRtd + VcOffset) = Dbk9VoltageD%
  gains%(RTD% * RdsPerRtd + VdOffset) = Dbk9VoltageD%
NEXT RTD%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), ReadingsPerScan%)

' Set sampling freq.
ret% = QBdaqAdcSetFreq%(10000)

' Define and arm trigger :
ret% = QBdaqAdcSetTrig%(DtsPacerClock%, 0, 0, 0, 0)

' Read the data
ret% = QBdaqAdcRdNFore%(buf%(), Nscans%)

' Setup the conversion for the first two RTDs
ret% = QBdaqRtdSetup%(ReadingsPerScan%, 0, 2, Dbk9RtdType100%, 1)

' Convert the data for the first two RTDs
ret% = QBdaqRtdConvert%(buf%(), Nscans%, temp1%(), Nscans% * 2)

' Setup and convert the data for the 1000 ohm RTD in one step
ret% = QBdaqRtdSetupConvert%(ReadingsPerScan%, 8, 1, Dbk9RtdType1K%, 1,
 buf%(), Nscans%, temp2%(), Nscans%)

'Display the temperatures for the 100 ohm RTDs
FOR scan = 0 TO Nscans%
  PRINT "Scan: "; scan
  FOR x = 0 TO 2
   tmptemperature! = (temp1%(scan * 1 + x)) / 10
   PRINT tmptemperature!; "   ";
  NEXT x
  PRINT
  NEXT scan

'Display the temperatures for the 1000 ohm RTDs
FOR scan = 0 TO Nscans%
  PRINT "Scan: "; scan
  tmptemperature! = (temp1%(scan * 1 + x)) / 10
  PRINT tmptemperature!; "   "
  NEXT scan
' Close DaqBook/100 and end program

FUNCTION IntToUint (IntVal AS INTEGER)
 'Converts 16-bit signed integer to unsigned integer.
 IF 0 <= IntVal THEN
   IntToUint = IntVal      'Return positive values with no change
 ELSE
   IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                            'positive
 END IF
END FUNCTION
```

Using DBK Card Calibration Files



Software calibration functions are designed to adjust Daq* readings to compensate for gain and offset errors. Calibration constants are calculated at the factory by measuring the gain and offset errors of a card at each programmable gain setting. These constants are stored in a calibration text file which can be read by a program at runtime. This allows new boards to be configured for calibration by updating this calibration file rather than recompiling the program. Calibration constants and instructions are shipped with the related DBK boards. Programs like DaqView support this calibration and use the same constants.

The calibration operation removes static gain and offset errors that are inherent in the hardware. The calibration constants are measured at the factory and do not change during the execution of a program but are different for each card and programmable-gain setting. They may even be different for each channel depending on the design of the expansion card. **Note**: the DBK19 is shipped with calibration constants. Other cards use on-board potentiometers to perform hardware calibration.

The calibration process has 3 steps:

- Initialization consists of reading the calibration file.
- Setup describes the characteristics of the data to be calibrated.
- Conversion does the actual calibration of the data.

Function prototypes, return error codes, and parameter definitions are located in the DAQBOOK.H header file for C (or similar files for other languages).

Cards that support the calibration functions are shipped with a floppy disk containing a calibration constants file. The name of the file will be the serial number of the card shipped with it. This file holds the calibration constants for each programmable-gain setting of that card. These constants should be copied to a calibration text file (DAQBOOK.CAL) located in the same directory as the program performing the calibration.

To set up the calibration file, perform the following steps:

1. Locate the floppy disk containing the calibration constants file.
2. Configure the card according to the hardware configuration section of the DBK chapter.
3. Edit the calibration file, DAQBOOK.CAL, using a text editor.
4. Add the card number information within brackets, as listed in the calibration file.
5. Add the calibration constants immediately after the card number. (These should be entered in the order given in the calibration file.)
6. Repeat steps 4 and 5 for each card.
7. Verify that no two cards are configured with the same card/channel number.

```
[MAIN]
32760,32769
32801,32750
32740,32777
32810,32768

[EXP3]
32780,32779
32800,32756
32768,32780
32750,32742

[EXP5]
32752,32764
32783,32757
32749,32767
32777,32730
```

The table shows an example of a calibration file for configuring the main Daq* unit and two DBK19 cards connected to Daq* expansion channels 3 and 5.

The initialization function for reading-in the calibration constants from the calibration text file is daqReadCalFile. The C language version of daqReadCalFile is similar to other languages and works as follows:

The filename with optional path information of the calibration file. If calfile is NULL or empty (“”), the default calibration file DAQBOOK.CAL will be read. This function is usually called once at the beginning of a program and will read all the calibration constants from the specified file. If calibration constants for a specific channel

number and gain setting are not contained in the file, ideal calibration constants will be used (essentially not calibrating that channel). If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and a non-zero error code will be returned by the daqReadCalFile function.

Once the calibration constants have been read from the cal file, they can be used by the daqCalSetup and daqCalConvert functions. The daqCalSetup function will configure the order and type of data to be calibrated. This function requires data to be from consecutive channels configured for the same gain, polarity, and channel type. The calibration can be configured to use only the gain calibration constant and not the offset constant. This allows the offset to be removed at runtime using the zero compensation functions described later in this section.

In this example, several Daq* channels will be read and calibrated. This example assumes the calibration file has been created according to the initializing calibration constants section of this chapter. Expansion cards could perform the same type of calibration if the calibration constants are available for the card and a specified channel number. First list the configuration:

Channel	Channel Type
0	Voltage1 @ X1 gain
1	Voltage2 @ X2 gain
2	Voltage3 @ X2 gain
3	Voltage4 @ X2 gain

Now specify the scan (the sequence of channel numbers and gains that are to be gathered as one burst of readings). In this example, all the channels at each gain will be read together in consecutive order to make the calibration easier.

Scan Position	Channel Type	Channel	Gain Code
0	Voltage1 @ X1 gain	0	DgainX1
1	Voltage2 @ X2 gain	1	DgainX2
2	Voltage3 @ X2 gain	2	DgainX2
3	Voltage4 @ X2 gain	3	DgainX2

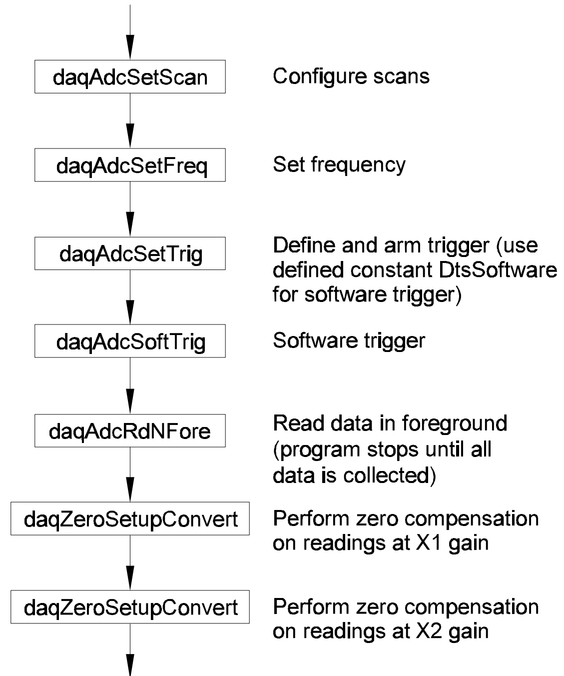
Now configure the Daq* with this information, and read 5 scans of data:

```
DIM chans%(4), gains%(4), buf%(20)

ret% = QBdaqInit%(LPT1%, 7)

' Set array of channels and gains
chans%(0) = 0
gains%(0) = DgainX1%
chans%(1) = 1
gains%(1) = DgainX2%
chans%(2) = 2
gains%(2) = DgainX2%
chans%(3) = 3
gains%(3) = DgainX2%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 4)

' Set Clock
ret% = QBdaqAdcFreq%(10)

' Define and arm trigger :
ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)

' Trigger
ret% = QBdaqAdcSoftTrig%

' Read the data
ret% = QBdaqAdcRdNFore%(buf%(), 5)

' Print the first scan of unconverted data
PRINT "Before Calibration:"
PRINT "Channel 0 at x1 gain: "; buf%(0)
PRINT "Channel 1 at x2 gain: "; buf%(1)
PRINT "Channel 2 at x2 gain: "; buf%(2)
PRINT "Channel 3 at x2 gain: "; buf%(3)

'Perform zero compensation on readings sampled at x1 gain
```

```
ret% = QBdaqCalSetupConvert%(4, 0, 1, 0, DgainX1%, 0, 1, 0, buf%(), 5)

'Perform zero compensation on readings sampled at x2 gain
ret% = QBdaqCalSetupConvert%(4, 1, 3, 0, DgainX2%, 1, 1, 0, buf%(), 5)

' Print the first scan of converted data
PRINT "After Calibration:"
PRINT "Channel 0 at x1 gain: "; buf%(0)
PRINT "Channel 1 at x2 gain: "; buf%(1)
PRINT "Channel 2 at x2 gain: "; buf%(2)
PRINT "Channel 3 at x2 gain: "; buf%(3)

FUNCTION IntToUint (IntVal AS INTEGER)
 'Converts 16-bit signed integer to unsigned integer.
 IF 0 <= IntVal THEN
   IntToUint = IntVal   'Return positive values with no change
 ELSE
   IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                            'positive
 END IF
END FUNCTION
```

Zero Compensation

Zero compensation removes offset errors while a program is running. This is useful in systems where the offset of a channel may change due to temperature changes, long-term drift, or hardware calibration changes. Reading a shorted channel on the same card at the same gain as the desired channel removes the offset at run-time.



Note: Zero compensation is not available for all expansion cards. The DBK19 has channel 1 permanently shorted for zero compensation; other cards require a channel to be shorted manually.

The zero-compensation functions require a shorted channel and a number of other channels to be sampled from the same card at the same gain as the shorted channel. These functions will work with cards that have one analog path from the input to the A/D converter such as the DBK12, DBK13, and DBK19. Other cards do not support the zero compensation functions because they have offset errors unique to each channel. The DBK19 is designed with channel 1 already shorted for performing zero compensation.

The daqZeroSetup function configures the location of the shorted channel and the channels to be zeroed within a scan, the size of the scan, and the number of readings to zero compensate. (This function does not do the conversion.) A non-zero return value indicates an invalid parameter error.

In this example, several Daq* channels will be read using various gains and zero-compensated to remove any offset errors. This example assumes that channel 0 of the Daq* has been manually shorted. Expansion cards could perform the same type of zero compensation as this example by shorting a channel on the expansion card and specifying card channel numbers. First list the configuration:

Channel	Channel Type
0	Shorted Channel
1	Voltage1 @ X1 gain
2	Voltage2 @ X2 gain
3	Voltage3 @ X2 gain
4	Voltage4 @ X2 gain

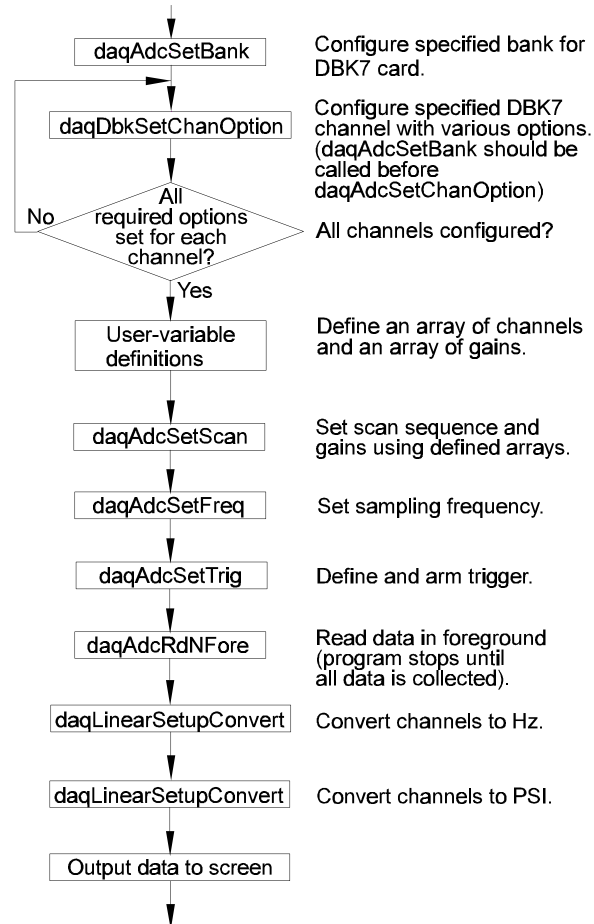
Now specify the scan, the sequence of channel numbers, and gains that are to be gathered as one burst of readings. In this example, we will first read the shorted channel at each gain that we plan on using, in this case ×1 and ×2. All the channels at each gain will be read together to make the actual zero compensation easier.

Scan Position	Channel Type	Channel	Gain Code
0	Shorted Channel @ X1	0	DgainX1
1	Shorted Channel @ X2	0	DgainX2
2	Voltage1 @ X1 gain	1	DgainX1
3	Voltage2 @ X2 gain	2	DgainX2
4	Voltage3 @ X2 gain	3	DgainX2
5	Voltage4 @ X2 gain	4	DgainX2

```
DIM chans%(6), gains%(6), buf%(30)

ret% = QBdaqInit%(LPT1%, 7)
' Set array of channels and gains
chans%(0) = 0
gains%(0) = DgainX1%
chans%(1) = 0
gains%(1) = DgainX2%
chans%(2) = 1
gains%(2) = DgainX1%
chans%(3) = 2
gains%(3) = DgainX2%
chans%(4) = 3
gains%(4) = DgainX2%
```

```
chans%(5) = 4
gains%(5) = DgainX2%

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 6)

' Set Clock
ret% = QBdaqAdcFreq%(10)

' Define and arm trigger :
ret% = QBdaqAdcSetTrig%(DtsSoftware%, 0, 0, 0, 0)

' Trigger
ret% = QBdaqAdcSoftTrig%

' Read the data
ret% = QBdaqAdcRdNFore%(buf%(), 5)

' Print the first scan of unconverted data
PRINT "Before Zero Compensation:"
PRINT "Channel 1 at x1 gain: "; buf%(2)
PRINT "Channel 2 at x2 gain: "; buf%(3)
PRINT "Channel 3 at x2 gain: "; buf%(4)
PRINT "Channel 4 at x2 gain: "; buf%(5)

'Perform zero compensation on readings sampled at x1 gain
ret% = QBdaqZeroSetupConvert%(5, 0, 2, 1, buf%(), 5)

'Perform zero compensation on readings sampled at x2 gain
ret% = QBdaqZeroSetupConvert%(5, 1, 3, 3, buf%(), 5)

' Print the first scan of converted data
PRINT "After Calibration:"
PRINT "Channel 0 at x1 gain: "; buf%(2)
PRINT "Channel 1 at x2 gain: "; buf%(3)
PRINT "Channel 2 at x2 gain: "; buf%(4)
PRINT "Channel 3 at x2 gain: "; buf%(5)

FUNCTION IntToUint (IntVal AS INTEGER)
'Converts 16-bit signed integer to unsigned integer.
 IF 0 <= IntVal THEN
   IntToUint = IntVal    'Return positive values with no change
 ELSE
   IntToUint = 65535 + CLNG(IntVal) + 1      'Convert negative values to
                                            'positive
 END IF
END FUNCTION
```

Linear Conversion



Several DBKs use conversions from A/D readings to corresponding values that are a linear (straight-line) relationship. (Non-linear relationships for RTDs and thermocouples require special conversion functions—refer to the *Thermocouple and RTD Linearization* section later in this chapter.) The linear conversion functions are built into the API.

Six parameters are used to specify a linear relationship: the A/D input range (minimum and maximum values), and the transducer input signal level and output voltage at two points in the range.

Three functions are used to perform linear conversions: **daqLinearSetup, daqLinearConvert,** and **daqLinearSetupConvert**. These functions are defined in the following pages. After their definitions, parameter examples and a program example show how they work.

DBK7 programmed for 50 to 60 Hz:

The DBK7 output range is from -5 V to +5 V, and the Daq* must be configured for bipolar operation at a gain of ×1 for the DBK7 channels. Thus, the input range -5 V to +5 V corresponds to the **ADmin** and **ADmax** settings. When a DBK7 programmed for a 50 to 60 Hz range measures a 50 Hz input signal, it outputs -5 V. With a 60 Hz input signal, it outputs +5 V. Thus, **signal1** is 50, **voltage1** is -5, **signal2** is 60, and **voltage2** is 5.

Measurement	Signal	Voltage
1	50 Hz	-5 V
2	60 Hz	+5 V

Pressure-transducer:

Assume that a pressure transducer outputs 1 to 4 mV to represent 0 to 1000 psi, and that a DBK13 with a gain of ×1000 is used with a Daq in bipolar mode to measure the signal. In bipolar mode, at a gain of 1000, the analog signal input range is 0 to 5 mV. Thus **ADmin** should be set to 0.000, and **ADmax** should be set to 0.005. A pressure of 0 psi generates an output of 1 mV, and 1000 psi generates 4 mV. Thus **signal1** is 0, **voltage1** is 0.001, **signal2** is 1000 and **voltage2** is 0.004.

Measurement	Signal	Voltage
1	0 psi	1 mV
2	1000 psi	4 mV

This program uses the linear conversion functions to convert voltage readings from a DBK7 frequency-to-voltage card and a DBK13 voltage input card with a pressure transducer to actual frequencies (Hz) and pressures (psi).

```
DIM buffer1%(80), buf%(80), chans%(3), gains%(3), hz!(20), psi!(10)

'Set Channel 16 to be a DBK7, this will configure and auto calibrate all
'channels on the DBK7 which includes channels 16,17,18 and 19. (4 ch.
 card)
'This step not required by DBK13.
ret% = QBdaqAdcSetBank%(16, bankDBK7)

'Set channel option common to both channels.
'This step not required by DBK13.
FOR chan% = 16 TO 19
  ret% = QBdaqDbkSetChanOption%(chan%, DcotSlope, 1)
```

```
   ret% = QBdaqDbkSetChanOption%(chan%, DcotDebounceTime,
  DcovDebounceNone)
   ret% = QBdaqDbkSetChanOption%(chan%, DcotMinFreq, 50!)
   ret% = QBdaqDbkSetChanOption%(chan%, DcotMaxFreq, 60!)
NEXT chan%

'Configure the scan sequence
chans%(0) = 16          'DBK7 channel 0
gains%(0) = DBK7X1      'X1 gain
chans%(1) = 17          'DBK7 channel 1
gains%(1) = DBK7X1      'X1 gain
chans%(2) = 32          'DBK7 channel 0
gains%(2) = DBK13X2000  'X1000 gain

' Load scan sequence FIFO :
ret% = QBdaqAdcSetScan%(chans%(), gains%(), 3)

' Set Clock : 1 Hz  - xtal set to 1MHz
ret% = QBdaqAdcSetFreq%(1000)

' Define and arm trigger :
ret% = QBdaqAdcSetTrig%(DtsPacerClock%, 0, 0, 0, 0)

' Read data in the foreground
ret% = QBdaqAdcRdNFore%(buffer1%(), 10)

'convert channels 16 and 17 to Hz where -5 volts corresponds to 50 Hz
'and 5 volts corresponds to 60 Hz
ret% = QBdaqLinearSetupConvert(3, 0, 2, -5!, 5, 50!, -5!, 60!, 5!, 1,
  buf%(), 10, hz!(), 20)

'convert channel 32 to PSI where 1 mV corresponds to 0 PSI
'and 4 millivolts corresponds to 1000 PSI
ret% = QBdaqLinearSetupConvert(3, 2, 1, -.005, .005, 0, .001, 1000!,
  .004, 1, buf%(), 10, psi!(), 10)

'Print results
PRINT "Results:"
FOR x = 0 TO 9
  PRINT "Scan "; x; "  "; hz!(x * 2); " Hz  "; hz!((x * 2) + 1); " Hz  ";
  psi!(x); " psi"
NEXT x

FUNCTION IntToUint (IntVal AS INTEGER)
'Converts 16-bit signed integer to unsigned integer.
  IF 0 <= IntVal THEN
    IntToUint = IntVal  'Return positive values with no change
  ELSE
    IntToUint = 65535 + CLNG(IntVal) + 1
                        'Convert negative values to 'positive
  END IF
END FUNCTION
```

Summary Guide of Selected Standard API Functions

The following table organizes the standard API functions by type including notes on when to use them.

Simple Foreground Routines

For single gain, consecutive channel, foreground transfers, use the following functions:

Foreground Operation	Single Scan	Multiple Scans
Single Channel	`daqAdcRd`	`daqAdcRdN`
Consecutive Multiple Channels	`daqAdcRdScan`	`daqAdcRdScanN`

Complex Scan Routines

For non-consecutive channels, high-speed digital channels, multiple gain settings, or multiple polarity settings, use the SetScan functions.

Function	Description
`daqAdcSetScan`	Set scan sequence using arrays of channel and gain values.
`daq200SetScan`	Set scan sequence using arrays of channel, gain, and polarity values.

Trigger Options

After the scan is set, the trigger needs to be set. The two triggering modes are one-shot or continuous.

- In one-shot mode, a trigger is required to start each A/D scan.
- In continuous mode, a single trigger starts the scans and the pacer clock determines the rate between scans.

Note: If the trigger source is analog, the trigger level is also required.

Function	Description
`daqAdcSetTrig`	Set the trigger using source, one-shot, and level parameters. Also sets the pacer clock gate source and Counter 0 clock source.
`daqAdcCalcTrig`	Using the selected trigger voltage, trigger direction, channel gain, and reference voltage, return the analog trigger source and value.

If a software trigger is selected, the start time of the scan depends on the application calling daAdcSoftTrig.

Multiple Scan Timing

If the acquisition is to have multiple scans and the trigger mode is one-shot, the pacer clock needs to be set with one of the following functions:

Function	Description
`daqAdcSetClk`	Set the pacer clock with the given frequency scalers.
`daqAdcSetFreq`	Set the pacer clock to the given frequency.

Data Transfer

After the acquisition is started, the data needs to be transferred to the application buffer. Three routines are used:

Function	Description
`daqAdcRdFore`	Read single sample from the A/D FIFO.
`daqAdcRdNFore`	Read multiple scans from the A/D FIFO.
`daqAdcRdNBack`	Inform acquisition routine where to store multiple scans. Also indicates whether the buffer should be recycled when it is full.

To find out whether a background A/D transfer is complete or to stop transfers, use the following functions:

Function	Description
`daqAdcGetBackStat`	Return whether the background transfers are in progress and the number of valid scans in the buffer.
`daqAdcStopBack`	Stop the background A/D transfers.

D/A Conversions

The 2 D/A outputs are multiplying DACs. The voltage output is a fraction of the voltage reference. This fraction is the digital value sent to the DAC divided by 4096. Using the internal -5 V reference, any voltage between 0 and 4.9988 V can be set. Two routines are used to set the D/A outputs:

Function	Description
`daqDacWt`	Set a single DAC.
`daqDacWtBoth`	Set both DACs.

DAC1 is also set by any A/D routine which uses analog triggering. This DAC is used to set the comparison level.

Digital Functions

Several routines read and write the digital inputs and outputs. The first routine to call is the configure routine:

Function	Description
`daqDigGetConf`	Using the 4 port input/output direction selections, return a configuration byte.
`daqDigConf`	Set the input/output configuration of a local or expansion port group.

After the digital group is configured, the ports can be read or written a byte at a time. (Port C low/high and P1 digital I/O are accessed a nibble at a time.) A single bit of a digital channel can be read or written using the following routines:

Function	Description
`daqDigRdBit`	Return indicated bit from selected channel.
`daqDigWrBit`	Send indicated bit to selected channel.

Counter Functions

Three counter/timer elements are in a DaqBook/112, and 9 counter/timer elements are in a DaqBook/100/200. Two counters are the ADC pacer clock. The FOUT counter element is a simple square-wave generator. Counter 0 is capable of more complex waveform and counter operations. Counters 1 through 5 are full-fledged counter/timer elements with many operating modes.

Counter 0 Functions

Counter 0 is a binary/BCD down-counter capable of 5 modes: event counter, pulse generator, rate generator, square-wave generator, software and hardware triggered strobes. Use the following commands with Counter 0:

Function	Description
`daqAdcConfCntr0`	Set up counter 0 in the indicated mode.
`daqAdcRdCntr0`	Read the contents of counter 0.
`daqAdcWtCntr0`	Set counter 0 countdown register.
`daqAdcSetTrig`	Sets Counter 0 clock source. Also sets the A/D scan trigger source and pacer clock gate source.

Counter 1 - Counter 5 Functions - For the DaqBook/100/200 Only

Counters 1 through 5 are binary/BCD, up/down 16-bit counters that can be internally cascaded. Each counter is capable of 24 modes including: hardware and software triggered strobes, rate generator, retriggerable and non-retriggerable one-shots, software- and hardware-triggered delayed one-shots, variable duty-cycle rate generator, rate generator with sync, frequency-shift keying, and hardware save. Most modes can be gated. Counters 1 and 2 can be set up as a time-of-day counter, with 100 Hz resolution. Counters 1 and 2 are also capable of alarm outputs. In the alarm mode, whenever the counter value equals the alarm value, the counter output is set. This can be used with the time-of-day mode to cause an alarm at a particular time of day. To use counters 1 through 5 or the FOUT square-wave generator, the master mode register must be set:

Function	Description
`daqCtrSetMasterMode`	Set FOUT source and scaler. Also set the counters 1 and 2 alarm mode and time-of-day mode.
`daqCtrSetAlarm`	Set the alarm comparison value for counter 1 or 2.

High-Level Counter Functions

The high-level counter functions simplify programming of a given counter task by eliminating many complex options available through low-level functions. After setting the Master Mode, counters 1 through 5 can be programmed using the following command:

Function	Description
`daqCtrRdFreq`	Read the frequency on a SRC or GATE line. This command uses counter 4 as a gate source and counter 5 for counting.

Low-Level Counter Functions

The low-level counter functions allow custom-programming of the counters. After setting the Master Mode, counters 1 through 5 can be programmed using the following commands:

Function	Description
`daqCtrSetCtrMode`	Set counter to given mode.
`daqCtrSetLoad`	Set counter load register.
`daqCtrSetHold`	Set counter hold register.

To read back a given counter, use one or both of:

Function	Description
`daqCtrMultCtrl`	Issue a command to the indicated counters. To read the current contents of a counter, issue the `DmccSave` command, and read the hold register.
`daqCtrGetHold`	Read a given hold register.

Counter Interrupt Save And Transfer

When an application program needs to read counters based on an external interrupt, the `daqCtrRd` functions are used. Whenever the interrupt on P3 is asserted, the programmed counters are saved in an application buffer.

Function	Description
`daqCtrRdNFore`	Read counters on interrupt and transfer data in foreground.
`daqCtrRdNBack`	Read counters on interrupt and transfer data in background.
`daqCtrGetBackStat`	Return whether the background transfers are in progress and the number of valid scans in the buffer.
`daqCtrStopBack`	Stop the background counter saves and transfers.

Notes

Daq* Command Reference (Standard API) 5

Overview

The first part of the chapter describes the Daq* driver commands for DOS, Windows 3.1, Windows for Workgroups (not NT), and the 16-bit mode of Windows95 (this is the **standard** API and is not to be confused with the **enhanced** API). The first table lists the commands by their function types and provides a page index. Included at the end of the chapter are several tables that define A/D Channel Descriptions, Thermocouple Types, A/D Trigger Software Definitions, A/D Gain Definitions, Digital I/O Port Connection, and the API Error Codes.

Function	Description	Page
High and Low-Level A/D Functions-		
`daqAdcCalcTrig`	Calculate the trigger level and trigger source for an analog trigger	5-3
`daqAdcConfCntr0`	Configure the counter 0 mode	5-4
`daqAdcConvertTagged`	Convert array that contains channel tags into two separate arrays	5-5
`daqAdcExpToChan`	Calculate a channel number that can be used with all other A/D functions	5-5
`daqAdcGetBackStat`	Read the status of a background A/D transfer	5-6
`daqAdcGetFreq`	Read the current pacer clock frequency	5-6
`daqAdcGetScan`	Read the current scan configuration	5-7
`daqAdcRd`	Configure an A/D acquisition and read one sample from a channel	5-7
`daqAdcRdCntr0`	Read the current value of the counter 0	5-8
`daqAdcRdFore`	Read a single A/D sample and increment the channel mux	5-8
`daqAdcRdN`	Configure an A/D acquisition and read multiple scans from a channel	5-9
`daqAdcRdNBack`	Read count A/D scans in the background using interrupts	5-10
`daqAdcRdNBackPreT`	Read multiple A/D scans, initiated by `daqAdcSetrigPreT`, in the background	5-11
`daqAdcRdNFore`	Read count A/D samples in the foreground (polled mode)	5-12
`daqAdcRdNForePreT`	Read multiple A/D scans, initiated by `daqAdcSetTrigPretT`, in the foreground	5-12
`daqAdcRdNForePreTWait`	Read multiple A/D scans, initiated by `daqAdcSetTrigPretT`, in the foreground without returning until the acquisition completes	5-13
`daqAdcRdScan`	Configure an A/D acquisition and read one scan	5-14
`daqAdcRdScanN`	Configure an A/D acquisition and read multiple scans	5-14
`daqAdcSetBank`	Sets which channels are DBK50 to allow programming of gains prior to the acquisition.	5-15
`daqAdcSetClk`	Set the pacer clock counters	5-15
`daqAdcSetFreq`	Configure the pacer clock frequency in Hz	5-16
`daqAdcSetMux`	Configure a scan specifying start and end channels	5-16
`daqAdcSetScan`	Configure up to 256 channels making up an A/D or HS digital input scan	5-17
`daqAdcSetTag`	Configure whether A/D data contains channel tags	5-17
`daqAdcSetTrig`	Configure an A/D trigger	5-18
`daqAdcSetTrigPreT`	Set the trigger of analog level triggering & initiates the collection of pre-trigger data acquisition	5-19
`daqAdcSoftTrig`	Save a software trigger command to the DaqBook/DaqBoard	5-20
`daqAdcStopBack`	Stop a background A/D transfer	5-20
`daqAdcWtCntr0`	Write a value to counter 0	5-21
Counter/Timer Functions		
`daqCtrGetBackStat`	Read the status of a background counter transfer	5-31
`daqCtrGetHold`	Read the hold register of the specified counter	5-31
`daqCtrMultCtrl`	Simultaneously configure multiple counters	5-32
`daqCtrRdFreq`	Read up to 9 frequency inputs	5-33
`daqCtrRdNBack`	Read count values from up to 5 counters using interrupts	5-34
`daqCtrRdNFore`	Read count values from up to 5 counters in the foreground	5-35
`daqCtrSetAlarm`	Set the specified alarm register	5-35
`daqCtrSetCtrMode`	Set the 9513's mode register for the specified counter	5-36
`daqCtrSetHold`	Output a value to the counter hold register	5-39
`daqCtrSetLoad`	Output a value to the counter load register	5-39
`daqCtrSetMasterMode`	Initialize various counter/timer values	5-40
`daqCtrStopBack`	Stop a background counter transfer	5-42
D/A Functions		
`daqDacWt`	Output a D/A value	5-42
`daqDacWtBoth`	Output D/A values to both DACs	5-43
`daqDacWtMany`	Output D/A values to several DACs	5-43

Digital I/O Functions		
daqDigConf	Configure the mode of the 8255 digital I/O ports	5-44
daqDigGetConf	Execute an interrupt handler on an external digital I/O interrupts	5-45
daqDigRdBit	a bit on a digital input port	5-45
daqDigRdByte	Read a byte from a digital input port	5-46
daqDigWtBit	Program a bit on a digital output port	5-46
daqDigWtByte	Output a byte to a digital output port	5-47
Thermocouple Functions		
daqRtdConvert	Converts raw A/D readings from RTDs to temperature readings	5-51
daqRtdSetup	Set up parameters for subsequent RTD temperature conversions	5-52
daqRtdSetupConvert	Set up and convert raw A/D readings from RTDs into temperature readings	5-53
daqTCConvert	Convert raw A/D readings from thermocouples to temperature readings	5-57
daqTCSetup	Set up parameters for subsequent thermocouple temperature conversions	5-59
daqTCSetupConvert	Set up and convert raw A/D readings from thermocouples into temperature readings	5-60
DaqBook/200 Functions		
daq200GetScan	Retrieves a scan sequence, similar to **daqAdcGetScan**	5-63
daq200SetMode	Program the gain amp and set the default polarity	5-63
daq200SetScan	Configure a scan sequence with polarity mode per channel	5-64
DaqBoard Functions		
daqBrdAdcSetTimeBase	Set the timebase for the ADC pacer clock	5-21
daqBrdDacClockSrc	Select the source for DAC FIFO pacer clock	5-22
daqBrdDacCtrl	Set the mode of the DAC FIFO	5-23
daqBrdDacPredefWave	Build a waveform and assign to a DAC channel	5-24
daqBrdDacRestFIFO	Reset DAC FIFO and its pointers	5-25
daqBrdDacSetMode	Set the mode of DAC FIFO, the cycling mode of FIFO and update rate per sample	5-25
daqBrdDacSetTimeBase	Set the time base for the DAC FIFO pacer clock	5-26
daqBrdDacStart	Start the waveforms	5-26
daqBrdDacStop	Stop the waveforms	5-27
daqBrdDacUserWave	Assign a user defined waveform to a DAC channel	5-27
daqBrdDacWriteFIFO	Load sample data directly into DAC FIFO	5-28
daqBrdSetDmaMode	Set the direction for DMA transfers	5-28
Software Calibration and Zero Compensation Functions		
daqCalConvert	Perform the actual calibration of one or more scans	5-29
daqCalSetup	Configure the order and type of data to be calibrated	5-29
daqCalSetupConvert	Perform both the setup and convert steps with one call	5-30
daqReadCalFile	Read all the calibration constants from the specified file	5-50
daqZeroConvert	Perform zero compensation on one or more scans	5-61
daqZeroDbk19	Configure thermocouple linearization functions for automatic zero compensation	5-61
daqZeroSetup	Configure data for zero compensation	5-62
daqZeroSetupConvert	Perform both the setup and convert steps with one call	5-62
Linear Conversion Functions		
daqLinearConvert	Convert ADC readings into floating point numbers	5-49
daqLinearSetup	Save data required for **daqLinearConvert**	5-49
daqLinearSetupConvert	Combine setup and conversion into one function	5-50
General Functions		
daqClose	End communication with the DaqBook/DaqBoard	5-30
daqGetProtocol	Return the current parallel port communications protocol	5-47
daqInit	Initialize a single DaqBook/DaqBoard	5-48
daqSelectPort	Select an initialized DaqBook/DaqBoard as the current DaqBook/DaqBoard	5-54
daqSetErrHandler	Specify a user-defined routine to call when an error occurs in any command	5-55
daqSetProtocol	Specify the type of parallel-port implementation and protocol available on the computer	5-56
daqVersion	Return the hardware version	5-61

Commands in Alphabetical Order

The following pages give the details for each API command. Listed in alphabetical order, each section starts with a table that summarizes the main features of the command (language prototypes for C, QuickBASIC, and Turbo Pascal, and the related parameters). An explanation follows with related information and in some cases a programming example. **Typographic note**: Commands, parameters, values, and code use a bold, mono-spaced **Courier** font to distinguish characters that can be ambiguous in other fonts.

daqAdcCalcTrig

DLL Function	`daqAdcCalcTrig(uchar bipolar, uint gain, float vset, uchar rising, float vref, uint *level, uchar *source);`
C	`daqAdcCalcTrig(uchar bipolar, uint gain, float vset, uchar rising, float vref, uint far *level, uchar far *source)`
QuickBASIC	`QBdaqAdcCalcTrig%(bipolar%, gain%, vset!, rising%, vref!, level%, source%)`
Turbo Pascal	`daqAdcCalcTrig(bipolar:byte; gain:word; vset:real; rising:byte; vref:real; level:WordP);`
Parameters	
`uchar bipolar`	A flag that should be non-zero if the trigger channel is bipolar, or zero if it is unipolar
`uint gain`	A gain value of the trigger channel
`float vset`	The analog trigger setpoint
`uchar rising`	A flag that if non-zero will calculate a rising analog trigger, otherwise calculates a falling analog trigger
`float vref`	The external reference voltage of D/A channel 1 Valid values: 0 >vref > -10
`uint _far *level`	The trigger level to be passed to the `daqAdcSetTrig` command
`uchar _far *source`	The trigger source to be passed to the `daqAdcSetTrig` command
Returns	`DerrInvDacLevel` - vset or vref out of valid range `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTrig, daqAdcSetMux, daqAdcSetScan, daq200SetScan`
Program References	None
Used With	

`daqAdcCalcTrig` calculates the trigger level and source for an analog trigger. The level and source parameters can be passed to the **`daqAdcSetTrig`** function to configure the analog trigger.

The source and trigger parameters are calculated from the unipolar/bipolar and gain settings of the trigger channel, the desired analog voltage setpoint and trigger polarity, and the external reference voltage of D/A channel 1. The trigger channel is the first channel in the current A/D scan group.

The **`bipolar`** parameter should be set according to the current bipolar/unipolar setting of the trigger channel. This parameter is jumper selectable when using a DaqBook/100/112 and DaqBoard/100A/112A and software-programmable when using the DaqBook/200/200A.

The **`gain`** value sent to the **`daqAdcCalcTrig`** should be the actual gain of the trigger channel, not the gain definition used by the rest of the DaqBook/DaqBoard A/D functions. For example, if the trigger channel uses the gain definition **`DgainX8`**, the gain parameter of **`daqAdcCalcTrig`** should be 8.

The **`vset`** and **`rising`** parameters define the analog voltage at which the Daq* will trigger and whether the analog signal must be rising or falling through this setpoint. The setpoint must be within the valid input range of the trigger channel. For example, the setpoint range for a bipolar channel with unity gain would be 0 to 10 V (for ×8 gain, the range would be 0 to 1.25 V) for a DaqBook or a DaqBoard.

The **`vref`** parameter is the external reference voltage of D/A channel 1. This reference must be negative for analog triggers to work. The value -5 should be passed if the internal reference is used.

When using the Daq PCMCIA, the external reference (**`float vref`**) and polarity (**`uchar bipolar`**) parameters are ignored.

daqAdcConfCntr0

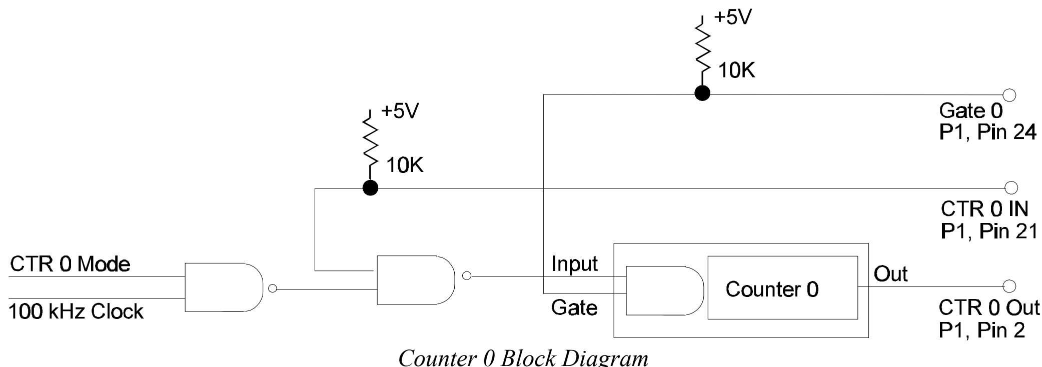
DLL Function	`daqAdcConfCntr0(uchar config);`	
C	`daqAdcConfCntr0(int conf)`	
QuickBASIC	`QBdaqAdcConfCntr0%(config%)`	
Turbo Pascal	`daqAdcConfCntr0(config:byte) :integer;`	
Parameters		
`uchar config`	The configuration of Counter 0 (See table below for definitions.)	
Counter 0 Configuration Definitions		
Description	**Value**	**Note**
`Dc0cHighTermCnt`	`30h`	High on terminal count
`Dc0cOneShot`	`32h`	Hardware retriggerable one-shot
`Dc0cRateGen`	`34h`	Rate Generator
`Dc0cSquareWave`	`36h`	Square wave
`Dc0cSoftTrigStrobe`	`38h`	Software triggered strobe
`Dc0cHardTrigStrobe`	`3Ah`	Hardware triggered strobe
Returns	`DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqAdcWtCntr0, daqAdcRdCntr0, daqAdcSetTrig`	
Program References	ADC5 (All Languages)	
Used With	**Does not apply to DaqPCMCIA models**	

`daqAdcConfCntr0` programs the control register of Counter 0 in 1 of 6 modes. Counter 0 is a general purpose counter with input, gate and output lines. The input of counter 0 can be configured using the **`ctr0mode`** parameters of the **`daqAdcSetTrig`** command.

The 6 modes are:

- **0**, high on terminal count, is typically used to count events. After the initial count value (see **`daqAdcWtCntr0`**) is set, the counter will decrement on each pulse of the Counter 0 input (pin 21 of P1). The count value at any time can be read using **`daqAdcRdCntr0`**. Counter 0 output (pin 2 of P1), which is initially low, will go high when the counter decrements to 0.
- **1**, hardware retriggerable 1-shot, is used to generate a pulse following the occurrence of a rising edge of the Counter 0 gate (pin 24 of P1). The output, which is initially high, will go low after the hardware trigger is received until the count decrements to 0.
- **2**, rate generator, a divide-by-N counter. The output will be high until the counter value decrements to 1, when the output goes low for 1 clock pulse before going high again.
- **3**, square wave generator, is similar to mode 2 except for the duty-cycle. The output will be high for half of the count value, and low for the other half. If the count value is odd, the output will remain high for the extra clock pulse.
- **4**, software triggered strobe, will strobe each time the count value is loaded. The output is initially high. After the count value is written and has decremented to 1, the output will go low for one clock pulse before going high again.
- **5**, hardware triggered strobe, is similar to mode 4 except the strobe is initiated by a hardware trigger (rising edge of Counter 0 gate).

Note: Using counter 0 requires the JP2 jumpers to be in the **-OCTOUT** and **-OCLKIN** positions.



Counter 0 Block Diagram

daqAdcConvertTagged

DLL Function	`daqAdcConvertTagged(uint *taggedData, uint *buf, uchar *tags, uint count);`
C	`daqAdcConvertTagged(int *taggedData, int *buf, int *tags, int count)`
QuickBASIC	`QBdaqAdcConvertTagged%(taggedData%(), buf%(), tags%(), count%`
Turbo Pascal	`daqAdcConvertTagged (taggedData:DataP; buf:DataP; tags:ByteP; count:word ) :integer;`
Parameters	
`uint *taggedData`	An array containing the raw tagged A/D scans
`uint *buf`	An array where the A/D scans will be returned or (`unit` *) 0 if the A/D data is not desired. Valid values: 0 - 4095
`uchar*tags`	An array where the channel tags will be returned or (`uchar` *) 0 if the channel tags are not desired. Valid values: 0 - 15
`uint count`	The number of scans in the `taggedData` array to convert
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTag`
Program References	ADC1, ADC2, ADC4 (All Languages)
Used With	**Does not apply to Daq PCMCIA models**

`daqAdcConvertTagged` converts 16-bit channel-tagged data into an array of 4-bit channel tags and an array of 12-bit A/D scans. The A/D converter of the DaqBook/DaqBoard returns the data in a 16-bit tagged format which consists of the 12-bit A/D reading in the upper 12 bits of the data and a channel number in the lower 4 bits. This function strips the channel tags and A/D scans into separate buffers and shifts the A/D data right by 4 bits. If an expansion card is used, the jumper setting of the expansion card will be returned in the channel tag location. This function is not necessary if data was collected after disabling channel tags with **`daqAdcSetTag`**. (**`taggedData`** and **`buf`** can be the same array.)

Note: **`daqAdcConvertTagged`** should not be used on data from the high-speed digital I/O port. The data from this port will be stripped and shifted along with the rest of the A/D data.

daqAdcExpToChan

DLL Function	`daqAdcExpToChan(uint expCard, uint expChan, uint *chan);`
C	`daqAdcExpChan(int expCard, int expChan, int *chan)`
QuickBASIC	`QBdaqAdcExpChan%(expCard%, expChan%, chan%)`
Turbo Pascal	`daqAdcExpToChan(expCard:byte; expChan:byte; chan:DataP ) :integer;`
Parameters	
`uint expCard`	The expansion card number Valid values: 0 - 15
`uint expChan`	The channel number on the expansion card Valid values: 0 -15
`uint *chan`	A variable to hold the channel number
Returns	`DerrInvChan` - Invalid analog input channel `DerrNoError` - No errors (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	

`daqAdcExpToChan` is used to calculate a channel number that can be used with all other A/D functions from an expansion card number and an expansion card channel.

daqAdcGetBackStat

DLL Function	daqAdcGetBackStat(uchar *active, ulong *count);
C	daqAdcGetBackStat(int *active, int *count)
QuickBASIC	QBdaqAdcGetBackStat%(active%, count%)
Turbo Pascal	daqAdcGetBackStat(active: ByteP; count:LongP):integer;
Parameters	
uchar *active	A flag which will be returned non-zero if a background transfer is in progress, or 0 if not
ulong *count	The number of scans acquired by the last or current background transfer
Returns	DerrOverrun - Internal data buffer overrun DerrFIFOFull - ADC FIFO Overrun DerrNoError - No error (also, refer to *API Error Codes* on page 5-68)
See Also	daqAdcRdNBack, daqAdcStopBack
Program References	ADC4 (All Languages)
Used With	

daqAdcGetBackStat determines if a background operation is still in progress. It also reads the number of bytes acquired by the last or current background operation initiated by the **daqAdcRdNBack** function.

daqAdcGetBackStat can return two possible error codes:

- **DerrFIFOFull** is returned if the data FIFO in the DaqBook/DaqBoard is filled before the data can be read. The data read may be invalid.
- **DerrOverrun** is returned if the **daqAdcRdNBack** is called with the cycle flag enabled. The software is just fast enough to read one buffer of data. If this error occurs, the amount of data available (specified by **count**) is valid, but the transfer was stopped.

daqAdcGetFreq

DLL Function	daqAdcGetFreq(float *freq);
C	daqAdcGetFreq(float *freq)
QuickBASIC	QBdaqAdcGetFreq%(freq!)
Turbo Pascal	daqAdcGetFreq(freq:FloatP) :integer;
Parameters	
float *freq	A variable to hold the currently defined sampling frequency in Hz Valid values: 100000.0 - 0.0002
Returns	DerrNoError - No errors (also, refer to *API Error Codes* on page 5-68)
See Also	daqAdcSetFreq, daqAdcSetClk
Program References	None
Used With	

daqAdcGetFreq reads the sampling frequency of the pacer clock.

Note: **daqAdcGetFreq** assumes that the 1 MHz/10 MHz jumper is set to the default position of 1 MHz.

daqAdcGetScan

DLL Function	`daqAdcGetScan(uint *chans, uchar *gains, uint *count);`
C	`daqAdcGetScan(int *buf)`
QuickBASIC	`QBdaqAdcGetScan%(buf%())`
Turbo Pascal	`daqAdcGetScan(chans:DataP; gains:ByteP; count:DataP ) :integer;`
Parameters	
`uint *chans`	An array to hold up to 512 channel numbers or 0 if the channel information is not desired.
`uchar *gains`	An array to hold up to 512 gain values or 0 if the channel gain information is not desired
`uint count`	A variable to hold the number of values returned in the chans and gains arrays
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetScan, daqAdcSetMux`
Program References	None
Used With	

`daqAdcGetScan` reads the current scan sequence consisting of up to 512 channels and gains.

daqAdcGetScan

DLL Function	`daqAdcRd(uint chan, uint *sample, uchar gain);`
C	`daqAdcRd(int *sample, uchar gain )`
QuickBASIC	`QBdaqAdcRd%(chan%, sample%)`
Turbo Pascal	`daqAdcRd(chan:word; sample:DataP; gain:byte ) :integer;`
Parameters	
`uint chan`	A single channel number
`unit *sample`	A pointer to a value where an A/D sample is stored. Valid values: (See daqAdcSetTag)
`unchar gain`	The channel gain
Returns	`DerrFIFOFull` - Buffer Overrun `DerrInvGain` -Invalid gain `DerrInvChan` - Invalid channel `DerrNoError` -No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcRdN, daqAdcSetMux, daqAdcSetTrig, daqAdcSoftTrig, daqAdcRdFore`
Program References	ADC1 (All Languages)
Used With	

`daqAdcRd` is used to take a single reading from the given local A/D channel. This function will use a software trigger to immediately trigger and acquire one sample from the specified A/D channel.

daqAdcRdCntr0

DLL Function	`daqAdcRdCntr0(uint *cntr0, uchar<_>latch);`
C	`daqAdcRdCntr0(uint *cntr0)`
QuickBASIC	`QBdaqAdcRdCntr0%(cntr0%)`
Turbo Pascal	`daqAdcRdCntr0(cntr0:DataP; mode:Byte ):integer;`
Parameters	
`uint *cntr0`	The value read back from the Counter 0 hold register Valid values: 0 -65535
`uchar latch`	If latch is non-zero, the count register will be latched into the hold register before reading. If latch is zero, the count register will be read directly. Direct reading should only be performed when no clock pulses are present.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcConfCntr0, daqAdcWtCntr0`
Program References	None
Used With	**Does not apply to Daq PCMCIA models**

`daqAdcRdCntr0` reads the hold register of counter 0. This function is normally used with mode 0 of counter 0 (see **`daqAdcConfCntr0`**) to read the current count value.

Note: Using counter 0 requires that the JP1 jumpers are in the -OCTOUT and -OCKLIN positions.

daqAdcRdFore

DLL Function	`daqAdcRdFore(uint *sample);`
C	`daqAdcRdFore(int *sample)`
QuickBASIC	`QBdaqAdcRdFore%(sample%)`
Turbo Pascal	`daqAdcRdFore(sample:DataP ) :integer;`
Parameters	
`uint *sample`	A pointer to a value where an A/D sample is stored Valid values: (See `daqAdcSetTag`)
Returns	`DerrFIFOFull` - Buffer overrun `DerrNoError` -No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqReadFIFO, daqAdcSetTag, adcSetClk, daqAdcSetTrig, daqAdcSetScan`
Program References	ADC2, ADC3 (All Languages)
Used With	

`daqAdcRdFore` will read one sample from the A/D data FIFO. This function, unlike **`daqAdcRd`**, will not configure the trigger source. It assumes that the A/D converter has already been configured to acquire data.

Note: If the A/D converter has not been configured to acquire data, this function may wait indefinitely and hang the computer.

daqAdcRdN

DLL Function	`daqAdcRdN(uint chan, uint *buf, uint count, uchar trigger, uchar oneshot, uint level, float freq, uchar gain);`
C	`daqAdcRdN(int chan, int *buf, int count, int mode, int cycle, int trigger, int level, float freq, uchar gain)`
QuickBASIC	`QBdaqAdcRdN%(chan%, buf%(), count%, mode%, trigger%, oneshot%, level %, freq!, gain%)`
Turbo Pascal	`daqAdcRdN(chan:word; buf:DataP; count:word; trigger:byte;):integer;`
Parameters	
`unit chan`	A single channel number
`uint *buf`	An array where the A/D scans will be returned
`uint count`	The number of scans to be taken Valid values: 1 - 32767
`uchar trigger`	The trigger source
`uchar one shot`	A flag that if non-zero enables one-shot trigger mode, otherwise enables continuous mode.
`uint level`	The trigger level if an analog trigger is specified Valid values: 0 -4095
`float freq`	The sampling frequency in Hz (100000.0 to 0.0002)
`uchar gain`	The channel gain
Returns	`DerrFIFOFull` - Buffer overrun `DerrInvGain` -Invalid gain `DerrIncChan` - Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid level (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcRd, daqAdcRdScan, daqAdcRdNScan, daqAdcRdNFore, daqAdcSetFreq, daqAdcSetMux, daqAdcSetTag, daqAdcSetClk, daqAdcSetTrig`
Program References	ADC1(VB,C)
Used With	

daqAdcRdN is used to take multiple scans from a single A/D channel. This function will:

- Configure the pacer clock
- Arm the trigger
- Acquire **count** scans from the specified A/D channel.

daqAdcRdNBack

DLL Function	daqAdcRdNBack(uint *buf, uint count, uchar cycle, uchar update Single);	
C	daqAdcRdNBack(int *buf, int count, int cycle)	
QuickBASIC	QBdaqAdcRdNBack%(buf%(), count%, cycle%, update single %)	
Turbo Pascal	daqAdcRdNBack(buf:DataP; count:word; cycle:byte; updateSingle:byte):integer;	
Parameters		
uint *buf	An array where the A/D scans will be placed	
uint count	The number of scans to be taken Valid values: 1 - 32767	
uchar cycle	A flag that if non-zero will enable continuous operation, or if 0 will disable it	
uchar updateSingle	One of three possible modes. See table below.	
Modes		
Mode	**Value**	**Description**
DusBlock	00h	
DusSingle	01h	
DusDMA	02h	DMA transfers are valid for DaqBoard products only
Returns	DerrMultBackXfer - Background read already in progress DerrNoError - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	daqAdcGetBackStat, daqAdcStopBack, daqAdcSetTag, daqAdcSetClk, daqAdcSetTrig	
Program References	ADC4 (All Languages)	
Used With		

daqAdcRdNBack reads multiple A/D scans in the background using interrupts. This function will return control back to the user's program after initiating the background transfer. The user can then monitor the status of the background transfer with the **daqAdcGetBackStat** function or stop the transfer with the **daqAdcStopBack** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes the A/D acquisition has already been setup.

If the **cycle** flag is true, the background transfer will run continuously looping back to the beginning of **buf** after **count** scans have been read. Thus large amounts of data can be read without calling **daqAdcRdNBack** multiple times. As long as you monitor how much data is in the buffer and process the data before it gets overwritten, the background transfer can run indefinitely. In this mode, you should get the total number of scans written into **buf** using **daqAdcGetBackStat** and keep track of the total number of scans processed in a variable. The difference between these two totals is the number of unprocessed valid scans in **buf** that you can process.

The **updateSingle** flag allows you to control whether the Daq* updates **buf** 1 sample at a time or in blocks of 256 samples. Enabling **updateSingle** allows the user to read A/D data during slow acquisitions as each sample is acquired. Because the **updateSingle** flag is directly tied to the number of interrupts generated on the computer, the flag should not be enabled if the acquisition rate is greater than 500 scans per second (sampling rate times the number of channels). For example, an acquisition running at 1 Hz might enable the **updateSingle** flag so that the data can be read each second rather than waiting for 256 seconds. An acquisition running at 10,000 Hz would disable the flag so that the computer does not hang.

If you are using a product that supports DMA, you can transfer scan readings to memory using DMA by calling this function with **updateSingle** equal to **DusDMA**. This will enable DMA transfers in the direction specified by the **daqBrdSetDmaMode** command.

daqAdcRdNBackPreT

DLL Function	`daqAdcRdNBackPreT(uint *buf, uint count,  uchar cycle);`
C	`daqAdcRdNBackPreT(uint *buf, uint count, uchar cycle)`
QuickBASIC	`QBdaqAdcRdNBackPreT%(buf%(), count%, cycle%)`
Turbo Pascal	`daqAdcRdNBackPreT(buf:count:cycle):integer;`
Parameters	
`uint *buf`	An array where the A/D scans will be placed
`uint count`	The number of scans to be taken (1-32767)
`uchar cycle`	A flag that if non-zero will enable continuous operation, or if 0 will disable it.
Returns	`DerrMultBackXfer` - Background read already in progress `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcGetBackStat, daqAdcStopBack, daqAdcSetTag, daqAdcSetTrigPreT`
Program References	ADCPRET3.C
Used With	

daqAdcRdNBackPreT reads multiple A/D scans, initiated by the **daqAdcSetTrigPreT** command, in the background. This function will return control to the user's program after initiating the background transfer. The user can then monitor the status of the background transfer with the **daqAdcGetBackStat** function or stop the transfer with the **daqAdcStopBack** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes the pre-trigger acquisition has already been setup using **daqAdcSetTrigPreT**.

If the **cycle** flag is true, the background transfer will run continuously looping back to the beginning of **buf** after **count** scans have been read. Under this mode the background transfer will continue until the acquisition completes. This allows the user to collect large amounts of data without calling **daqRdNBackPreT** several times. As long as the user monitors how much data is in the buffer and processes the data before it gets overwritten, the background transfer can run until the acquisition completes. In this mode the user should get the total number of scans written into **buf** using the **daqAdcGetBackStat** function and keep track of the total number of scans processed in a variable. The difference between these two totals is the number of unprocessed valid scans in **buf** that the user can process.

If, however, the **cycle** flag is false, the background transfer will only collect the number of scans specified in **count**. If this is the case, then a number of **daqAdcRdNBackPreT** calls may be necessary to read all the data collected during the pre-trigger mode acquisition.

daqAdcRdNFore

DLL Function	`daqAdcRdNFore(uint *buf, uint count);`
C	`daqAdcRdNFore(int *buf, int count)`
QuickBASIC	`QBdaqAdcRdNFore%(buf%(), count %)`
Turbo Pascal	`daqAdcRdNFore(buf:DataP; count:word ):integer;`
Parameters	
`uint *buf`	An array where the A/D samples will be placed
`uint count`	The number of scans to be taken. Valid values: 1 -32767
Returns	`DerrFIFOFull` - Buffer overrun `DerrNoError` -No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTag, daqAdcSetClk, daqAdcSetTrig`
Program References	ADC2, ADC3 (All Languages)
Used With	

`daqAdcRdNFore` reads multiple A/D scans in the foreground. Unlike **`daqAdcRdNBack`**, this function does not use interrupts and does not return control immediately to the program. It will return only when **`count`** scans have been read. This function will not configure the A/D acquisition and assumes that the A/D converter has already been configured to acquire data.

Note: If the A/D converter has not been configured to acquire data, this function may wait indefinitely, hanging the computer.

daqAdcRdNForePreT

DLL Function	`daqAdcRdNForePreT(uint *buf, uint count, uint *retcount, uchar *active);`
C	`daqAdcRdNForePreT(uint *buf, uint count, uint *retcount, uchar *active)`
QuickBASIC	`QBdaqAdcRdNForePreT%(buf%(), count %, retcount%, active%)`
Turbo Pascal	`daqAdcRdNForePreT(buf:count:retcount:active):integer;`
Parameters	
`uint *buf`	An array where the A/D samples will be placed
`uint count`	The number of scans to be taken. Valid values: 1 -32767
`uint *retcount`	Pointer to an integer representing the number of scans actually taken
`uchar *active`	Pointer to a flag indicating whether or not the pre-trigger acquisition is still active
Returns	`DerrFIFOFull` - Buffer overrun `DerrNoError` -No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTrigPreT, daqAdcSetTag, daqAdcRdNForePreTWait, daqAdcRdNBackPreT`
Program References	ADCPRET1.C
Used With	

`daqAdcRdNForePreT` reads multiple A/D scans, initiated by the **`daqAdcSetTrigPreT`** command, in the foreground. Unlike the **`daqAdcRdNBackPreT`** command, this function does not use interrupts and does not return control immediately to the application program. It will only return when either the specified **`count`** has been satisfied or the acquisition completes. **Note**: If the A/D converter has not been configured to acquire data, this function may wait indefinitely, hanging the computer.

This function may be called subsequent to configuring a pre-trigger acquisition using the **`daqAdcSetTrigPreT`** command. Once this command has been called, it will return only when one of two possible conditions are met:

- The specified number of scans has been collected.
- The trigger has been detected and the acquisition has completed. The returned **`active`** flag will be 0, and the number of scans actually collected will be returned in **`retcount`**.

daqAdcRdNForePreTWait

DLL Function	`daqAdcRdNForePreTWait(uint *buf, uint count, uint *retcount);`
C	`daqAdcRdNForePreTWait(uint *buf, uint count, uint *retcount)`
QuickBASIC	`QBdaqAdcRdNForePreTWait% (buf%(), count %, retcount%)`
Turbo Pascal	`daqAdcRdNForePreTWait(buf:count:retcount:active):integer;`
Parameters	
`uint *buf`	An array where the A/D samples will be placed
`uint count`	The number of scans to be taken Valid values: 1 - 32767
`uint *retcount`	Pointer to an integer representing the number of scans actually taken
Returns	`DerrFIFOFull` - Buffer overrun `DerrNoError` -No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTrigPreT, daqAdcSetTag, daqAdcRdNForePreT, daqAdcRdNBackPreT`
Program References	ADCPRET2.C
Used With	

daqAdcRdNForePreTWait reads multiple A/D scans, initiated by the **daqAdcSetTrigPreT** command, in the foreground. Unlike the **daqAdcRdNForePreT** command, this function will not return until the acquisition completes. It will only return when the specified trigger event has occurred and the specified post trigger count has been satisfied.

This function may be called subsequent to configuring a pre-trigger acquisition using the **daqAdcSetTrigPreT** command. Once this command has been called, it will return only when the trigger has been detected and the acquisition has completed. The amount in the **count** parameter specifies the length of the supplied buffer in scans. Unlike **daqAdcRdNForePreT**, this command will not return when **count** is satisfied; instead it will continue acquiring by wrapping the scans to the beginning of the buffer until the final post-trigger scan is collected and the acquisition completes.

When the acquisition completes, control will be returned to the application program along with the actual number of scans collected in the **retcount** parameter.

Note: If the A/D converter has not been configured to acquire data or the trigger event never occurs, this function may wait indefinitely, hanging the computer.

daqAdcRdScan

DLL Function	`daqAdcRdScan(uint startChan, uint endChan, uint *buf, uchar gain);`
C	`daqAdcRdScan(int startChan, int endChan, int *buf)`
QuickBASIC	`QBdaqAdcRdScan%(startChan%, endChan%, buf%(), gain%)`
Turbo Pascal	`daqAdcRdScan(startChan:word; endChan:word;buf:DataP; gain:byte ):integer;`
Parameters	
`uint startChan`	The starting channel of the scan group
`uint endChan`	The ending channel of the scan group
`uint *buf`	An array where the A/D scans will be placed
`unchar gain`	The channel gain
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcRd, daqAdcRdN, daqAdcRdNScan, daqAdcSetMux, daqAdcRdFore, daqAdcSetTag, daqAdcSetClk, daqAdcSetTrig`
Program References	ADC1 (All Languages)
Used With	

`daqAdcRdScan` reads a single sample from multiple channels. This function will use a software trigger to immediately trigger and acquire 1 scan consisting of each channel starting with **`startChan`** and ending with **`endChan`**.

daqAdcRdScanN

DLL Function	`daqAdcRdScanN(uint startChan, uint endChan, uint *buf, uint count, uchar trigger, uchar one Shot, uint level, float freq, uchar gain );`
C	`daqAdcRdScanN(int startCan, int endChan, int *buf, int scans, int mode, int cycle, int trigger, float freq)`
QuickBASIC	`QBdaqAdcRdScanN%(startChan%, endChan%, buf%(), scan%, mode%, cycle%, trigger%, freq!, gain%)`
Turbo Pascal	`daqAdcRdScanN(startChan:word; endChan:word; buf:DataP; count:word; trigger:byte; oneShot:byte;level:word; freq:real; gain:byte ) :integer;`
Parameters	
`uint startChan`	The starting channel of the scan group (see table at end of chapter)
`uint endChan`	The ending channel of the scan group (see table at end of chapter)
`uint *buf`	An array where the A/D scans will be placed
`uint count`	The number of scans to be read Valid values: 1 - 65536
`uchar trigger`	The trigger source (see table at end of chapter)
`uchar one Shot`	A flag that if non-zero enables one-shot trigger mode
`uint level`	The trigger level if an analog trigger is specified Valid values: 0 -4095
`float freq`	The sampling frequency in Hz Valid values: 100000.0 - 0.0002
`uchar gain`	The channel gain (See tables at end of chapter).
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid Level `DerrFIFOFull` -Buffer Overrun `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcRd, daqAdcRdN, daqAdcRdScan, daqAdcRdNFore, daqAdcSetTag, daqAdcSetClk, daqAdcSetTrig`
Program References	ADC1 (VB, C)
Used With	

`daqAdcRdScanN` reads multiple scans from multiple A/D channels. This function will configure the pacer clock, arm the trigger and acquire **`count`** scans consisting of each channel starting with **`startChan`** and ending with **`endChan`**.

daqAdcSetBank

DLL Function	`daqAdcSetBank ( int chan, int type )`
C	`daqAdcSetBank( unsigned int chan, unsigned int bankType );`
QuickBASIC	`QBdaqAdcSetBank%(chan%, bankType%)`
Turbo Pascal	`daqAdcSetBank( chan:word; bankType:word ) : integer;`
Parameters	
`int chan`	Channel number on the DBK card. Channel numbers are in groups of 16 channels per bank.
`int type`	Type of channel bank.
Returns	`DerrInvChan` - Invalid Channel Number (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	

`daqAdcSetBank` internally programs intelligent DBK card channels so the Daq* gains may be set just before the acquisition. A bank consists of 16 channels, but **`daqAdcSetBank`** must be called once for each card in the bank. For example, if four 4-channel cards (such as a DBK7) are used in the first expansion bank, you must call **`daqAdcSetBank`** 4 times with channels 16, 20, 24, and 28. With only 1 such card, you cannot fill the remainder of the bank with another type of device.

daqAdcSetClk

DLL Function	`daqAdcSetClk(uint ctr1, uint ctr2);`
C	`daqAdcSetClk(uint ctr1, uint ctr2)`
QuickBASIC	`QBdaqAdcSetClk%(ctr1%, ctr2%)`
Turbo Pascal	`daqAdcSetClk(ctr1:word; ctr2:word ):integer;`
Parameters	
`uint ctr1`	The value of the counter 1 divisor Valid values: 0 - 65535
`uint ctr2`	The value of the counter 2 divisor Valid values: 0 - 65535
Returns	`DerrInvClock` - Invalid clock `DerrNoError` -No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetFreq, daqAdcGetFreq`
Program References	ADC3(VB, C) ? ADC4(All Languages)
Used With	

`daqAdcSetClk` sets the frequency of the pacer clock using the two specified counter values. The pacer clock can be used to control the sampling rate of the A/D converter.

The pacer clock is primarily used in two ways:

- As an internal trigger source to acquire scan data at a constant frequency. The trigger source is programmed as the PacerClock (**`daqAdcSetTrig()`**); and setting the scan frequency is done using **`daqAdcSetClk()`** or **`daqAdcSetFreq()`**. When configured, scan data will be acquired immediately at the frequency selected and without an external or software trigger.
- To control the frequency at which continuous mode acquisition is acquired. Here, the pacer clock is not used as a trigger source to initiate an acquisition. Instead it is used to pace the acquisition of data that has been initiated by another trigger source in continuous mode. The pacer clock is configured using **`daqAdcSetClk()`** and **`daqAdcSetFreq()`** functions. The trigger source, however, is something other than the pacer clock. When configured, the Daq* will wait for the selected trigger. When the trigger is detected, the Daq* will collect scans at the selected pacer-clock frequency.

The frequency is defined to be x_{tal} / ctr1* ctr2) where x_{tal} is the frequency of the board crystal (either 1 MHz or 10 MHz). For Daq PCMCIA, the following equation can be used to calculate frequency —x_{tal} can be 5 MHz, 1 MHz, 100 kHz, or Ext and is set by the **`daqBrdAdcSetTimeBase`** command.

$$\text{frequency} = \text{xtal} / [(\text{ctr1} * 65536) + \text{ctr2} + 1]$$

daqAdcSetFreq

DLL Function	daqAdcSetFreq(float freq);
C	daqAdcSetFreq(float freq)
QuickBASIC	QBdaqAdcSetFreq%(freq!)
Turbo Pascal	daqAdcSetFreq(freq:real) :integer;
Parameters	
float freq	The sampling frequency in Hz Valid values: 100000.0 - 0.0002
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 5-68)
See Also	daqAdcGetFreq, daqAdcSetClk
Program References	None
Used With	

daqAdcSetFreq calculates and sets the frequency of the pacer clock using the frequency specified in Hz. The frequency is converted to two counter values that control the frequency of the pacer clock (in this conversion, some resolution of the frequency may be lost). **daqAdcRdFreq** can be used to read the exact frequency setting of the pacer clock. **daqAdcSetClk** can be used to explicitly set the two counter values of the pacer clock. The pacer clock can be used to control the sampling rate of the A/D converter.

Note: The Daq PCMCIA may limit the maximum frequency settings when configured for differential mode operation.

daqAdcSetMux

DLL Function	daqAdcSetMux(uint startChan, uint endChan, uchar gain);
C	daqAdcSetMux(uint startChan, uint endChan)
QuickBASIC	QBdaqAdcSetMux%(startChan%, EndChan%)
Turbo Pascal	daqAdcSetMux(startChan:word; endChan:word; gain:byte) :integer;
Parameters	
uint startChan	The starting channel of the scan group
uint endChan	The ending channel of the scan group
uchar gain	The gain value for all channels
Returns	DerrInvGain - Invalid gain DerrIncChan - Invalid channel DerrNoError - No error (also, refer to *API Error Codes* on page 5-68)
See Also	daqAdcSetScan, daqAdcGetScan, daq200GetScan, daq200SetScan
Program References	ADC2 (All Languages)
Used With	

daqAdcSetMux sets a simple scan sequence of local A/D channels from startChan to **endChan** with the specified gain values. This command provides a simple alternative to **daqAdcSetScan** if consecutive channels need to be acquired.

daqAdcSetScan

DLL Function	`daqAdcSetScan(uint *chans, uchar *gains, uint count);`
C	`daqAdcSetScan(uint *chans, uchar *gains, uint count)`
QuickBASIC	`QBdaqAdcSetScan%(buf%())`
Turbo Pascal	`daqAdcSetScan(chans:DataP; gains:ByteP; count:word ) :integer;`
Parameters	
`uint *chans`	An array of up to 512 channel numbers
`uchar *gains`	An array of up to 512 gain values
`uint count`	The number of values in the chans and gains arrays Valid values: 1 -512
Returns	`DerrNotCapable` - No high speed digital `DerrInvGain` - Invalid gain `DerrInvChan` - Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcGetScan, daqAdcSetMux, daq200SetScan`
Program References	ADC3, ADC4 (All Languages)
Used With	

`daqAdcSetScan` configures a scan sequence consisting of multiple channels and corresponding gains. As many as 512 entries can be made in the scan configuration. Any analog input channel at any gain can be included in the scan including local channels and channels on an expansion card. Channels can be entered multiple times at the same or different gain. The high-speed digital I/O port can be included although its gain value will be ignored.

daqAdcSetTag

DLL Function	`daqAdcSetTag(uchar tag);`
C	`daqAdcSetTag(int cycle, int tag)`
QuickBASIC	`QBdaqAdcSetTag%(tag%)`
Turbo Pascal	`daqAdcSetTag(tag:byte ) :integer;`
Parameters	
`uchar tag`	A flag which if non-zero will enable the channel tag in the A/D data, or if 0 will disable it
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcConvertTagged`
Program References	ADC1, ADC2 (All Languages) ADC4 (VB, C)
Used With	**For Daq*100 series only**

`daqAdcSetTag` enables/disables channel numbers or tags to be included in the A/D data. The A/D converter of the Daq* returns data in a 16-bit tagged format which consists of the 12-bit A/D reading in the upper 12 bits and a channel number in the lower 4 bits. Disabling channel tagging will cause all A/D functions to strip the channel tags and shift the A/D data right by 4 bits before the data is returned to the user. If an expansion card is used, the jumper setting of the expansion card will be returned in the channel tag location. The **`daqAdcConvertTagged`** function can be used to convert data acquired with channel tags into separate channel and data arrays. High-speed acquisitions should enable channel tagging to increase throughput because no time is spent converting the data.

Note: The channel tag should not be disabled when reading data from the high-speed digital I/O port. The data from this port will be stripped and shifted like A/D data.

Note: The Daq PCMCIA/112B does not support channel tagging. Enabling channel tagging with a Daq/112B will simply cause the 12-bit data to be converted to 16-bit data. The least significant nibble will be filled with **`0H`** rather than a channel **`tag`** value. The remaining 12 bits will be shifted left 4 places.

daqAdcSetTrig

DLL Function	`daqAdcSetTrig(uchar trigger, uchar one shot, int level, uchar ctr0 mode, uchar pacer Mode);`
C	`daqAdcSetTrig(int source, int slope, int level)`
QuickBASIC	`QBdaqAdcSetTrig%(source%, slope%, level%)`
Turbo Pascal	`daqAdcSetTrig(trigger:byte; oneShot:byte; level:word; ctr0Mode:byte;pacerMode: byte ):integer;`
Parameters	
`uchar trigger`	The trigger source
`uchar one Shot`	A flag that if non-zero enables 1-shot trigger mode, otherwise enables continuous mode
`uint level`	The trigger level if an analog trigger is specified Valid values: 0 -4095
`ctr0mode`	A flag that if non-zero, selects an internal 100 kHz clock to be the input to counter 0. If the flag is zero, only the external clock on P1, pin21 is the input to counter 0 (see figure in `daqAdcConfCntr0`). Counter 0 can act as a trigger source if the Counter 0 output (pin 2 of P1) is connected to the external trigger input (pin 25 of P1). The JP1 jumper must be configured for Counter 0 for this operation.
`pacer Mode`	A flag that if zero, disables the external TTL Trigger (P1, pin 25) from affecting the pacer clock. If the flag is non-zero, any low-level on the TTL trigger will cause the pacer clock to pause.
Returns	`DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid level `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcConfCntr0`
Program References	ADC2, ADC3, ADC4, ADC5 (All Languages)
Used With	

`daqAdcSetTrig` sets and arms the trigger of the A/D converter. Eight trigger sources and several mode flags can be used for a variety of acquisitions. **`daqAdcSetTrig`** will stop current acquisitions, empty acquired data, and arm the Daq* using the specified trigger.

The **pacer-clock trigger** is used to acquire data at a constant frequency. The sampling rate can be set using the **`daqAdcSetClk`** or **`daqAdcSetFreq`** functions. The 1-shot flag has no meaning when using this trigger source, and the analog-level value is ignored.

The **software trigger** allows the user to trigger the A/D from software using the **`daqAdcSoftTrig`** function. When the 1-shot mode is enabled, a single scan will be initiated by the software trigger. In the continuous mode (1-shot disabled), sending a software trigger will cause the A/D converter to sample at the rate of the pacer clock. The analog level value is ignored.

An **external TTL pulse trigger** can be used to initiate a scan or start an acquisition when using the external TTL rising or falling edge trigger source. The external TTL pulse should be applied to trig0 (pin 25 of P1). The pulse will initiate a single scan in one-shot mode, and a continuous acquisition at the pacer clock frequency in continuous mode. The analog level value is ignored.

Four **analog triggers** use a rising or falling slope and a positive or negative level. Data acquisition will start when the first channel of the scan group (defined by **`daqAdcSetScan`** or **`daqAdcSetMux`**) passes through the specified trigger level with the proper slope. Analog triggers are used with the 1-shot mode disabled, so data will be collected at the frequency of the pacer clock after the analog trigger is satisfied. With an analog trigger, D/A channel 1 is set to the specified trigger level.

Note: The Daq* includes hysteresis to prevent false triggers on the wrong slope of the waveform due to noise. This hysteresis may cause the actual trigger level to differ from the program trigger level. For example, with a rising slope specified, the actual trigger level will be higher than the program trigger level depending on the frequency of the waveform on the trigger channel.

Setting the **counter 0 mode** flag **`true`** enables an onboard 100 kHz clock to be ANDed with the counter 1 input (pin 21 of P1) to produce the input to counter 0. If nothing is connected to counter 1 input, the line will float high and clock counter 0 from the 100 kHz clock. If the flag is **`false`**, counter 0 can only be clocked from the counter 0 input pin. Counter 0 can be used as an alternative trigger source by connecting the counter 0 output (pin 2 of P1) to trig0 (pin 25 of P1) and choosing an external TTL trigger. Counter 0 can also be used for general counter applications.

The **pacer mode** flag enables/disables operation of the pacer clock. If this flag is non-zero, the pacer clock will be gated with trig0 (pin 25 of P1). If it is zero, the pacer clock will be enabled.

daqAdcSetTrigPreT

DLL Function	`daqAdcSetTrigPreT(uchar trigger, uint channel, uint level, uint precount, uint postcount);`
C	`daqAdcSetTrigPreT(uchar trigger, uint channel, uint level, uint precount, uint postcount)`
QuickBASIC	`QBdaqAdcSetTrigPreT%(trigger%, channel%, level%, precount%, postcount%)`
Turbo Pascal	`daqAdcSetTrigPreT(trigger:channel:level:precount:postcount):integer;`
Parameters	
`uchar trigger`	The analog trigger source - DtsAnalogRisePos, DtsAnalogFallPos, DtsAnalogRisNeg, DtsAnalogFallNeg
`uint channel`	The channel in the current scan group to trigger on
`uint level`	The level for the specified channel at which to detect the trigger
`uint precount`	The number of pre-trigger scans to collect before arming the trigger (1-32767)
`uint postcount`	The number of post-trigger scans to collect after the occurrence of the trigger (1-32767)
Returns	`DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid level `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetFreq, daqAdcSetCtr, daqAdcRdNForePreT, daqAdcRdNForePreTWait, daqAdcRdNBackPreT`
Program References	ADCPRET1.C, ADCPRET2.C, ADCPRET3.C
Used With	

`daqAdcSetTrigPreT` sets the trigger for analog level triggering and initiates the collection of a pre-trigger data acquisition. **`daqAdcSetTrigPreT`** will stop a current acquisition, empty data acquired, arm the Daq* using the specified analog-level trigger source, and immediately begin the collection of the specified amount of pre-trigger data.

This command can configure a data acquisition that includes both pre-trigger and post-trigger data.

- The **pre-trigger** amount indicates the number of pre-trigger scans to be collected before the trigger is armed. The trigger event will only be recognized after the specified pre-trigger amount has been satisfied and the trigger is armed. The specified pre-trigger amount represents the minimum amount of pre-trigger data that will be collected.
- The **post-trigger** amount represents the number of scans taken after the detection of the trigger event. This amount represents the exact number of scans taken subsequent to the detection of the trigger event.

The pacer clock may be used to set up the sampling rate for the acquisition. The sampling rate can best be set by using the **`daqAdcSetClk`** or **`daqAdcSetFreq`** commands.

The four analog trigger sources (rising or falling slope with a positive or negative level) can be used with any one of the channels in the currently defined scan group. This channel parameter represents the relative channel within the scan group (not necessarily the actual channel number).

The **`level`** parameter is the A/D count level (normalized to a 12-bit quantity) at which the trigger is to occur. Since the internal DAC is not used to set the trigger level, there is no need to call **`daqAdcCalcTrig`** to determine the appropriate level. The level is simply calculated by:

$$\text{Level (if bipolar)} = (V_{SET}/V_{SPAN} \times 4096) + 2048$$

$$\text{Level (if unipolar)} = (V_{SET}/V_{SPAN} \times 4096)$$

When setting up a pre-trigger acquisition, a specific command set must be used to retrieve the data. This command set includes **`daqAdcRdNForePreT, daqAdcRdNForePreTWait`** and **`daqAdcRdNBackPreT`** (refer to their description in this chapter).

daqAdcSoftTrig

DLL Function	`daqAdcSoftTrig(void);`
C	`daqAdcSoftTrig(void)`
QuickBASIC	`QBdaqAdcSoftTrig%`
Turbo Pascal	`daqAdcSoftTrig():integer;`
Parameters	None
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcSetTrig`
Program References	ADC3 (All Languages)
Used With	

daqAdcSoftTrig is used to send a software trigger command to the DaqBook/DaqBoard. This software trigger can be used to initiate a scan or an acquisition from a program after configuring the software trigger as the trigger source.

daqAdcStopBack

DLL Function	`daqAdcStopBack(void);`
C	`daqAdcStopBack(void)`
QuickBASIC	`QBdaqAdcStopBack%()`
Turbo Pascal	`daqAdcStopBack():integer;`
Parameters	None
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcRdNBack, daqAdcGetBackStat`
Program References	ADC4 (All Languages)
Used With	

daqAdcStopBack stops a background operation initiated by the daqAdcRdNBack function.

daqAdcWtCntr0

DLL Function	`daqAdcWtCntr0(uint cntr0);`
C	`daqAdcWtCntr0(uint cntr0)`
QuickBASIC	`QBdaqAdcWtCntr0%(cntr0%)`
Turbo Pascal	`daqAdcWtCntr0(cntr0:word ) :integer;`
Parameters	
`uint cntr0`	The value to write to the count down register of Counter 0 Valid values: 0 -65535
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcConfCntr0, daqAdcRdCntr0`
Program References	ADC5 (All Languages)
Used With	

`daqAdcWtCntr0` loads the count-down register of Counter 0. See **`daqAdcConfCntr0`** for various applications of counter 0.

Note: Using counter 0 requires the JP2 jumpers to be in the -OCTOUT and -OCLKIN positions.

daqBrdAdcSetTimeBase

DLL Function	`daqBrdAdcSetTimeBase( uchar frequency);`	
C	`daqBrdAdcSetTimeBase( unsigned char frequency);`	
QuickBASIC	`QBdaqBrdAdcSetTimeBase%(frequency%)`	
Turbo Pascal	`daqBrdAdcSetTimeBase( frequency:byte):integer;`	
Parameters		
`uchar`	One of four predefined constants (see below).	
Frequencies:		
Description	**Value**	**Notes**
`TB10MHz`	`00h`	Used by DaqBoards only
`TB5MHz`	`01h`	Used by DaqBoard and Daq PCMCIA
`TB1MHz`	`02h`	Used by DaqBoard and Daq PCMCIA
`TB100kHz`	`03h`	Used by DaqBoard and Daq PCMCIA
`TBExternal`	`04h`	Used by Daq PCMCIA only
Returns	`DerrInvClock` - An invalid frequency was specified `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqBrdDacSetTimeBase`	
Program References	None	
Used With		

`daqBrdAdcSetTimeBase` is used to set the main timebase for the ADC pacer clock. The timebase can be set to 10 MHz, 5 MHz, 1 MHz, or 100 kHz. A 1-MHz clock is set as the default value when the hardware is initialized. The main timebase is then divided by Counter 1 & Counter 2 of the 8254 to determine the pacer clock frequency that will drive the ADC scan rate.

daqBrdDacClockSrc

DLL Function	`daqBrdDacClockSrc( uchar source);`	
C	`daqBrdDacClockSrc(source:byte):integer;`	
QuickBASIC	`QBdaqBrdDacClockSrc%(source%)`	
Turbo Pascal	`daqBrdDacClockSrc(source:byte):integer;`	
Parameters		
`uchar source`	One of five predefined constants. See table below for listing.	
Clock Sources:		
Source	**Value**	**Description**
`DacPcrExt`	`00hD`	AC FIFO driven by user supplied external clock
`DacPcrTB9513`	`01hD`	AC FIFO driven by 9513 Counter 1
`DacPcrTBInt`	`02hD`	AC FIFO driven by DAC clock which is set by software to 10 MHz, 5 MHz, 1 MHz or 100 kHz
`DacPcrGated`	`03h`	The internal DAC time base (10, 5, 1, 0.1 MHz set by `daqBrdDacTimeBase`) is gated by the external TTL trigger found on pin 25 of P1.
`DacPcrAdcPcr`	`04h`	DAC FIFO driven by ADC pacer clock
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqBrdDacCtrl, daqBrdDacResetFIFO, daqBrdDacSetTimeBase, daqBrdDacWriteFIFO`	
Program References	DAC3 (All Languages)	
Used With		

`daqBrdDacClockSrc` is used to select the source for the DAC FIFO pacer clock. There are 4 sources for the pacer clock:

- the internal DAC time base (set to 1 of 4 frequencies by **`daqBrdDacSetimeBase`**)
- a time base driven by Counter 1 of the 9513
- an external time base that is supplied by P1 pin 21
- the ADC pacer time base.

The first 3 of these potential clock sources pass through and are divided by Counter 0 of the 8254 before they reach the DAC FIFO.

The DAC FIFO pacer clock's frequency is the rate that a sample will be sent from the FIFO to the DACs. If the **`DACFIFOChan0`** or **`DACFIFOSimul`** modes are selected, samples will be sent to DAC 0 or both DACs at this rate.

If the mode is set to **`DACFIFOInterleave`**, samples will leave the FIFO at this rate. Since samples are sent alternately to DAC 0 and DAC 1, each DAC's value will be updated at half this rate.

daqBrdDacCtrl

DLL Function	`daqBrdDacCtrl( uchar mode, uchar retransmit);`	
C	`daqBrdDacCtrl(mode:byte; retransmit:byte):integer;`	
QuickBASIC	`QBdaqBrdDacCtrl%(mode%, retransmit%)`	
Turbo Pascal	`daqBrdDacCtrl(mode:byte; retransmit:byte):integer;`	
Parameters		
`uchar mode`	One of four predefined constants (see below)	
`uchar retransmit`	A flag that if non-zero will enable the FIFO to be output continuously to the DAC channels.	
Modes:		
Mode	**Value**	**Description**
`DacFIFOBypass`	`00h`	DaqBook compatible mode. DAC value updated by **`daqDacWt`** command. To stop the waveforms, use this mode.
`DacFIFOChan0`	`01h`	FIFO data goes to chan0, chan1 set in FIFO bypass mode.
`DacFIFOInterleave`	`02h`	Chan 0 and chan1's samples are interleaved in the FIFO.
`DacFIFOSimul`	`03h`	FIFO data goes to chan0 and chan1 simultaneously.
Returns	**`DerrNotCapable`** - Hardware is not capable of this function **`DerrNoError`** - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqBrdDacClockSrc, daqBrdDacResetFIFO, daqBrdDacSetTimeBase, daqBrdDacWriteFIFO`	
Program References	DAC3 (All Languages)	
Used With		

`daqBrdDacCtrl` is used to set the mode of the DAC FIFO. The first parameter determines which DAC channels the samples in the FIFO are sent to. The second parameter determines whether the FIFO is transmitted continuously or only once.

daqBrdDacPredefWave

DLL Function	`daqBrdDacPredefWave( uchar dac, uint samples, uchar waveType, uint amplitude, uint offset, uint dutyCycle, uint phaseShift );`
C	`daqBrdDacPredefWave(dac:byte; samples:word; waveType:byte amplitude:word; offset:word; dutyCycle:word; phaseShift:word):integer;`
QuickBASIC	`QBdaqBrdDacPredefWave%(DAC%,samples%, waveType%, amplitude%, offset%, dutyCycle%, phaseShift%)`
Turbo Pascal	`daqBrdDacPredefWave(DAC:byte; samples:word; waveType:byte amplitude:word; offset:word; dutyCycle:word; phaseShift:word):integer;`
Parameters	
`uchar dac`	The DAC channel to assign the waveform to.
`uint samples`	The number of samples in one cycle of the waveform. (QuickBASIC Note: Waveforms constructed with this command are limited to 256 samples per waveform.)
`uchar waveType`	One of the three predefined waveforms from table below.
`uint amplitude`	The peak to peak amplitude of the waveform in D/A counts. 0 -4095
`uint offset`	The voltage level that the waveform will be centered around in D/A counts. 0 -4095
`uint dutyCycle`	The duty cycle of the waveform as a percentage.
`uint phaseShift`	The number of degrees that the waveform is shifted from the waveform of the other DAC channel.
Predefined Waveforms:	
Description	**Value**
`PdwSine`	`00h`
`PdwSquare`	`01h`
`PdwTriangle`	`02h`
Returns	`DerrInvDacChan` - The DAC channel number doesn't exist `DerrInvDacParam` - Parameters were out of range `DerrInvPredefWave` - Predefined waveform is not supported `DerrMemAlloc` - Not enough memory was available to build the waveform `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacSetMode, daqBrdDacStart, daqBrdDacStop, daqBrdDacUserWave`
Program References	DAC2 (All Languages)
Used With	

daqBrdDacPredefWave builds a waveform to the user's specifications and assigns it to one of the DAC channels. Waveforms assigned with this command are started with the **daqBrdDacStart** command and stopped with the **daqBrdDacStop** command. **daqBrdSetMode** is used to set the update rate and cycling mode for this waveform.

daqBrdDacResetFIFO

DLL Function	`daqBrdDacResetFIFO(void);`
C	`daqBrdDacResetFIFO():integer;`
QuickBASIC	`QBdaqBrdDacResetFIFO%()`
Turbo Pascal	`daqBrdDacResetFIFO():integer;`
Parameters	None
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacClockSrc, daqBrdDacCtrl, daqBrdDacSetTimeBase, daqBrdDacWriteFIFO`
Program References	DAC3 (All Languages)
Used With	

`daqBrdDacResetFIFO` resets the DAC FIFO and its pointers.

daqBrdDacSetMode

DLL Function	`daqBrdDacSetMode( ulong period, uchar mode, uchar cycle);`	
C	`daqBrdDacSetMode(period:longintmode:byte; cycle:byte):integer;`	
QuickBASIC	`QBdaqBrdDacSetMode%(period!, mode%, cycle%)`	
Turbo Pascal	`daqBrdDacSetMode(period:longintmode:byte; cycle:byte):integer;`	
Parameters		
`ulong period`	The rate that sample data will be sent to the DACs specified in microseconds. Note: This rate is per sample per DAC channel regardless of what mode the FIFO is set to.	
`uchar mode`	One of four modes for the DAC FIFO from the table below.	
`uchar cycle`	A flag that if non-zero will enable the waveforms to be output continuously to the DAC channels.	
Modes:		
Mode	**Value**	**Description**
`DacFIFOBypass`	`00h`	DaqBook compatible mode. DAC value updated by `daqDacWt` command.
`DacFIFOChan0`	`01h`	FIFO goes to chan0, chan1 set in FIFO bypass mode.
`DacFIFOBoth`	`02h`	Sends waveforms defined by `daqBrdUserWave` and `daqBrdDacPredfedWave` to the DACs.
`DacFIFOSimul`	`03h`	FIFO goes to chan0 and chan1 simultaneously from waveform defined in `Dac0` by `daqBrdDacUserWave` or `daqBrdDacPredefWave`
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqBrdDacPredefWave, daqBrdDacStart, daqBrdDacStop, daqBrdDacUserWave`	
Program References	DAC2 (All Languages)	
Used With		

daqBrdDacSetMode sets the mode of the DAC FIFO (which DAC channels the FIFO samples will be sent to), the cycling mode of the FIFO, and the update rate per sample. This command works in conjunction with the **`daqBrdDacStart, daqBrdDacStop, daqBrdDacPredefWave`** and **`daqBrdDacUserWave`** commands. Note: It does not configure the hardware registers until **`daqBrdDacStart`** is called.

daqBrdDacSetTimeBase

DLL Function	`daqBrdDacSetTimeBase( uchar frequency);`
C	`daqBrdDacSetTimeBase(frequency:byte):integer;`
QuickBASIC	`QBdaqBrdDacSetTimeBase%(frequency%)`
Turbo Pascal	`daqBrdDacSetTimeBase(frequency:byte):integer;`
Parameters	
`uchar frequency`	**One of four predefined constants (see below)**
Frequencies:	
Description	**Value**
`TB10MHz`	`00h`
`TB5MHz`	`01h`
`TB1MHz`	`02h`
`TB100kHz`	`03h`
Returns	`DerrInvClock` - An invalid frequency was specified `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacClockSrc, daqBrdDacCtrl, daqBrdDacResetFIFO, daqBrdDacWriteFIFO`
Program References	DAC3 (All Languages)
Used With	

`daqBrdDacSetTimeBase` is used to set the main timebase for the DAC FIFO pacer clock. This clock can be set to 10 MHz, 5 MHz, 1 MHz, or 100 kHz. The default value is 1 MHz and is set when the hardware and the driver are initialized. This timebase is then divided by Counter 0 of the 8254 chip. This frequency will be the rate at which the DACs will be updated with samples from the FIFO (if the clock source is set to **`DacPcrTBInt`**). **Note**: This timebase may be changed by **`daqBrdDacSetMode`**.

daqBrdDacStart

DLL Function	`daqBrdDacStart( void );`
C	`daqBrdDacStart():integer;`
QuickBASIC	`QBdaqBrdDacStart%()`
Turbo Pascal	`daqBrdDacStart():integer;`
Parameters	None
Returns	`DerrInvDacWave` - Inappropriate dac mode is set `DerrInvBackDac` - Waveforms would not fit in FIFO or their sample sizes were not equal `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacPredefWave, daqBrdDacSetMode, daqBrdDacStop, daqBrdDacUserWave`
Program References	DAC2 (All Languages)
Used With	

`daqBrdDacStart` starts the waveforms specified by **`daqBrdDacPredefWave`**, **`daqBrdDacSetMode`**, and **`daqBrdDacUserWave`**. The total size in samples of all the waveforms started with this command must be smaller or equal to the size of the DAC FIFO (4096 samples). If two waveforms are configured, the number of samples in each waveform must be equal.

daqBrdDacStop

DLL Function	`daqBrdDacStop( void );`
C	`daqBrdDacStop():integer;`
QuickBASIC	`QBdaqBrdDacStop%()`
Turbo Pascal	`daqBrdDacStop():integer;`
Parameters	None
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacPredefWave, daqBrdDacSetMode, daqBrdDacStart, daqBrdDacUserWave`
Program References	DAC2 (All Languages)
Used With	

`daqBrdDacStop` stops the waveforms specified by **`daqBrdDacPredefWave,`** **`daqBrdDacSetMode`**, and **`daqBrdDacUserWave`** that were started with **`daqBrdDacStart`**. **Note**: Use the DAC FIFO bypass mode to stop the waveforms.

daqBrdDacUserWave

DLL Function	`daqBrdDacUserWave( uchar dac, uint *buf, uint samples );`
C	`daqBrdDacUserWave(dac:byte; buf:WordP; samples:word):integer;`
QuickBASIC	`QBdaqBrdDacUserWave%(DAC%, buf%(), samples%)`
Turbo Pascal	`daqBrdDacUserWave(DAC:byte; buf:WordP; samples:word):integer;`
Parameters	
`uchar dac`	The dac channel to assign the waveform to.
`uint *buf`	A pointer to the beginning of an array that contains the samples that will be assigned to the DAC channel.
`uint samples`	The number of samples contained in the array. (QuickBASIC Note: Waveforms constructed with this command are limited to 256 samples per waveform.)
Returns	`DerrInvDacChan` - The DAC channel number doesn't exist `DerrInvBuf` - A waveform buffer was not specified `DerrMemAlloc` - Not enough memory was available to build the waveform `DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacPredefWave, daqBrdDacSetMode, daqBrdDacStart, daqBrdDacStop`
Program References	DAC2 (All Languages)
Used With	

`daqBrdDacUserWave` assigns a user-defined waveform to one of the DAC channels. Any arbitrary waveform can be built in an array. **`daqBrdDacUserWave`** can then be called by specifying a pointer to the beginning of the waveform, the size of the array, and the target DAC channel to send the waveform. The waveform will start when **`daqBrdDacStart`** is called, as long as all parameters have been set properly. **Note**: the waveform is not loaded into its FIFO until **`daqBrdDacStart`** is called.

daqBrdDacWriteFIFO

DLL Function	`daqBrdDacWriteFIFO( uint samples,uint far *storage );`
C	`daqBrdDacWriteFIFO(samples:word; storage:WordP):integer;`
QuickBASIC	`QBdaqBrdDacWriteFIFO%(samples, storage%()) )`
Turbo Pascal	`daqBrdDacWriteFIFO(samples:word; storage:WordP):integer;`
Parameters	
`uint samples`	The number of samples to be loaded into the FIFO
`uint far *storage`	A pointer to the beginning of an array that contains the samples that will be load into the DAC FIFO
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqBrdDacClockSrc, daqBrdDacCtrl, daqBrdDacResetFIFO, daqBrdDacSetTimeBase`
Program References	DAC3 (All Languages)
Used With	

daqBrdDacWriteFIFO loads sample data directly into the DAC FIFO.

daqBrdSetDmaMode

DLL Function	`daqBrdSetDmaMode( int mode );`	
C	`daqBrdSetDmaMode( int mode );`	
QuickBASIC	`QBdaqBrdSetDmaMode%( mode% )`	
Turbo Pascal	`daqBrdSetDmaMode( mode:integer ):integer;`	
Parameters		
`int`	One of two predefined constants (see below)	
Modes:		
Mode	**Value**	**Description**
`DmaNone`	`00h`	Do not use DMA
`DmaRead`	`02h`	Use DMA to transfer scan data from the ADC FIFO to memory
Returns	`DerrNotCapable` - Hardware is not capable of this function `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqAdcRdNBack, daqInit`	
Program References	ADC6	
Used With		

`daqBrdSetDmaMode` is used to set the direction for DMA transfers. Of the two possible modes, **`DmaNone`** will disable any DMA transfers and **`DmaRead`** will enable scan data to be transferred from the ADC FIFO to memory via a DMA channel. For this transfer to take place, the DMA channel number (5, 6, or 7) must be specified when **`daqInit`** is called and **`daqAdcRdNBack`** must be called with the **`updateSingle`** parameter set to **`DusDma`** (02h).

daqCalConvert

DLL Function	`daqCalConvert(uint *counts, uint scans);`
C	`daqCalConvert(uint *counts, uintscans)`
QuickBASIC	`QBdaqCalConvert%(counts%, scans%)`
Turbo Pascal	`daqCalConvert(counts:scans):integer;`
Parameters	
`uint *counts`	The raw data from one or more scans.
`uint scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	daqReadCalFile, daqCalSetup, daqCalSetupConvert
Program References	None
Used With	

`daqCalConvert` function performs the actual calibration of one or more scans according to the previously called **`daqCalSetup`** function. This function will modify the array of data passed to it.

daqCalSetup

DLL Function	`daqCalSetup(uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint startChan, uint bipolar, uint noOffset);`
C	`daqCalSetup(uintnscan, uint readingsPos, uint nReadings, uintchanType, uintchanGain, uint startChan, uintbipolar, uintnoOffset)`
QuickBASIC	`QBdaqCalSetup%(scan%, readingsPos%, nReadings%, chanType%, chanGain%, startChan, bipolar%, noOffset%)`
Turbo Pascal	`daqCalSetup(nscan:readingsPos:nReadings:chanType:chanGain:startChan:bipolar:noOffset):integer;`
Parameters	
`uint nscan`	The number of readings in a single scan.
`uchar readingsPos`	The position of the readings to be calibrated within the scan.
`uchar nReadings`	The number of readings to calibrate.
`uint chanType`	The type of channel/board from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel of a DBK14 or DBK19, and 0 when reading any other channel.
`uint chanGain`	The gain setting of the channels to be calibrated.
`uint startChan`	The channel number of the first channel to be converted.
`uint bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`uint noOffset`	If non-zero, the offset cal constant will not be used to calibrate the readings.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqReadCalFile, daqCalConvert, daqCalSetupConvert`
Program References	None
Used With	

daqCalSetup will configure the order and type of data to be calibrated. This function requires all data to be calibrated to be from consecutive channels configured for the same gain, polarity, and channel type. The calibration can be configured to only use the gain calibration constant and not the offset constant. This allows the offset to be removed at runtime using the zero compensation functions.

daqCalSetupConvert

DLL Function	`daqCalSetupConvert(uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint startChan, uint bipolar, uint noOffset, uint *counts, uint scans);`
C	`daqCalSetupConvert(uintnscan, uint readingsPos, uint nReadings, uintchanType, uintchanGain, uint startChan, uintbipolar, uintnoOffset, uint *counts, uint scans)`
QuickBASIC	`QBdaqCalSetupConvert%(scan%, readingsPos%, nReadings%, chanType%, chanGain%, startChan, bipolar%, noOffset%, counts%, scans%)`
Turbo Pascal	`daqCalSetupConvert(nscan:readingsPos:nReadings:chanType:chanGain:startChan:bipolar:noOffset:counts:scans):integer;`
Parameters	
`uint nscan`	The number of readings in a single scan.
`uchar readingsPos`	The position of the readings to be calibrated within the scan.
`uchar nReadings`	The number of readings to calibrate.
`uint chanType`	The type of channel/board from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel of a DBK14 or DBK19, and 0 when reading any other channel.
`uint chanGain`	The gain setting of the channels to be calibrated.
`uint startChan`	The channel number of the first channel to be converted.
`uint bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`uint noOffset`	If non-zero, the offset cal constant will not be used to calibrate the readings.
`uint *counts`	The raw data from one or more scans.
`uint scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqReadCalFile, daqCalSetup, daqCalConvert`
Program References	None
Used With	

For convenience, both the setup and convert steps can be performed with one call to **`daqCalSetupConvert`**. This is useful when the calibration needs to be performed multiple times because data was read from non-consecutive channels or at different gains.

daqClose

DLL Function	`daqClose(void)`
C	`daqClose(void)`
QuickBASIC	`QBdaqClose%(void%)`
Turbo Pascal	`daqClose():integer;`
Parameters	None
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqInit`
Program References	INIT, ADC1, ADC2, ADC3, ADC4, ADC5,CTR1, CTR2, DAC1, DAC2, DIG1 (All Languages)
Used With	

`daqClose` is used to end communications with the DaqBook/DaqBoard. If **`daqClose`** is called, **`daqInit`** must be called before calling any other function.

daqCtrGetBackStat

DLL Function	`daqCtrGetBackStat(uchar *active, ulong *count);`
C	`daqCtrGetBackStat(int *active, uint *count)`
QuickBASIC	`QBdaqCtrGetBackStat%(active%, count%)`
Turbo Pascal	`daqCtrGetBackStat (active:ByteP; count:LongP ) :integer;`
Parameters	
`uchar *active`	A flag which will be returned non-zero if a background transfer is in progress, or 0 if not
`ulong *count`	The number of scans acquired by the last or current background transfer
Returns	`DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrRdNBack, daqCtrStopBack`
Program References	CTR2 (All Languages)
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqCtrGetBackStat` reads the status of the last or current background operation initiated by the **`daqCtrRdNBack`** function.

daqCtrGetHold

DLL Function	`daqCtrGetHold(uchar ctrNum, uint  *ctrVal);`
C	`daqCtrGetHold(uchar ctrNum, uint *ctrVal)`
QuickBASIC	`QBdaqCtrGetHold%(ctrNum%, ctrVal%)`
Turbo Pascal	`daqCtrGetHold(ctrNum:byte; ctrVal:DataP ):integer;`
Parameters	
uchar ctrNum	The counter number Valid values: 1 - 5
uint *ctrVal	The value read from the hold register of the selected counter is placed in this variable Valid values: 0 - 65535
Returns	`DerrInvCtrNum` - Invalid counter `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrSetCtrMode`
Program References	(None)
Used With	**Only applies to DaqBook/100/200 and DaqBoard/100A/200A**

`daqCtrGetHold` reads the hold register of the specified counter. This register is used in event-counting applications to store accumulated counter values. The hold register can be read while the count process is running without interrupting the process.

daqCtrMultCtrl

DLL Function	`daqCtrMultCtrl(uchar ctrCommand, uchar ctr1, uchar ctr2, uchar ctr3, uchar ctr4, uchar ctr5);`
C	`daqCtrMultCtrl(int ctrCommand, int ctrl1, int ctr2, int ctr3, int ctr4, int ctr5)`
QuickBASIC	`QBdaqCtrMultCtrl%(ctrCommand%, ctr1 %, ctr2 %, ctr3%, ctr4%, ctr5%)`
Turbo Pascal	`daqCtrMultCtrl(ctrCmd:byte; ctr1:byte; ctr2:byte; ctr3:byte; ctr4:byte; ctr5:byte ):integer;`
Parameters	
`uchar ctrCommand`	The counter command (see below)
`uchar ctr1`	A flag that if non-zero enables the counter command to be executed on counter 1, or if 0 do nothing to counter 1
`uchar ctr2`	A flag that if non-zero enables the counter command to be executed on counter 2, or if 0 do nothing to counter 2
`uchar ctr3`	A flag that if non-zero enables the counter command to be executed on counter 3, or if 0 do nothing to counter 3
`uchar ctr4`	A flag that if non-zero enables the counter command to be executed on counter 4, or if 0 do nothing to counter 4
`uchar ctr5`	A flag that if non-zero enables the counter command to be executed on counter 5, or if 0 do nothing to counter 5
Multiple Counter Commands:	
Description	Value
`DmccArm`	`20h`
`DmccLoad`	`40h`
`DmccLoadArm`	`60h`
`DmccDisarmSave`	`80h`
`DmccSave`	`A0h`
`DmccDisarm`	`C0h`
Returns	`DerrInvCtrCmd` - Invalid counter command `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	**daqCtrSetCtrMode, daqCtrSetMasterMode**
Program References	CTR1, CTR2 (All Languages)
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

daqCtrMultCtrl performs a command including loading, latching, saving, enabling, and disabling on multiple counters simultaneously.

- **DmccLoad** - The initial counter value can be transferred from the load or hold register with the load command.
- **DmccArm** - The arm command will enable the counter to begin counting.
- **DmccDisarm** - The disarm command will disable the counter.
- **DmccSave** - The save command will transfer the current counter value to the hold register where it can be read without disturbing the counters.

daqCtrRdFreq

DLL Function	`daqCtrRdFreq(uint interval, uchar cntSource, uint *count);`	
C	`daqCtrRdFreq(int interval, int source, int count)`	
QuickBASIC	`QBdaqCtrRdFreq%(interval%, source%, count%)`	
Turbo Pascal	`daqCtrRdFreq(interval:word; cntSource:byte; count:DataP ) :integer;`	
Parameters		
`uint interval`	The gate interval in milliseconds Valid values: 1 -32767	
`uchar cntSource`	The count source (see below)	
`uint *count`	A variable to hold the number of counts accumulated in the gating interval Valid values: 0 - 65535	
Count Source Definitions:		
Description	**Value**	**Description**
`DcsSrc1`	`01h`	Counter 1 input (pin 36 of P3)
`DcsSrc2`	`02h`	Counter 2 input (pin 19 of P3)
`DcsSrc3`	`03h`	Counter 3 input (pin 17 of P3)
`DcsSrc4`	`04h`	Counter 4 input (pin 15 of P3)
`DcsSrc5`	`05h`	Counter 5 input (pin 13 of P3)
`DcsGate1`	`06h`	Counter 1 gate (pin 37 of P3)
`DcsGate2`	`07h`	Counter 2 gate (pin 18 of P3)
`DcsGate3`	`08h`	Counter 3 gate (pin 16 of P3)
`DcsGate4`	`09h`	Counter 4 gate (pin 14 of P3)
Returns	`DerrInvInterval` - Invalid interval `DerrInvCntSource` - Invalid source `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also		
Program References	None	
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A	

`daqCtrRdFreq` is used to read the frequency of one of 9 external inputs. The 9 available inputs include the 5 counter inputs (P3 pins 36, 19, 17, 15, or 13) and the gates of counters 1 to 4 (P3 pins 37, 18, 16, and 14). This function counts the number of pulses on the specified input within a specified time interval, thereby providing the frequency of the signal. This frequency can be obtained by dividing the number of pulses by the interval (freq in kHz = count/interval).

Note: The counter 4 output (P3 pin 32) must be externally connected to the counter 5 gate (P3 pin 12). This function will reconfigure counters 4 and 5.

daqCtrRdNBack

DLL Function	`daqCtrRdNBack(uint *ctr1Buf, uint *ctr2Buf, uint *ctr3Buf, uint *ctr4Buf, uint *ctr5Buf, uint count, uchar cycle);`
C	`daqCtrRdNBack(int *ctr1Buf, int *ctr2Buf, int *ctr3Buf, int *ctr4Buf, int *ctr5Buf, int count, int cycle)`
QuickBASIC	`QBdaqCtrRdNBack%(ctr1Buf%(), ctr2Buf%(), ctr3Buf%(), ctr4Buf%(), ctr5Buf%(), count%, startIP0%, cycle%)`
Turbo Pascal	`daqCtrRdNBack(ctr1Buf:DataP; ctr2Buf:DataP; ctr3Buf:DataP; ctr4Buf:DataP; ctr5Buf:DataP;count:word; cycle:byte ):integer;`
Parameters	
`uint *ctr1Buf`	An array to hold count values from counter 1 or 0 if counter 1 is not to be read Valid values: 0 - 65535
`uint* ctr2Buf`	An array to hold count values from counter 2 or 0 if counter 2 is not to be read Valid values: 0 - 65535
`uint*ctr3Buf`	An array to hold count values from counter 3 or 0 if counter 3 is not to be read Valid values: 0 - 65535
`uint *ctr4Buf`	An array to hold count values from counter 4 or 0 if counter 4 is not to be read Valid values: 0 - 65535
`uint *ctr5Buf`	An array to hold count values from counter 5 or 0 if counter 5 is not to be read Valid values: 0 - 65535
`uint count`	The number of scans to be taken Valid values: 1 - 32767
`uchar cycle`	A flag that if non-zero will enable continuous operation, or if 0 will disable it
Returns	`DerrMultBackXfer` - Background task already started `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrStopBack, daqCtrGetBackStat`
Program References	None
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

daqCtrRdNBack reads the values of the specified counters in the background using interrupts. An interrupt will occur on the rising edge of the interrupt input (P3 pin 1) if the interrupt enable line (P3 pin 2) is pulled low. When an interrupt occurs, the save command (see **daqCtrMultCtrl**) will be sent to the selected counters and the hold register of the selected counters will be read (see **daqCtrGetHold**) and placed into the user's buffer. This function will return control back to the user's program after initiating the background process. The user can then monitor the status of the background transfer with the **daqCtrGetBackStat** function or stop the transfer with the **daqCtrBackStop** function. The user can perform other tasks in the foreground.

If the cycle flag is true, the background transfer will run continuously looping back to the beginning of the user's buffers after **count** readings have been read. This allows the user to read large amounts of data without calling **daqCtrRdNBack** multiple times. This background transfer can run indefinitely as long as the user monitors the status of the counter buffers and processes the data before it gets overwritten. In this mode, the user should get the total number of readings written into the buffer using **daqCtrGetBackStat** and keep track of the total number of scans processed in a variable. The difference between these two totals is the number of unprocessed valid readings the user can process.

daqCtrRdNFore

DLL Function	`daqCtrRdNFore(uint *ctr1Buf, uint *ctr2Buf, uint *ctr3Buf, uint *ctr4Buf, uint *ctr5Buf, uint count);`
C	`daqCtrRdNFore(int *ctr1Buf, int *ctr2Buf, int *ctr3Buf, int *ctr4Buf, int *ctr5Buf, int count)`
QuickBASIC	`QBdaqCtrRdNFore%(ctr1Buf%(), ctr2Buf%(), ctr3Buf%(), ctr4Buf%(), ctr5Buf%(), count%, startIP0%)`
Turbo Pascal	`daqCtrRdNFore(ctr1Buf:DataP; ctr2Buf:DataP; ctr3Buf:DataP; ctr4Buf:DataP; ctr5Buf:DataP; count:word ):integer;`
Parameters	
`uint ctr1Buf[]`	An array to hold count values from counter 1 or 0 if counter 1 is not to be read Valid values: 0 - 65535
`uint ctr2Buf`	An array to hold count values from counter 2 or 0 if counter 2 is not to be read Valid values: 0 - 65535
`uint ctr3Buf[]`	An array to hold count values from counter 3 or 0 if counter 3 is not to be read Valid values: 0 - 65535
`uint ctr4Buf[]`	An array to hold count values from counter 4 or 0 if counter 4 is not to be read Valid values: 0 - 65535
`uint ctr5Buf`	An array to hold count values from counter 5 or 0 if counter 5 is not to be read Valid values: 0 - 65535
`uint count`	The number of scans to be taken Valid values: 0 - 32767
Returns	`DerrMultBackXfer` - Background task already started `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqCtrRdNFore` operates identically to **`daqCtrRdNBack`** (using interrupts to acquire data) except that it will not return control to the user's program until all the counter readings are acquired.

daqCtrSetAlarm

DLL Function	`daqCtrSetAlarm(uchar alarmNum, uint alarmVal)`
C	`daqCtrSetAlarm(uchar alarmNum, uint alarmVal)`
QuickBASIC	`QBdaqCtrSetAlarm%(alarmNum%, alarmVal%)`
Turbo Pascal	`daqCtrSetAlarm(alarmNum:byte; alarmVal:word ):integer;`
Parameters	
`uchar alarmNum`	The alarm register number Valid values: 1 - 2
`uint alarmVal`	The value to write to the selected alarm register Valid values: 0 - 65535
Returns	`DerrInvCtrNum` - Invalid counter number `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrSetMasterMode`
Program References	None
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqCtrSetAlarm` sets the specified alarm register. This alarm register can be used with the comparators described in **`daqCtrSetMasterMode`**. The alarm register is only used if the corresponding comparator has been enabled using the **`daqCtrSetMasterMode`** function.

daqCtrSetCtrMode

DLL Function	`daqCtrSetCtrMode(uchar ctrNum, uchar gateCtrl, uchar cntEdge, uchar cntSource, uchar specGate, uchar reload, uchar cntRepeat, uchar cntType, uchar cntDir, uchar outputCtrl);`
C	`daqCtrSetCtrMode(int CtrNum, int gateCtrl, int cntEdge, int cntSource, int specGate, int reload, int cntRepeat, int cntType, int cntDir, int outputCtrl)`
QuickBASIC	`QBdaqCtrSetCtrMode%(ctrNum%, gateCtrl%, cntEdge%, cntSource%, specGate%, reload%, cntRepeat%, cntType%, cntDir%, outputCtrl%)`
Turbo Pascal	`daqCtrSetCtrMode(ctrNum:byte; gateCtrl:byte; cntEdge:byte; cntSource:byte; specGate: byte;reload:byte;cntRepeat:byte; cntType:byte; cntDir:byte; outputCtrl:byte ):integer;`

Parameters

`uchar ctrNum`	The counter number; Valid values: 1 - 5
`uchar gateCtrl`	The gating control mode (see below)
`uchar cntEdge`	A flag that if non-zero will select a positive count edge, or if 0 will select a negative count edge
`uchar cntSource`	The count source (see below)
`uchar specGate`	A flag that if non-zero will enable the special gate, or if 0 will disable it
`unchar reload`	A flag that if non-zero will select reload from load or hold, or if 0 will select reload from load
`uchar cntRepeat`	A flag that if non-zero will select count repetitively, or if 0 will select count once
`uchar cntType`	A flag that if non-zero will select a BCD count, or if 0 will select a binary count
`uchar cntDir`	A flag that if non-zero will select count up, or if 0 will select count down
`uchar outputCtrl`	The output control mode (see below)

Gating Control Definitions:

Definition	Value	Description
`DgcNoGating`	`00h`	Gating Disabled
`DgcHighTCNM1`	`20h`	Active level high of TC-toggled output of previous (N-1) counter
`DgcHighLevelGateNP1`	`40h`	Active level high of gate of next (N+1) counter
`DgcHighLevelGateNM1`	`60h`	Active level high of gate of next (N-1) counter
`DgcHighLevelGateN`	`80h`	Active level high of gate of selected (N) counter
`DgcLowLevelGateN`	`A0h`	Active level low of gate of selected (N) counter
`DgcHighEdgeGateN`	`C0h`	Active rising edge of gate of selected (N) counter
`DgcLowEdgeGateN`	`E0h`	Active falling edge of gate of selected (N) counter

Count Source Definitions:

Definition	Value	Description
`DcsTCNM`	`00h`	TC toggled output of previous (N-1) counter
`DcsSrc1`	`01h`	Counter 1 input (pin 36 of P3)
`DcsSrc2`	`02h`	Counter 2 input (pin 19 of P3)
`DcsSrc3`	`03h`	Counter 3 input (pin 17 of P3)
`DcsSrc4`	`04h`	Counter 4 input (pin 15 of P3)
`DcsSrc5`	`05h`	Counter 5 input (pin 13 of P3)
`DcsGate1`	`06h`	Counter 1 gate (pin 37 of P3)
`DcsGate2`	`07h`	Counter 2 gate (pin 18 of P3)
`DcsGate3`	`08h`	Counter 3 gate (pin 16 of P3)
`DcsGate4`	`09h`	Counter 4 gate (pin 14 of P3)
`DcsGate5`	`0Ah`	Counter 5 gate (pin 12 of P3)
`DcsF1`	`0Bh`	Onboard 1 MHz clock
`DcsF2`	`0Ch`	Onboard 100 kHz clock
`DcsF3`	`0Dh`	Onboard 10 kHz clock
`DcsF4`	`0Eh`	Onboard 1 kHz clock
`DcsF5`	`0Fh`	Onboard 100 Hz clock

Output Control Definitions:

Definition	Value	Description
`DocInactiveLow`	`00h`	Inactive - Always low
`DocHighTermCntPulse`	`01h`	High impulse on terminal count
`DocTCToggled`	`02h`	Toggled on terminal count
`DocInactiveHighImp`	`04h`	Inactive - High impedance
`DocLowTermCntPulse`	`05h`	Low pulse on terminal count

Returns	`DerrInvCtrNum` - Invalid channel `DerrInvGateCtrl` - Invalid gate `DerrInvCntSource` - Invalid source `DerrInvOutputCntrl` - Invalid output `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrSetLoad, daqCtrSetHold, daqCtrGetHold, daqCtrMultCtrl`
Program References	CTR1 (C Only)
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

daqCtrSetCtrMode is used to set the 9513's mode register for a specified counter. Setting this register defines how the specific counter works, including a wide variety of square wave and pulse generation and event counting. This function is usually followed by the **daqCtrSetLoad** or the **daqCtrSetHold** to set the initial counter values. Finally the **daqCtrMultCtrl** function is called to load an arm multiple counters to start. The **daqCtrMultCtrl** can also be used when counting events.

The gate control parameter (**gateCtrl**) controls how the counter will use its gate input (P3 pins 37, 18, 16, 14 and 12) or another counter's gate input. If the gate is disabled using the **DgcNoGating** definition, it will be ignored and the counter will run as long as it is armed. If a level gate control is selected (using the **DgcHighLevelGateNPI, DgcHighLevelGateNMI**, or **DgcHighLevelGateN** definitions), the counter will operate only while counter is armed and the selected high or low level is applied to the gate. If an edge-sensitive gate control is selected using the **DgcHighEdgeGate** or **DgcHighEdgeGateN** definitions, the counter will operate after a rising or falling edge is detected on the gate input. Most gate control modes select the selected gate (N), but the gate inputs of the previous (N-1) and next (N+1) counters can be used. For example, counter 3 could use the gate input of counter 2 by selecting gate N-1 or counter 4 by selecting gate N+1. (Counter 1 and counter 5 are considered adjacent when selecting gate input N+1 or N-1.) The final gate control mode allows the TC-toggled output (see the output control description) of the previous counter (N-1) to be the gate. The selected counter will operate only when the previous counter's TC-toggled output is high.

Count Edge (**cntEdge**) selects whether the counter will count when it receives a rising or falling edge on its count source (see the count source description).

Count Source (**cntSource**) selects the source used as input to the specified counter. The Count Edge selects whether the rising or falling edge of this source is counted. Count Source can be any one of the counter inputs, **Src1** to **Src5** (P3 pins 36, 19, 17, 15 or 13), any one of the counter gates, **Gate1** to **Gate5** (P3 pins 37, 18, 17, 16 or 14), an internal frequency, **F1** to **F5**, or the TC-toggled output (see the output control description) of the previous counter (N-1). The internal frequencies are divide-by-10 divisions of the onboard oscillator which is by default 1 MHz, but can be jumpered to 10 MHz. The sources F1 through F5 correspond to the frequencies 1 MHz, 100 kHz, 10 kHz, 1 kHz and 100 Hz. The TC-toggled output of the previous counter can be used as a source allowing counters to be cascaded without external connections.

Count Direction (**cntDir**) selects whether the counter will count up or down. The counter is normally configured for down counting when generating a pulse or square wave. The load register would be set to a positive value which will decrement to zero, defining the duration or width of the waveform. In event counting, the counter would initially be set to zero and configured to count up. The hold register in this case would then contain the number of events received.

Count Type (**cntType**) selects binary or BCD counting. Binary format accepts a 16-bit number ranging from 0 - 65,535. BCD (binary coded decimal) accepts four 8-bit numbers representing 0-10, back in 16-bits, ranging from 0-9999.

Output Control (**outputCtrl**) controls the state of the counter output (P3 pins 35, 34, 33, 32, 31). There are 2 inactive and 3 active output modes. If the output is inactive, it can either be driven low or it can be high impedance. The active modes are all associated with the terminal count (TC) which is the moment in time when the counter reaches 0. This can happen by counting up past 65535 in binary count mode or 9999 in BCD count mode, or counting down past 1. The output can be driven high during the TC and low otherwise, low during the TC and high otherwise, or toggle the output every time a TC occurs. The TC-toggled mode is used to generate variable duty-cycle square waves.

The Count Repeat (**cntRepeat**), Reload (**reload**) and Special Gate (**specGate**) parameters have complex relationships that define the operation of the counter. The count repeat flag enables/disables rearming the counter after TC occurs. Applications such as software-retriggerable 1-shots would disable the repeat flag so the 1-shot occurs only after the counter arm command is sent. Other applications such as rate generators, square waves and hardware-retriggerable 1-shots, would enable the count repeat so the counter runs until disarmed.

The Reload flag programs the counter to use the count value in the load and/or hold registers for counting. If the reload flag is disabled, the counter will use the contents of the load register only for counting. Enabling the reload flag will allow the counter to use the contents of either or both registers depending on the special gate flag. If the reload flag is enabled and the special gate is disabled, the counter will alternate between registers. This allows a variable duty-cycle output waveform depending on the relative values of the hold and load registers. If the reload flag is enabled and the special gate is enabled, the operation will depend on the gate control parameter. In this situation, an active gate control will allow hardware retriggering on the active-going edge, and an inactive gate control will configure the counter to use the hold register for counting if the counters gate is high, or the load register if the gate is low. Refer to the *Am9513A/AM9513 Technical Manual* for further reference.

The table summarizes the operating modes of the counter/timer.

Counter Mode Operating Summary																								
Counter Mode	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**
Special Gate (CM7)	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
Reload Source (CM6)	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1
Repetition (CM5)	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
Gate Control (CM15-CM-13); N=no gating; L=level; E=edge	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E	N	L	E
Count to TC once, then disarm	X	X	X											X	X									
Count to TC twice, then disarm							X	X	X										X					
Count to TC repeatedly without disarming				X	X	X				X	X	X					X	X				X		X
Gate input does not gate counter input	X			X			X			X									X			X		
Count only during active gate level		X			X			X			X			X			X							
Start count on active gate edge and stop count on next TC			X			X									X			X						X
Start count on active gate edge and stop count on second TC									X			X												
No hardware retriggering	X	X	X	X	X	X	X	X	X	X	X	X							X			X		X
Reload counter from Load Register on TC	X	X	X	X	X	X								X	X		X	X						X
Reload counter on each TC, alternating reload source between Load and Hold Registers							X	X	X	X	X	X												
Transfer Load Register into counter on each TC that gate is LOW, transfer Hold Register into counter on each TC that gate is HIGH																			X			X		
On active gate edge transfer counter into Hold Register and then reload counter from Load Register														X	X		X	X						
On active gate edge transfer counter into Hold Register, but counting continues																								X

daqCtrSetHold

DLL Function	`daqCtrSetHold(uchar ctrNum, uint ctrVal);`
C	`daqCtrSetHold(uchar ctrNum, uint ctrVal)`
QuickBASIC	`QBdaqCtrSetHold%(ctrNum%, ctrVal%)`
Turbo Pascal	`daqCtrSetHold(ctrNum:byte; ctrVal:word ):integer;`
Parameters	
`uchar ctrNum`	The counter number Valid values: 1 - 5
`uint ctrVal`	The value to write to the hold register of the selected counter Valid values: 0 - 65535
Returns	`DerrInvCtrNum` - Invalid channel `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrSetMasterMode, daqCtrSetCtrMode`
Program References	CTR1, CTR2, (All Languages)
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqCtrSetHold` outputs a value to the hold register of the specified counter. The hold register can be used to set the counter's initial value using the **`daqCtrMultCtrl`** function. The **`daqCtrSetMasterMode`** and **`daqCtrSetCtrMode`** functions describe various uses of the hold register.

daqCtrSetLoad

DLL Function	`daqCtrSetLoad(uchar ctrNum, uint ctrVal);`
C	`daqCtrSetLoad(int ctrNum, uint ctrVal)`
QuickBASIC	`QBdaqCtrSetLoad%(ctrNum%, ctrVal%)`
Turbo Pascal	`daqCtrSetLoad(ctrNum:byte; ctrVal:word ):integer;`
Parameters	
`uchar ctrNum`	The counter number Valid values: 1 - 5
`uint ctrVal`	The value to write to the load register of the selected counter Valid values: 0 - 65535
Returns	`DerrInvCtrNum` - Invalid channel `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrSetMasterMode, daqCtrSetCtrMode`
Program References	CTR1, CTR2, (All Languages)
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqCtrSetLoad` outputs a value to the load register of the specified counter. The load register can be used to set the counter's initial value using the **`daqCtrMultCtrl`** function. The **`daqCtrSetMasterMode`** and **`daqCtrSetCtrMode`** functions describe various uses of the load register.

daqCtrSetMasterMode

DLL Function	`daqCtrSetMasterMode(uchar foutDiv, uchar cntSource, uchar comp1, uchar comp2, uchar tod);`	
C	`daqCtrSetMasterMode(int foutDiv, int foutSource, int comp1, int comp2, int tod)`	
QuickBASIC	`QBdaqCtrSetMasterMode%(foutDiv%, foutSource%, comp1 %, comp2%, tod%)`	
Turbo Pascal	`daqCtrSetMasterMode(foutDiv:byte; cntSource: byte; comp1:byte; comp2:byte; tod:byte ):integer;`	
Parameters		
`uchar foutDiv`	The fout divider. A divider of 0 selects divide by 16 Valid values: 1 -16	
`uchar cntSource`	The fout source (see below)	
`uchar comp1`	A flag that if non-zero will enable the compare 1 operation, or if 0 will disable it	
`uchar comp2`	A flag that if non-zero will enable the compare 2 operation, or if 0 will disable it	
`uchar tod`	The time of day mode (see below)	
Count Source Definitions:		
Description	**Value**	**Description**
`DcsFOut Disabled`	`00h`	Fout set low
`DcsSrc1`	`01h`	Counter 1 input (pin 36 of P3)
`DcsSrc2`	`02h`	Counter 2 input (pin 19 of P3)
`DcsSrc3`	`03h`	Counter 3 input (pin 17 of P3)
`DcsSrc4`	`04h`	Counter 4 input (pin 15 of P3)
`DcsSrc5`	`05h`	Counter 5 input (pin 13 of P3)
`DcsGate1`	`06h`	Counter 1 gate (pin 37 of P3)
`DcsGate2`	`07h`	Counter 2 gate (pin 18 of P3)
`DcsGate3`	`08h`	Counter 3 gate (pin 16 of P3)
`DcsGate4`	`09h`	Counter 4 gate (pin 14 of P3)
`DcsGate5`	`0Ah`	Counter 5 gate (pin 12 of P3)
`DcsF1`	`0Bh`	Onboard 1 MHz clock
`DcsF2`	`0Ch`	Onboard 100 kHz clock
`DcsF3`	`0Dh`	Onboard 10 kHz clock
`DcsF4`	`0Eh`	Onboard 1 kHz clock
`DcsF5`	`0Fh`	Onboard 100 Hz clock
Time-Of-Day Definitions:		
Description	**Value**	
`DtodDisabled`	`00h`	
`DtodDivideBy5`	`01h`	
`DtodDivideBy6`	`02h`	
`DtodDivideBy10`	`03h`	
Returns	`DerrInvCntSource` - Invalid source `DerrInvTod` - Invalid time of day mode `DerrInvDir` - Invalid divisor `DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqCntSetLoad, daqCntMultCtr, daqCntGetHold, daqCntSetCntMode`	
Program References	CTR1 (C Only)	
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A	

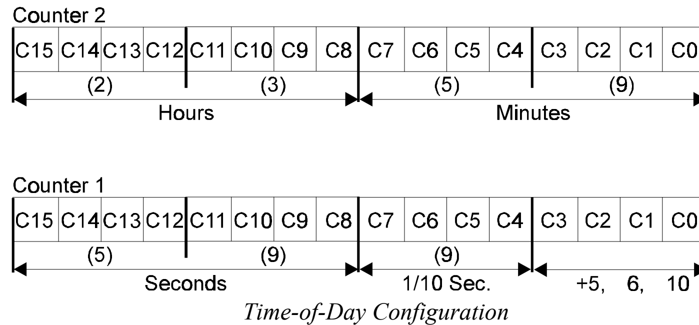
`daqCtrSetMasterMode` is used to set the counter's master mode register. This register is used to configure the fout pin (P3 pin 30), the comparators of counter 1 and 2 and the time-of-day operation of the 9513 chip. The master mode parameters default to zero after **`daqInit`**.

The fout source selects what signal will be output on the fout pin. The fout source can be any one of the counter inputs, **`Src1`** to **`Src5`** (P3 pins 36, 19, 17, 15 or 13), any one of the counter gates, **`Gate1`** to **`Gate5`** (P3 pins 37, 18, 17, 16 or 14) or an internal frequency, **`F1`** to **`F5`**, which are internal 1 MHz, 100 kHz, 10 kHz, 1 kHz and 100 Hz frequencies. The fout divider will divide the selected source by 1 to 16 before outputting the signal on fout.

The 2 comparator flags control the comparators associated with counter 1 and 2. If a comparator is enabled, the value in the corresponding alarm register, set with the **`daqCtrSetAlarm`** function, will be compared with the value in the counter. The output of the corresponding counter will go **`true`** when the value in the counter reaches the value in the alarm register and remain **`true`** until the counter value changes. The polarity of the output depends on the output control, set with the **`daqCtrSetCtrMode`** function, configuration of counter 1 or 2. When the output control is high,

terminal count pulsed or terminal count toggled, then the output will be high while the comparator is **`true`**. When the output control is low and terminal count pulsed, then the output will be low while the comparator is **`true`**.

The time-of-day parameter is used to enable or disable the time-of-day operation. The time-of-day operation is a special mode which causes counters 1 and 2 to turn over at counts that generate 24-hour time-of-day accumulations. The resolution of the time-of-day operation is 0.1 seconds. A 100 Hz, 60 Hz or 50 Hz signal must be applied to the input of counter 1 (P3 pin 36), while in the divide-by-10, divide-by-6 and divide-by-5 time-of-day modes respectively. This will produce the 10 Hz clock source needed to drive the time-of-day clock. The hold registers of counters 1 and 2 will hold the 24-hour time.



Time-of-Day Configuration

The following steps must be performed to use the time-of-day operation:

1. Set the master mode register as described above.
2. For general-purpose time keeping, configure counter 1 using **`daqCtrSetCtrMode`** with the no gating, count on rising edge, special gating disabled, reload from hold only, count repetitively, BCD counting and count up. The count source can be any of the available sources. The output control does not affect time-of-day operation.
3. Set the mode of counter 2 with the same settings as counter 1, except the count source should be TC toggled of the previous (N-1) counter. This allows internal concatenation of counter 1 to counter 2.
4. Set the load registers of counter 1 and 2 to zero using the **`daqCtrSetLoad`** function.
5. Initialize the current 24-hour time-of-day by setting the load registers of counters 1 and 2 using the format shown in the figure above, again using **`daqCtrSetLoad`**.
6. Repeat step 4.

daqCtrStopBack

DLL Function	`daqCtrStopBack(void);`
C	`daqCtrStopBack(void)`
QuickBASIC	`QBdaqCtrStopBack%()`
Turbo Pascal	`daqCtrStopBack:integer;`
Parameters	None
Returns	`DerrNotCapable` - No 9513 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqCtrRdNBack, daqCtrGetBackStat`
Program References	None
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

daqCtrStopBack stops a background operation initiated by the daqCtrRdNBack function.

daqDacWt

DLL Function	`daqDacWt(uchar chan, uint dataVal);`
C	`daqDacWt(int chan, int dataVal)`
QuickBASIC	`QBdaqDacWt%(chan%, dataVal%)`
Turbo Pascal	`daqDacWt(chan:byte; dataVal: word ):integer;`
Parameters	
`uchar chan`	The D/A channel to output to Valid values: 0 - 1
`uint dataVal`	The value to output to the selected D/A channel Valid values: 0 -4095
Returns	`DerrInvChan` - Invalid channel `DerrInvDacVal` - Invalid data value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDacWtBoth, daqAdcSetTrig`
Program References	DAC1 (All Languages)
Used With	**Does not apply to DaqPCMCIA models.**

`daqDacWt` outputs a voltage between 0 and 5 V to the specified 12-bit D/A channel. The voltage has a resolution of approximately 1.22 mV (5 V/4095).

Note: **`daqAdcSetTrig`** will configure the D/A channel 1 if an analog trigger is source selected for the A/D converter.

daqDacWtBoth

DLL Function	`daqDacWtBoth(uint chan0Val, uint chan1Val);`
C	`daqDacWtBoth(uint chan0Val, uint chan1Val);`
QuickBASIC	`QBdaqDacWtBoth%(chan1Val%, chan2Val%)` **[note: actually DAC channels 0 and 1]**
Turbo Pascal	`daqDacWtBoth(chan0Val:word; chan1Val:word ):integer;`
Parameters	
`uint chan0Val`	The value to output to the D/A channel 0 Valid values: 0 -4095
`uint chan1Val`	The value to output to the D/A channel 1 Valid values: 0 -4095
Returns	`DerrInvDacVal` - Invalid data value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDacWt, daqAdcSetTrig`
Program References	DAC1 (All Languages)
Used With	Does not apply to DaqPCMCIA models.

`daqDacWtBoth` outputs voltages between 0 and 5 V to both 12-bit D/A channels. Each voltage has a resolution of approximately 1.22 mV (5 V/4095).

Note: **`daqAdcSetTrig`** will configure the D/A channel 1 if an analog trigger source is selected for the A/D converter.

daqDacWtMany

DLL Function	`daqDacWtMany( unsigned int startChan, unsigned int _far *dataVals, unsigned char count );`
C	`daqDacWtMany( unsigned int startChan, unsigned int _far *dataVals, unsigned char count );`
Quick BASIC	`QBdaqDacWtMany%(startChan%, dataVals%(), count%)`
Turbo Pascal	`daqDacWtMany(startChan:integer; dataVals:WordP; count:integer):integer;`
Parameters	
`deviceTypes`	Specifies the DAC types
`chans`	Specifies the DAC channels
`dataVals`	The value to output to the D/A channel Valid values: `0 -4095`
`count`	
Returns	`DerrInvDacVal` - Invalid data value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDacWt`
Program References	DACEX1.C, DAQEX.FRM (VB)
Used With	DaqBoard100A, DaqBoard112A, DaqBoard200A, DaqBoard216A

`daqDacWtMany` outputs voltages between 0 and 5 V to all active 12-bit D/A channels. Each voltage has a resolution of approximately 1.22 mV (5 V/4095).

Note: **`daqAdcSetTrig`** will configure the D/A channel 1 if an analog trigger source is selected for the A/D converter.

daqDigConf

DLL Function	`daqDigConf(uchar chip, uchar config);`	
C	`daqDigConf(int port, int config)`	
QuickBASIC	`QBdaqDigConf%(port%, config%)`	
Turbo Pascal	`daqDigConf(chip:byte; config:byte ):integer;`	
Parameters		
`uchar chip`	The chip to configure (see below)	
`uchar config`	The configuration byte to write to the control register of the specified chip Valid values: 0 - 255	
Digital I/O Chip Definitions:		
Description	**Value**	**Address Select Jumper Location**
`DdcLocal`	`13h`	Local 8255
`DdcExp0`	`63h`	Address Select Location A
`DdcExp1`	`67h`	Address Select Location A
`DdcExp2`	`6Bh`	Address Select Location B
`DdcExp3`	`6Fh`	Address Select Location B
`DdcExp4`	`73h`	Address Select Location C
`DdcExp5`	`77h`	Address Select Location C
`DdcExp6`	`7Bh`	Address Select Location D
`DdcExp7`	`7Fh`	Address Select Location D
Returns	`DerrInvChip` - Invalid chip `DerrNotCapable` -No 8255 available `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqDigWtByte, daqDigRdByte, daqDigWtBit, daqDigRdBit`	
Program References	DIG1 (All Languages)	
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A	

daqDigConf configures the operation of the 3 8-bit ports of the selected 8255 digital I/O chip. This chip can be local or on a DBK20/21 digital I/O expansion card. **daqDigConf** configures ports A and B as inputs or outputs and port C for simple input/output or more complicated handshaking. **daqDigGetConf** can be used to generate a configuration byte for the basic input/output mode. This byte can then be passed to the **daqDigConf** function. In the basic input/output mode, port A and port B can be independently configured for input or output. Port C is divided into 2 4-bit nibbles that can be independently configured for input or output.

Note: For information on the strobed input/output and bi-directional bus modes of the 8255, reference the 8255 technical reference manual from Intel.

daqDigGetConf

DLL Function	`daqDigGetConf(uchar portA, uchar  portB, uchar portCHigh, uchar  portCLow, uchar *config)`
C	`daqDigGetConf(uchar portA, uchar portB, uchar portCHigh, uchar portClow, uchar *config)`
QuickBASIC	`QBdaqDigGetConf%(portA%, portB% , portCHigh%, portClow%, config%)`
Turbo Pascal	`daqDigGetConf(portA:byte; portB:byte; portCHigh:byte; portCLow:byte; config:DataP ) :integer;`
Parameters	
`uchar portA`	A flag that if non-zero will configure port A as input, otherwise will configure port A as output.
`uchar portB`	A flag that if non-zero will configure port B as input, otherwise will configure port B as output.
`uchar portCHigh`	A flag that if non-zero will configure the most significant 4 bits of port C as input, otherwise will configure them as output.
`uchar portCLow`	A flag that if non-zero will configure the least significant 4 bits of port C as input, otherwise will configure them as output.
`uchar *config`	A variable that will be returned with the configuration byte
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDigConf`
Program References	DIG1
Used With	Only applies to DaqBook/100/200 and DaqBoard/100A/200A

`daqDigGetConf` is used to generate a configuration byte that can be passed to the **`daqDigConf`** function to set the operation of the 3 8-bit ports of an 8255 chip. This byte is derived from 4 flags that configure port A, port B, the most significant 4-bits of port C and the least significant 4-bit of port C. Each of these ports can be independently selected as input or output.

daqDigRdBit

DLL Function	`daqDigRdBit(uchar port, uchar bitNum, uchar *bitVal);`
C	`daqDigRdBit(int port, int bitNum, int *bitVal)`
QuickBASIC	`QBdaqDigRdBit% (port%, bitNum%, bitVal%)`
Turbo Pascal	`daqDigRdBit(port:byte; bitNum:byte; bitVal:ByteP ) :integer;`
Parameters	
`uchar port`	The digital I/O port to read from
`uchar bitNum`	The bit number of the specified digital I/O port to read Valid values: 0 - 7 For 8-bit ports 0 - 3 For 4-bit ports
`uchar *bitVal`	A variable to hold the value of the specified bit (non-zero if asserted, 0 if unasserted)
Returns	`DerrInvBitNum` - Invalid bit number `DerrInvPort` - Invalid port `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDigConf, daqDigWtByte, daqDigRdByte, daqDigWtBit`
Program References	DIG1 (All Languages)
Used With	

`daqDigRdBit` reads the state of a single bit on a digital I/O port. The port read from can be the local 8255 chip, an 8255 chip on a DBK20 or DBK21 digital expansion board, or the P1 digital I/O nibble. When the port is on an 8255 chip, this function can read port A, port B, or port C or the most significant or least significant 4-bit nibble of port C.

Note: The DaqBook/112 can only read digital nibble in P1.

daqDigRdByte

DLL Function	`daqDigRdByte(uchar port, uchar *byteVal);`
C	`daqDigRdByte(int port, int *byteVal)`
QuickBASIC	`QBdaqDigRdByte%(port%,byteVal%)`
Turbo Pascal	`daqDigRdByte(port:byte; byteVal:DataP ):integer;`
Parameters	
`uchar port`	The digital I/O port to read from
`uchar *byteVal`	A variable to hold the value read from the specified port Valid values: 0 -255 for 8-bit ports 0-15 for 4-bit ports
Returns	`DerrInvPort` - Invalid port `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDigConf, daqDigWtByte, daqDigWtBit, daqDigRdBit`
Program References	DIG1 (All Languages)
Used With	

`daqDigRdByte` reads an 8-bit byte or a 4-bit nibble from a digital I/O port. The port read from can be the local 8255 chip, an 8255 chip on a DBK20 digital expansion board, or the P1 digital I/O nibble. When the port is on an 8255 chip, this function can read port A, port B, or port C or the most significant or least significant 4-bit nibble of port C.

Note: The DaqBook/112 can only read digital nibble in P1.

daqDigWtBit

DLL Function	`daqDigWtBit(uchar port, uchar bitNum, uchar bitVal);`
C	`daqDigWtBit(int port, int BitNum, int bitVal)`
QuickBASIC	`QBdaqDigWtBit%(port%, bit Num%, bitVal%)`
Turbo Pascal	`daqDigWtBit(port:byte; bitNum:byte; bitVal:byte ) :integer;`
Parameters	
`uchar`	The digital I/O port to write to
`uchar bitNum`	The bit number of the specified digital I/O port to assert/unassert Valid values: 0 - 7 For 8-bit ports 0 - 3 For 4-bit ports
`uchar bitVal`	A flag that if non-zero will assert the specified bit, if 0 the bit is unasserted
Returns	`DerrInvBitNum` - Invalid bit number `DerrInvPort` - Invalid port `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDigConf, daqDigWtByte, daqDigRdByte, daqDigRdBit`
Program References	DIG1 (All Languages)
Used With	

`daqDigWtBit` sets or clears a single bit on a digital I/O port. The port written to can be the local 8255 chip, an 8255 chip on a DBK20 or DBK21 digital expansion board, or the P1 digital I/O nibble. When the port is on an 8255 chip, this function can write to port A, port B, or port C or the most significant or least significant 4-bit nibble of port C.

Note: The DaqBook/112 can only read digital nibble in P1.

daqDigWtByte

DLL Function	`daqDigWtByte(uchar port, uchar byteVal);`
C	`daqDigWtByte(int port, int byteVal)`
QuickBASIC	`QBdaqDigWtByte%(port%, byteVal%)`
Turbo Pascal	`daqDigWtByte(port:byte; byteVal:byte ):integer;`
Parameters	
`uchar port`	The digital I/O port to write to
`uchar byteVal`	The value to write to the specified port Valid values: 0 - 255 for 8-bit ports 0-15 for 4-bit ports
Returns	`DerrInvPort` - Invalid port `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqDigConf, daqDigRdByte, daqDigWtBit, daqDigRdBit`
Program References	DIG1 (All Languages)
Used With	

`daqDigWtByte` writes to an 8-bit byte or a 4-bit nibble from a digital I/O port. The port written to can be the local 8255 chip, an 8255 chip on a DBK20 or DBK21 digital expansion board, or the P1 digital I/O nibble. When the port is on an 8255 chip, this function can write to port A, port B, or port C or the most significant or least significant 4-bit nibble of port C.

Note: The DaqBook/112 can only read digital nibble in P1.

daqGetProtocol

DLL Function	`daqGetProtocol(int *protocol)`	
C	`daqGetProtocol(int *protocol)`	
QuickBASIC	`QBdaqGetProtocol(protocol%)`	
Turbo Pascal	`daqGetProtocol(protocol):integer;`	
Parameters		
`protocol`	A pointer to a value that will be set to the current protocol chosen from the protocol codes listed below.	
Protocol Codes:		
Name	**Description**	**Value**
`DaqProtocol8`	8-bit I/O	1
`DaqProtocol4`	4-bit I/O	2
`DaqProtocolFPort`	Far Point F/Port EPP Interface	10
`DaqProtocolSL`	82360 SL EPP Interface	20
`DaqProtocolISA`	ISA Bus Interface (DaqBoard Only)	100
`DaqProtocolEPPBIOS`	EPP BIOS (Draft Revision 3)	40
`DaqProtocolSMC666`	Quatech SMC666 EPP Interface	30
Note: Additional EPP implementation codes may be described in the README file		
Returns	An error number, or 0 is no error (also, refer to *API Error Codes* on page 5-68)	
See Also	`daqInit, daqSetProtocol`	
Program References	None	
Used With		

`daqGetProtocol` returns the current parallel port communications protocol. **`daqInit`** initially sets the protocol to either **`DaqProtocol8`** or **`DaqProtocol4`**, indicating either 8-bit or 4-bit standard parallel port protocol. **`daqSetProtocol`** may be used to specify other protocols.

daqInit

DLL Function	`daqInit(uint lptPort, uchar lptIntr);`		
C	`daqInit(uint lptPort, uchar intr)`		
QuickBASIC	`QBdaqInit%(lptPort%, intr%)`		
Turbo Pascal	`daqInit(lptPort:integer; lptIntr:byte ):integer;`		
DaqBook Parameters			
`uint lptPort`	The LPT port number (see below)		
`uchar lptIntr`	The interrupt level		
DaqBook LPT Ports:			
Description	**Value**		
`LPT1`	`00h`		
`LPT2`	`01h`		
`LPT3`	`02h`		
`LPT4`	`03h`		
DaqBoard Parameters			
`uint lptPort`	The ISA bus address (see below)		
`uchar lptIntr`	The lower nibble (4 least significant bits) contains the interrupt level (10-15). The upper nibble (4 most significant bits) contains the DMA channel (see below for defined constants) Adding the interrupt level to the DMA constant will pack these two parameters into one byte. (See sample code.)		
DaqBoard Ports:		**DaqBoard DMA Channels:**	
Description	**Value**	**Description**	**Value**
`PORT_0300`	`300h`	`DMANone`	`00h`
`PORT_0304`	`304h`	`DMA5`	`50h`
`...`	`...`	`DMA6`	`60h`
`PORT_031F`	`31Fh`	`DMA7`	`70h`
Daq PCMCIA Parameters			
`uint lptPort`	The ISA bus address at which the card is configured (see below)		
`uchar lptIntr`	The interrupt level at which the card is configured.		
Daq PCMCIA Ports:			
Description	**Value**		
`PORT_0300`	`300h`		
`PORT_0304`	`304h`		
`...`	`...`		
`PORT_031F`	`31Fh`		
Sample code (DaqBoard Only)	`daqInit(PORT_0330, DMA5 + 10)`		
Returns	`DerrNotOnLine` - No communication with DaqBook/DaqBoard `DerrBadChannel` - Invalid LPT channel or ISA bus address `DerrNoDaqBook` - No DaqBook/DaqBoard detected `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)		
See Also	`daqSelectPort, daqClose`		
Program References	INIT, ADC1, ADC2, ADC3, ADC4, ADC5, CTR1, CTR2, DAC1, DAC2, DAC3, DIG1 (All Languages)		
Used With			

daqInit is used to perform multiple functions: initialize subroutine library variables, establish communications with a Daq*, reset the Daq* hardware to power-on conditions, and select the current Daq*. **daqInit** can be called to reinitialize the Daq* only after the **daqClose** command is called to terminate communications.

daqInit will perform the following tasks:

- Stop any current acquisition
- Set the scan group to channel 1 with a gain of ×1
- Set the pacer clock to 100 kHz
- Enable tagging of A/D data
- Set the D/A converter to 0 V (**Note**: does not apply to Daq PCMCIA)
- Configure all digital I/O as inputs
- Reset the counter/timers

Note: **daqInit** must be called before any other **daq*** function.

daqLinearConvert

DLL Function	`daqLinearConvert( unsigned *counts, unsigned scans, float fValues, unsigned nValues)`
C	`daqLinearConvert(unsigned _far *counts,` `unsigned scans, float _far *fValues, unsigned nValues);`
QuickBASIC	`QBdaqLinearConvert%(counts%(), scans%, fValues!(), nValues%)`
Turbo Pascal	`daqLinearConvert( counts:WordP; scans:word; fValues:SingleP; nValues:word) : integer;`
Parameters	
`*counts`	The acquired ADC readings to be converted.
`scans`	The number of scans to be converted.
`*fValues`	An array to hold the converted readings.
`nValues`	The size of the reading array.
Returns	`DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
Used With	

`daqLinearConvert` converts the ADC readings into floating point numbers using the linear relationship that was specified with **`daqLinearSetup`**. **`daqLinearConvert`** may be invoked repeatedly to perform multiple conversions, each using the same linear relationship.

daqLinearSetup

DLL Function	`daqLinearSetup(unsigned nscan,unsigned readingsPos,unsigned nReadings,float ADmin, float ADmax, float signal1, float voltage1, float signal2, float voltage2, unsigned avg)`

C	`daqLinearSetup(unsigned nscan,`
	`unsigned readingsPos, unsigned nReadings, float ADmin, float ADmax,`
	`float signal1, float voltage1, float signal2, float voltage2, unsigned avg);`

QuickBASIC	`QBdaqLinearSetup%(nscan%, readingsPos%, nReadings%, ADmin!, ADmax!, signal1!, voltage1!, signal2!, voltage2!, avg%)`
Turbo Pascal	`daqLinearSetup( nscan:word; readingsPos:word; nReadings:word; ADmin:single; ADmax:single;` `signal1:single; voltage1:single; signal2:single; voltage2:single; avg:word) : integer;`
Parameters	
`nscan`	The number of readings in a single scan (1 to 512).
`readingsPos`	The position within the scan of the first reading to convert (0 to nscan - 1).
`nReadings`	The number of consecutive ADC readings to convert (1 to nscan - readingPos)
`ADmin, ADmax`	The input voltages that correspond to the minimum and maximum possible A/D readings.
`signal1, signal2`	The transducer input signals that produce voltage1 and voltage2.
`voltage1, voltage2`	The transducer output voltages for two different input signals.
`avg`	The type of averaging to use. 0 = block averaging, 1 = no averaging, 2 or greater = moving average. “0” specifies block averaging in which all of the scans are averaged together to compute a single value for each channel. “1” specifies no averaging. Each scan’s readings are converted into measured signals. “2” (or more) specifies moving average of the specified number of scans. Each scan’s readings are averaged with the avg-1 preceding scans’ readings before conversion. The first scan is not averaged because there is not enough data. For example, if avg is “3”, then the results from the first scan are not averaged at all; the results from the second scan are averaged with the first scan; the results from the third and subsequent scans are averaged with the preceding two scans as shown in the next table.
Returns	`DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqLinearConvert, daqLinearSetupConvert`
Program References	
Used With	

`daqLinearSetup` saves the data required for **`daqLinearConvert`** to perform conversions. Six parameters are used to specify a linear relationship: the A/D input range (minimum and maximum values), and the transducer input signal level and output voltage at 2 points in the range.

	Readings from Channel		Results from Channel	
Scan	0	1	0	1
1	1A	2A	1A	2A
2	1B	2B	(1A+1B)/2	(2A+2B)/2
3	1C	2C	(1A+1B+1C)/3	(2A+2B+2C)/3
4	1D	2D	(1B+1C+1D)/3	(2B+2C+2D)/3
5	1E	2E	(1C+1D+1E)/3	(2C+2D+2E)/3
6	1F	2F	(1D+1E+1F)/3	(2D+2E+2F)/3

daqLinearSetupConvert

DLL Function	`daqLinearSetupConvert(unsigned nscan, unsigned readingsPos, unsigned nReadings, float ADmin, float ADmax, float signal1, float voltage1, float signal2, float voltage2, unsigned avg, unsigned _far *counts, unsigned scans, float _far *fValues, unsigned nValues)`
C	`daqLinearSetupConvert(unsigned nscan, unsigned readingsPos, unsigned nReadings, float ADmin, float ADmax, float signal1, float voltage1, float signal2, float voltage2, unsigned avg, unsigned _far *counts, unsigned scans, float _far *fValues, unsigned nValues);`
QuickBASIC	`QBdaqLinearSetupConvert%(nscan%, readingsPos%, nReadings%, ADmin!, ADmax!, signal1!, voltage1!, signal2!, voltage2!, avg%, counts%(), scans%, fValues!(), nValues%)`
Turbo Pascal	`daqLinearSetupConvert( nscan:word; readingsPos:word; nReadings:word; ADmin:single; ADmax:single; signal1:single; voltage1:single; signal2:single; voltage2:single; avg:word; counts:WordP; scans:word; fValues:SingleP; nValues:word) : integer;`
Parameters	See **daqLinearSetup** and **daqLinearConvert** for a description of the parameters.
Returns	`DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
Used With	

`daqLinearSetupConvert` combines the setup and conversion processes into one function.

daqReadCalFile

DLL Function	`daqReadCalFile(char *calfile);`
C	`daqReadCalFile(char *calfile)`
QuickBASIC	`QBdaqReadCalFile%(calfile%)`
Turbo Pascal	`daqReadCalFile(calfile):integer;`
Parameters	
`char *calfile`	The file name with optional path information of the calibration file. If calfile is NULL or empty (“”), the default calibration file DAQBOOK.CAL will be read.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68) `DerrInvCalfile` - Error occurred while opening or reading calibration file
See Also	`daqCalSetup, daqCalConvert, daqCalSetupConvert`
Program References	None
Used With	

`daqReadCalFile` is the initialization function for reading in the calibration constants from the calibration text file.

This function (usually called once at the beginning of a program) will read all the calibration constants from the specified file. If calibration constants for a specific channel number and gain setting are not contained in the file, ideal calibration constants will be used (essentially performing no calibration for that channel). If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and **`daqReadCalFile`** will return a non-zero error code.

daqRTDConvert

DLL Function	`daqRtdConvert (uint *counts, uint  scans ,int *temp, uint ntemp);`
C	`daqRtdConvert(unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);`
QuickBASIC	`QBdaqRtdConvert% (counts%(), scans%, temp%(), ntemp%)`
Turbo Pascal	`daqRtdConvert(var counts; scans:word; var temp; ntemp:word) : integer;`
Parameters	
`uint *counts`	Raw A/D data from one or more scans
`uint scans`	Number of scans of raw data in counts
`int * temp`	Variable array to hold converted temperatures
`uint ntemp`	Size of temperature array (should be number of RTDs specified in setup times the number of scans)
Returns	`DerrRtdNoSetup` - Setup was not called `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68) `DerrRtdTArraySize` - Temperature array is not large enough
See Also	`DaqRtdSetup, DaqRtdSetupConvert`
Program References	None
Used With	

daqRTDConvert takes raw A/D readings from RTDs and converts them to temperature readings in tenths of degrees Celsius. **Note**: Total number of conversions (scan * (RTD chans per scan) * 4) must be less than 32K.

The Daq* measures temperatures sensed by RTDs attached via a DBK9 RTD expansion card. Up to 8 RTDs can attach to each DBK9. Up to 32 DBK9s may be attached to a single Daq* for a maximum of 256 temperatures. The software currently supports 100-, 500-, and 1000-ohm RTDs.

The RTD measurement functions are designed for simple temperature measurement in which each RTD channel is read 4 times. These 4 readings must be grouped together in the scan and in order: **`Dbk9VoltageA`** (gain=0), **`Dbk9VoltageB`** (gain=1), **`Dbk9VoltageD`** (gain=3), **`Dbk9VoltageD`** (gain=3). The RTDs must be of the same type, and the reading groups must follow each other in the scan sequence.

The temperature conversion functions use input data from one or more Daq* scans. They take 4 voltage readings for each RTD channel, apply the appropriate averaging method, convert the voltages to a resistance and then, using the appropriate curves for the RTD type, convert the resistance into a temperature. For example, assume the following readings:

	Readings Channel 0				Readings Channel 1			
Scan	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
2	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
3	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
4	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd
5	Ch 0 Va	Ch 0 Vb	Ch 0 Vd	Ch 0 Vd	Ch 1 Va	Ch 1 Vb	Ch 1 Vd	Ch 1 Vd

The 4 readings for each channel are grouped together in order. If this scan data is passed to **`daqRtdConvert`** (through the counts parameter) with averaging disabled (**`avg`** parameter set to 1), the function will return the temp parameters shown in the table. **Note**: Temperatures returned will be in tenths of a degree Celsius.

Temperatures		
Scan	**0**	**1**
1	Ch 0 °C	Ch 1 °C
2	Ch 0 °C	Ch 1 °C
3	Ch 0 °C	Ch 1 °C
4	Ch 0 °C	Ch 1 °C
5	Ch 0 °C	Ch 1 °C

If the scan data is passed to **`daqRtdConvert`** (in the counts parameter) with averaging set to block averaging (**`avg`** parameter set to 0) the function will return the temp parameters shown in the table.

Temperatures		
	0	**1**
Average of all Temperatures	Ch 0 °C	Ch 1 °C

The conversion process has 2 steps: setup and conversion. Setup describes the characteristics of the temperature measurement; and Conversion changes raw readings into temperatures. For convenience, both setup and conversion can be performed at once by **`daqRtdSetupConvert`**. All of the functions return error codes which are defined in DaqBook.h.

daqRtdSetup

DLL Function	`daqRtdSetup(unsigned nScan, unsigned startPosition, unsigned nRtd, unsigned rtdValue, unsigned avg)`
C	`daqRtdSetup(unsigned nScan, unsigned startPosition, unsigned nRtd, unsigned rtdValue, unsigned avg);`
QuickBASIC	`QBdaqRtdSetup% (nscan%, startPosition%, nRtd%, rtdValue%, avg%)`
Turbo Pascal	`daqRtdSetup(nScan, startPosition, nRtd, rtdValue, avg:word) : integer;`
Parameters	
`uint nReadings`	The total number of readings in a scan. valid range 1-512
`uint startPosition`	Position of the first RTD reading group in the scan. Valid range 1-509
`int nRtd`	Number of RTD reading groups in the scan. Valid range 1- 128
`uint rtdValue`	Value of RTD being used. `Dbk9RtdType100`- 100 ohm RTD `Dbk9RtdType500`- 500 ohm RTD `Dbk9RtdType1K`- 1000 ohm RTD
`uint avg`	Type of averaging to be used. 0 = block averaging 1 = no averaging 2 to (number of scans -1) = moving average
Returns	`DerrRtdParam` - Setup parameter out of range `DerrRtdValue` - Invalid RTD type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	

`daqRtdSetup` sets up parameters for subsequent RTD temperature conversions. Refer to discussion of daqRTDConvert.

daqRtdSetupConvert

DLL Function	`daqRtdSetupConvert(unsigned nScan, unsigned startPositionunsigned nRtdunsigned rtdValue, unsigned avg, unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp)`
C	`daqRtdSetupConvert(unsigned nScan, unsigned startPosition, unsigned nRtd, unsigned rtdValue, unsigned avg, unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);`
QuickBASIC	`QBdaqRtdSetupConvert% (nscan%, startPosition%, nRtd%, rtdValue%, avg%, counts%(), scans%, temp%(), ntemp%)`
Turbo Pascal	`daqRtdSetupConvert(nScan, startPosition, nRtd, rtdValue, avg:word var counts; scans:word; var temp; ntemp:word) : integer;`
Parameters	
`uint nReadings`	The total number of readings in a scan. valid range 1-512
`uint startPosition`	Position of the first RTD reading group in the scan. Valid range 1-509
`uint nRtd`	Number of RTD reading groups in the scan. Valid range 1-128
`uint rtdValue`	Value of RTD being used. `Dbk9RtdType100`- 100 ohm RTD `Dbk9RtdType500`- 500 ohm RTD `Dbk9RtdType1K`- 1000 ohm RTD
`uint avg`	Type of averaging to be used 0 = block averaging 1 = no averaging 2 to (number of scans -1) = moving average
`uint *counts`	Raw A/D data readings from one or more scans.
`uint scans`	Number of scans of raw data in contained in *counts.
`int *temp`	Array to hold converted temperatures.
`uint nTemp`	Size of temperature array. Should be the number of RTDs times the number of scans for no averaging and moving averages or the number of RTDs for block averaging.
Returns	`DerrRtdParam` - Setup parameter out of range `DerrRtdValue` - Invalid RTD type `DerrRtdTArraySize` - temperature storage array not large enough `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	

`daqRtdSetupConvert` sets up and converts raw A/D readings from RTDs into temperature readings. Refer to discussion of **`daqRTDConvert`**.

daqSelectPort

DLL Function	`daqSelectPort(uint lptPort);`
C	`daqSelectPort(uint lptPort)`
QuickBASIC	`QBdaqSelectPort%(lptPort%)`
Turbo Pascal	`daqSelectPort(lptPort:integer) :integer;`
Parameters	
uint lptPort	The LPT port number or ISA bus address (see below)
DaqBook LPT Ports:	
Description	**Value**
`LPT1`	`00h`
`LPT2`	`01h`
`LPT3`	`02h`
`LPT4`	`03h`
DaqBoard Ports:	
Description	**Value**
`PORT_0300`	`300h`
`PORT_0304`	`304h`
`...`	`...`
`PORT_031F`	`31Fh`
Returns	`DerrNotOnLine` - No communications with DaqBook/DaqBoard `DerrBadChannel` - Invalid LPT channel `DerrNoDaqBook` - No DaqBook/DaqBoard detected `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqInit`
Program References	None
Used With	

daqSelectPort selects an initialized Daq*. This function causes any subsequent function calls to be performed on this Daq*. Because **daqInit** initializes then selects a Daq*, **daqSelectPort** is only needed when using multiple Daq*s.

Note: **daqInit** must be called with the corresponding LPT port before **daqSelectPort** can select it.

daqSetErrHandler

DLL Function	`daqSetErrHandler(daqErrorHandlerFPT daqErrorHandler);`
C	`daqSetErrHandler(daqErrorHandlerFPT daqErrorHandler);`
QuickBASIC	`QBdaqSetErrHandler%(errHandler%)`
Turbo Pascal	`daqSetErrHandler( daqErrorHandler:ErrorFuncT ):integer;`
Parameters	
`daqErrorHandlerFPT daqErrorHandler`	The routine to call when an error occurs, or null (0) to have nothing called when an error occurs.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	All
Used With	

`daqSetErrHandler` specifies the routine to call when an error occurs in any command. The default routine displays a message, then terminates the program. If this is not desirable, use this command to specify your own routine to be called when errors occur. If you want no action to occur when a command error is detected, use this command with a null (0) parameter.

The default error routine is **`daqDefaultHandler`**. Use that as the command parameter to restore default error processing. This command may be called at any time, even before **`daqInit`**.

daqSetProtocol

DLL Function	`daqSetProtocol( int protocol)`	
C	`daqSetProtocol(int protocol)`	
QuickBASIC	`QBdaqSetProtocol(protocol%)`	
Turbo Pascal	`daqSetProtocol(protocol):integer;`	
Parameter		
`protocol`	One of the predefined protocol codes listed below.	
Protocol Codes		
Name	**Description**	**Value**
`DaqProtocol8`	8-bit I/O	1
`DaqProtocol4`	4-bit I/O	2
`DaqProtocolFPort`	Far Point F/Port EPP Interface	10
`DaqProtocolSL`	82360 SL EPP Interface	20
`DaqProtocolISA`	ISA Bus Interface (DaqBoard Only)	100
`DaqProtocolEPPBIOS`	EPP BIOS (Draft Revision 3)	40
`DaqProtocolSMC666`	Quatech SMC666 EPP Interface	30
Additional protocol codes may be described in the README file.		
Returns	An error number, or 0 if no error (also, refer to *API Error Codes* on page 5-68)	
Related Functions	`daqInit, daqGetProtocol`	
Used With	Only applies to DaqBook; does not apply to DaqBoard or DaqPCMCIA.	

`daqSetProtocol` specifies to the DaqBook/DaqBoard driver the type of parallel-port implementation and protocol that is available on the computer. The driver then attempts to configure the computer and the DaqBook/DaqBoard to communicate using the specified protocol. As establishing the protocol may affect the settings of the DaqBook/DaqBoard, **`daqSetProtocol`** should only be invoked immediately after **`daqInit`** has established communications with and reset the DaqBook. Switching protocols during normal operation is not recommended.

Two types of parallel port implementations are supported by the DaqBook: standard and enhanced. Standard parallel ports, using the DaqBook's proprietary protocols, are capable of receiving data either 4 or 8 bits at a time. When possible, the faster 8-bit operation is preferred, but not all standard parallel ports support 8-bit data reception.

Enhanced parallel ports (EPP) include extra hardware that increases the rate of data transfer to 3 to 10 times the rate of a standard parallel port. Unfortunately, not every computer includes EPP capability and attempting to use EPP on an incompatible computer may cause the driver to access I/O locations which are not part of the printer port interface. Such accesses may interfere with other operations and cause the computer to operate incorrectly. For this reason, EPP operation must be explicitly requested by the program.

When the DaqBook is initialized by **`daqInit`**, it is initially configured for a standard parallel port protocol: either 8-bit, if possible, or the slower 4-bit protocol. After **`daqInit`** has completed, **`daqSetProtocol`** may be used to switch to any other supported protocols as listed below.

If **`daqSetProtocol`** is unable to establish communications using the specified protocol, then it will try to establish communications using the standard port protocols, first 8-bit, then the slower 4-bit. In such an event, **`daqSetProtocol`** will not return an error indication unless it is unable to establish any protocol.

In any case, **`daqGetProtocol`** may be used to check the current operating protocol.

daqTCConvert

DLL Function	`daqTCConvert (uint *counts, uint  scans ,int *temp, uint ntemp);`
C	`daqTCConvert(uint *counts, uint scans, int *temp, uint ntemp)`
QuickBASIC	`QBdaqTCConvert%(counts%(), scans%, temp%(), ntemp%)`
Turbo Pascal	`daqTCConvert(var counts; scans: word; var temp; ntemp: word):integer;`
Parameters	
`uint *counts`	An array of one or more scans of raw data as received from the Daq. The ADC data bits are in the 12 most significant bits of the 16-bit integers, and the tag bits (which are discarded) are in the 4 least-significant bits. Channel tagging must be enabled using the `daqAdcSetTag` command. Valid range: Each raw data item may be any 16-bit value.
`uint scans`	The number of scans of data in counts. Valid range: 1 to 32768/nscan (counts is limited to 64 Kbytes).
`int * temp`	Variable array to hold converted temperature results. The integer values are 10 times the temperatures in °C. For example, 50°C would be represented as 500 and -10°C would be -100. Valid range: Results range from -2000 (-200°C) to +13720 (+1372°C) depending on the thermocouple type.
`uint ntemp`	The number of entries in the temperature array. This value is checked by the functions to avoid writing past the end of the array. Valid range: If avg is 0, then ntc or greater. If avg is non-zero, then scans * ntc or greater.
Returns	`DerrTCE_NOSETUP` - Setup was not called `DerrTCE_PARAM` - Param out of range `DerrNoError` -No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`DaqTCSetup, DaqTCSetupConvert`
Program References	None
Used With	

`daqTCConvert` takes raw A/D readings and converts them to temperature readings in tenths of degrees Celsius (the total number of conversions (scan * chans/scan) must be less than 32K). The Daq* measures thermocouple temperatures by way of a DBK19 or DBK52 that includes a cold-junction compensation circuit (CJC) attached to channel 0. Channel 1 is shorted for performing auto-zero compensation. Channels 2 through 15 accept thermocouples for temperature measurement. Up to 16 expansion cards may be attached to a single Daq* to measure a maximum of 224 (16×14) temperatures. The software supports type J, K, T, E, N28, N14, S, R and B thermocouples.

Two software techniques (calibration and zero compensation) can be used to increase the accuracy of the DBK19 card.

- Software calibration uses gain and offset calibration constants, unique to each card, to compensate for inherent errors on the card.
- Zero compensation is a method by which any offset voltage on the card can be removed at run-time. This is done by measuring a shorted channel at the same gain on the actual input to find the offset and by subtracting this value from the actual reading.

The thermocouple linearization function has a special auto-zero compensation feature that will perform zero compensation on the raw thermocouple data before linearizing when using a DBK19. The auto-zero feature is enabled by default, but can be disabled using the **`daqZeroDbk19`** function. It is not available when using unipolar mode.

The temperature measurement conversion functions are designed for temperature measurement where:

- The cold-junction compensation circuit (CJC) channel (channel 0) reading from the T/C card is immediately followed in the scan sequence by the T/C channel readings, all of which must be from the same type of T/C (including: J, K, T, E, N28, N14, S, R, or B).
- If a DBK19 is used with auto-zeroing enabled, the CJC channel reading described above must be preceded by a reading from the shorted channel (channel 1). This reading must be at the same gain setting as the CJC reading and a reading from the shorted channel (channel 1) at the same gain setting as the T/C to be converted.
- If software calibration is used with the DBK19, the calibration constants for the card to be used should be entered into the calibration file.
- The CJC and T/C readings are taken with the optimal gains (as described below).
- All non-thermocouple data conversion, if any, must be done by other means.

The temperature conversion functions take input data from one or more scans from the Daq*. They then examine the CJC and thermocouple readings within that scan and, after optional averaging, convert them to temperatures which are stored as output. For example, see readings in the table.

Scan	Reading					
	0	1	2	3	4	5
1	V or CJC Zero	V or J Zero	CJC	J1a	J1b	J1c
2	V or CJC Zero	V or J Zero	CJC	J2a	J2b	J2c
3	V or CJC Zero	V or J Zero	CJC	J3a	J3b	J3c
4	V or CJC Zero	V or J Zero	CJC	J4a	J4b	J4c

The first 2 readings of each scan are non-temperature voltage readings to compensate for the CJC circuit and the shorted channel 0. The third reading is from the CJC, and the remaining 3 readings are from 3 type J thermocouples. If the auto-zero feature is disabled, the first 2 readings will be ignored. Otherwise, the first 2 readings will be used to remove offset errors in the CJC and T/C reading. The CJC and T/C readings are used to produce one temperature result for each T/C reading. Thus, the 24 original readings are reduced to 12 temperatures.

The conversion process has 2 steps: setup and conversion. Setup describes the characteristics of the temperature measurement, and Conversion changes the raw readings into temperatures. All of the functions return error codes which are defined in DaqBook.h which also includes the function prototypes and the definitions of the thermocouple type codes.

To measure temperatures, the scan must be set up so the T/C measurements consecutively follow their corresponding CJC measurement (the CJC measurement need not be the first element in the scan). If auto-zeroing is enabled, the CJC measurement must be preceded by both a CJC zero measurement and a T/C zero measurement.

All of the thermocouples converted with a single invocation of the conversion functions must be of the same type: J, K, T, E, N28, N14, S, R, or B. To measure with more than one type of thermocouple, they must be sorted by type within the scan, and each type must be preceded by the related CJC.

The scan is not restricted to thermocouple measurements. The scan may include other types of signals such as voltage, current, or digital input, but conversion of these readings is up to you. The temperature conversion functions cannot handle them.

GAIN CODES				
Type	Unipolar Gain Code	Unipolar Gain	Bipolar Gain Code	Bipolar Gain
CJC	`Dbk19UniCJC`	90	`Dbk19BiCJC`	60
J	`Dbk19UniTypeJ`	180	`Dbk19BiTypeJ`	90
K	`Dbk19UniTypeK`	180	`Dbk19BiTypeK`	90
T	`Dbk19UniTypeT`	240	`Dbk19BiTypeT`	180
E	`Dbk19UniTypeE`	90	`Dbk19BiTypeE`	60
N28	`Dbk19UniTypeN28`	240	`Dbk19BiTypeN28`	240
N14	`Dbk19UniTypeN14`	180	`Dbk19BiTypeN14`	90
S	`Dbk19UniTypeS`	240	`Dbk19BiTypeS`	240
R	`Dbk19UniTypeR`	180	`Dbk19BiTypeR`	240
B	`Dbk19UniTypeB`	240	`Dbk19BiTypeB`	240

The temperature measurements must be made with the correct gain settings. The gain settings for the different thermocouple types depend on the channel type and the bipolar/unipolar setting of the Daq* as specified in the table. **Note**: Unipolar operations are not recommended for thermocouple measurement unless the measured temperatures will be greater than the Daq* temperature.

When measuring thermocouples using the gains above, the following temperature ranges apply.

Thermocouple mV Outputs For Temperature Ranges Depending on Ambient Temperature						
T/C Type	Measured Temperature Range @ 0°C ambient		Measured Temperature Range @ 25°C ambient		Measured Temperature Range @ 50°C ambient	
	Temperature °C	0°C Output (mV)	Temperature°C	25°C Output (mV)	Temperature°C	50°C Output (mV)
J	-200 to 760	-7.9 to 42.9	-200 to 760	-9.2 to 41.6	-200 to 760	-11.8 to 39.0
K	-200 to 1372	-5.9 to 54.9	-200 to 1372	-6.9 to 53.9	-200 to 1372	-8.9 to 52.9 (50.0
T	-200 to 400	-5.6 to 20.9	-200 to 400	-6.6 to 19.9	-200 to 400	-8.7 to 17.7
E	-270 to 1000	-9.8 to 76.4	-270 to 1000	-11.3 to 74.9	-270 to 1000	-14.5 to 71.7
N28	-270 to 400	-4.3 to 13.0	-270 to 400	-5.0 to 12.3	-270 to 400	-6.4 to 10.9
N14	0 to 1300	0.0 to 47.5	0 to 1300	-0.7 to 46.8	0 to 1300	-2.0 to 45.5
S	-50 to 1780	-0.2 to 18.8	-50 to 1780	-0.4 to 18.7	-50 to 1780	-0.7 to 18.4
R	-50 to 1780	-0.2 to 21.3	-50 to 1780	-0.4 to 21.1	-50 to 1780	-0.7 to 20.8
B	50 to 1780	0.0 to 13.4	50 to 1780	0.0 to 13.4	50 to 1780	0.0 to 13.4

daqTCSetup

DLL Function	`daqTCSetup(uint nscan, uint cjcPosition, uint  ntc , uint tcType, uchar bipolar, uint avg);`
C	`daqTCSetup(uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, unsigned avg)`
QuickBASIC	`QBdaqTCSetup%(nscan%, cjcPosition%, ntc%, tcType%, bipolar%, avg%)`
Turbo Pascal	`daqTCSetup(nscan, cjcPosition, ntc, tcType: word; bipolar: boolean; avg: word):integer;`
Parameters	
`uint nscans`	The number of readings in a single scan of DaqBook/DaqBoard data. The daqTC functions can convert several consecutive scans worth of data in a single invocation. Valid range: 2 to 512.
`uint cjcPosition`	The position of the actual cold-junction compensation circuit (CJC) reading within each scan (not the CJC zero reading, if any). The first reading of the scan is position 0, and the last reading is position -1. Each scan of temperature data must include a reading of the CJC signal on the expansion board to which the thermocouples are attached. The CJC readings must be taken with the gain in the section *Scan Setup*. Valid range: 0 to nscan-2 with no zero compensation; 2 to nscan-2 with zero compensation.
`uint ntc`	The number of thermocouple signals that are to be converted to temperature values. The thermocouple signal readings must immediately follow the CJC reading in the scan data. The first thermocouple signal is at scan position cjcPosition+1,; the next is at cjcPosition+2,; and so on. Valid range: 1 to nscan-1-cjcPosition.
`uint tcType`	The type of thermocouples that generated the measurements. Valid range: One of the pre-defined values: `Dbk19TCTypeJ, Dbk19TCTypeK, Dbk19TCTypeT, Dbk19TCTypeE, Dbk19TCTypeN28, Dbk19TCTypeN14, Dbk19TCTypeS, Dbk19TCTypeR` or `Dbk19TCTypeB`.
`uchar bipolar`	Must be set true (non-zero) if the readings were acquired with the Daq set for bipolar operation. Must be set false (zero) for unipolar operation. The required gain settings for the CJC and thermocouple channels change depending on the unipolar/bipolar mode. Valid range: 0 for unipolar or any non-zero value for bipolar.
`uint avg`	The type of averaging to be performed. Valid range: any unsigned integer. Since the thermocouple voltage may be small compared to the ambient electrical noise, averaging may be necessary to yield a steady temperature output. 0 specifies block averaging in which all of the scans are averaged together to compute a single temperature measurement for each of the ntemp thermocouples. 1 specifies no averaging. Each scan's readings are converted into ntemp measured temperatures for a total of scans*ntemp results. 2 or more specifies moving average of the specified number of scans. Scan readings are averaged with the avg-1 preceding scans' readings before conversion. The first avg-1 scans are averaged with all of the preceding scans because they do not have enough preceding scans. For example, if avg is 3, then the results from the first scan are not averaged at all, the results from the second scan are averaged with the first scan, the results from the third and subsequent scans are averaged with the preceding two scans as shown in the table.
Returns	`DerrTCE_PARAM` - Parameter out of range `DerrTCE_TYPE` - Invalid thermocouple type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqTCConvert, daqTCSetupConvert`
Program References	None
Used With	

`daqTCSetup` sets up parameters for subsequent temperature conversions. The table shows how averages are computed.

Scan	Readings from Channel		Results from Channel	
	0	1	0	1
1	1A	2A	1A	2A
2	1B	2B	(1A+1B)/2	(2A+2B)/2
3	1C	2C	(1A+1B+1C)/3	(2A+2B+2C)/3
4	1D	2D	(1B+1C+1D)/3	(2B+2C+2D)/3
5	1E	2E	(1C+1D+1E)/3	(2C+2D+2E)/3
6	1F	2F	(1D+1E+1F)/3	(2D+2E+2F)/3

daqTCSetupConvert

DLL Function	`daqTCSetupConvert(uint nscan,uint cjcPosition, uint ntc,  uint tcType,uchar bipolar,uint avg, uint *counts, uint scans, int *temp, );`
C	`daqTCSetupConvert(uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg, uint*counts, uint scans, int*temp, uint ntemp)`
QuickBASIC	`QBdaqTCSetupConvert%(nscan%, cjcPosition%, ntc%, tcType%, bipolar%, avg%, counts%(), scans%, temp%(), ntemp%)`
Turbo Pascal	`daqTCSetupConvert(nscan, cjcPosition, ntc, tcType: word; bipolar: boolean; avg: word, var counts; scans: word; var temp; ntemp: word):integer;`
Parameters	
`uint nscan`	The number of readings in a single scan. Valid range: 1- 512
`uint cjcPosition`	The position of the CJC reading within the scan. Valid range: 0 -(nscan-1) 2 -(nscan-1), if auto-zeroing is used with DBK19.
`uint ntc`	The number of thermocouple readings that immediately follow the CJC reading within the scan. Valid range: 1 -(nscan-cjcposition-1)
`uint tcType`	The type of thermocouples being measured.
`uchar bipolar`	Non-zero if the DaqBook/DaqBoard is configured for bipolar readings.
`uint avg`	The type of averaging to be performed: block, none or moving.
`uint *counts`	The raw data (with tags) from one or more scans.
`uint scans`	The number of scans of raw data in counts.
`int *temp`	The converted temperatures in tenths of a degree C.
`uint ntemp`	The number of elements provided in the temp array (for error checking).
Returns	`DerrTCE_PARAM` - Parameter out of range `DerrTCE_TYPE` - Invalid thermocouple type `DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqTCSetup, daqTCConvert`
Program References	None
Used With	

`daqTCSetupConvert` sets up and converts raw A/D readings into temperature readings.

daqVersion

DLL Function	`daqVersion(uint *hardware);`
C	`daqVersion(uint *hardware)`
QuickBASIC	`QBdaqVersion%(hardware%)`
Turbo Pascal	`daqVersion(hardware: WordP):integer;`
Parameters	
`hardware`	Pointer to variable to receive hardware version 100 for DaqBook/100; 112 for DaqBook/112; 200 for DaqBook/200; 216 for DaqBook/216; 1100 for DaqBoard/100A; 1112 for DaqBoard/112A,; 1200 for DaqBoard/200A; 1116 for DaqBoard/216A
Returns	`DerrNoError` - No Error (also, refer to *API Error Codes* on page 5-68)
See Also	None
Program References	None
Used With	

`daqVersion` returns the hardware version.

daqZeroConvert

DLL Function	`daqZeroConvert(uint *counts, uint scans);`
C	`daqZeroConvert(uint *counts, uint scans)`
QuickBASIC	`QBdaqZeroConvert%(counts%, scans%)`
Turbo Pascal	`daqZeroConvert(counts:scans):integer;`
Parameters	
`uint *counts`	The raw data from one or more scans.
`uint scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqZeroSetup, daqZeroSetupConvert, daqZeroDbk19`
Program References	None
Used With	

`daqZeroConvert` compensates one or more scans according the previously called **`daqZeroSetup`** function. This function will modify the array of data passed to it.

daqZeroDbk19

DLL Function	`daqZeroDbk19(uint zero);`
C	`daqZeroDbk19(uint zero)`
QuickBASIC	`QBdaqZeroDbk19%(zero%)`
Turbo Pascal	`daqZeroDbk19(zero):integer;`
Parameters	
`uint zero`	If non-zero will enable auto zero compensation in the `daqTC`… functions
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqZeroSetup, daqZeroConvert, daqZeroSetupConvert, daqTCSetup, daqTCConvert, daqTCSetupConvert`
Program References	None
Used With	

`daqZeroDbk19` will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the DBK19. When enabled, the thermocouple conversion functions will require a CJC zero reading and a TC zero reading to precede the actual CJC and TC reading. This can easily be done by configuring the scan group to read channel 1 using the DBK19 CJC gain code (CJC zero), channel 1 using the gain code of the connected TC (TC zero), channel 0 using the DBK19 CJC gain code (CJC) and finally the thermocouple channels using the gain code of the connected thermocouples.

Note: the offset of the real CJC value should be specified, not the offset of the CJC zero, when calling the thermocouple linearization setup functions.

daqZeroSetup

DLL Function	`daqZeroSetup(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings);`
C	`daqZeroSetup(uint nscan, uint ZeroPostition, uint readingsPosition, uint nReadings)`
QuickBASIC	`QBdaqZeroSetup%(nscan%, ZeroPostition%, readingsPosition%, nReadings%)`
Turbo Pascal	`daqZeroSetup(nscan: zero position: readings position: nReadings):integer;`
Parameters	
`uint nscan`	The number of readings in a single scan.
`uint zeroPosition`	The position of the zero reading within the scan
`uint readingsPosition`	The position of the readings to be zeroed within the scan.
`uint nReadings`	The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqZeroConvert, daqZeroSetupConvert, daqZeroDbk19`
Program References	None
Used With	

`daqZeroSetup` configures the location of the shorted channel and the channels to be zeroed within a scan, the size of the scan and the number of readings to zero. This function does not do the conversion. A non-zero return value indicates an invalid parameter error.

daqZeroSetupConvert

DLL Function	`daqZeroSetupConvert(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings, uint *counts, uint scans);`
C	`daqZeroSetupConvert(uint nscan, uint ZeroPostition, uint readingsPosition, uint nReadings, uint *counts, uint scans)`
QuickBASIC	`QBdaqZeroSetupConvert%(nscan%, ZeroPostition%, readingsPosition%, nReadings%,counts%, scans%)`
Turbo Pascal	`daqZeroSetupConvert(nscan: zero position: readings position: nReadings):integer;`
Parameters	
`uint nscan`	The number of readings in a single scan.
`uint zeroPosition`	The position of the zero reading within the scan
`uint readingsPosition`	The position of the readings to be zeroed within the scan.
`uint nReadings`	The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading.
`uint *counts`	The raw data from one or more scans.
`uint scans`	The number of scans of raw data in the counts array.
Returns	`DerrZCInvParam` - Invalid parameter value `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqZeroSetup, daqZeroConvert, daqZeroDbk19`
Program References	None
Used With	

For convenience, both the setup and convert steps can be performed with one call to **`daqZeroSetupConvert`**. This is useful when the zero compensation needs to be performed multiple times because data was read from channels at different gains or from different boards.

daq200GetScan

DLL Function	`daq200GetScan(uint *chans, uchar *gains, uchar *polarity, uint  count);`
C	`daq200GetScan(uint *chans, uchar *gains, uchar *polarity, uint *count)`
QuickBASIC	`QBdaq200GetScan%(chans%(), gains%(), polarity%(), count%)`
Turbo Pascal	`daq200GetScan(chans: DataP;  gains: ByteP; polarity: ByteP; count: WordP):integer;`
Parameters	
`uint *chans`	An array to hold up to 512 channel numbers or 0 if the channel information is not desired.
`uchar *gains`	An array to hold up to 512 gain values or 0 if the channel gain information is not desired.
`uchar *polarity`	An array to hold up to 512 polarity values or 0 if the polarity information is not desired.
`uint count`	A variable to hold the number of values returned in the chans and gains arrays
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcGetScan, daq200SetScan`
Program References	None
Used With	Does not apply to DaqBook/100/112/120

`daq200GetScan` retrieves a scan sequence much like daqAdcGetScan.

daq200SetMode

DLL Function	`daq200SetMode(uchar  di_se, uchar   polarity, uchar comp);`
C	`daq200SetMode(uchar di_se, uchar polarity, uchar comp)`
QuickBASIC	`QBdaq200SetMode%( di se%, polarity%, comp%)`
Turbo Pascal	`daq200SetMode(die se: Word; polarity:word ; cop:word):integer;`
Parameters	
`uchar di_se`	Zero value causes DaqBook to go to single-ended mode (power-on default). Non-zero value causes differential mode.
`uchar polarity`	Zero value causes DaqBook to default to Unipolar mode. Non-zero value causes default Bipolar mode. All ADC conversions except those started with daq200SetScan will use the default polarity.
`uchar comp`	Non-zero value causes DaqBook/DaqBoard complement all data from Bipolar channels. This makes the acquired data integer values; negative numbers correspond with negative voltages and positive numbers correspond with positive voltages.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	
Program References	None
Used With	**Does not apply to DaqBook/100/112/120**

`daq200SetMode` is used to program the gain amp for single-ended or differential operation and to set the default polarity.

- **Single-ended** operation measures the voltage of the selected channel referred to analog ground.
- **Differential** operation measures differences in voltage between the pair of selected channels.

Voltage polarity can be unipolar or bipolar:

- **Unipolar** maximum voltage range is 0 V to +10 V
- **Bipolar** maximum voltage range is -10 V to +10 V.

daq200SetScan

DLL Function	`daq200SetScan(uint  *chans, uchar  * gains, uchar *polarity, uint  count);`
C	`daq200SetScan(uint *chans, uchar *gains, uchar *polarity, uint count)`
QuickBASIC	`QBdaq200SetScan%(chans%(), gains%(), polarity%(), count%)`
Turbo Pascal	`daq200SetScan(chans: DataP;  gains: ByteP; polarity: ByteP; count: Word):integer;`
Parameters	
`uint *chans`	An array of up to 512 channel numbers Valid values: 0-15 For local A/D channels 16-271 For local expansion A/D channels 272 For the high speed digital I/O input
`uchar *gains`	An array of up to 512 gain values
`uchar *polarity`	An array of up to 512 polarity values. Zero value causes DaqBook/DaqBoard to select Unipolar mode. Non-zero values causes Bipolar mode.
`uint count`	The number of values in the chans and gain arrays Valid values: 1-512
Returns	`DerrInvCount` - Invalid count `DerrInvChan` -Invalid channel `DerrInvGain` - Invalid gain `DerrNotCapable` - No programmable polarity `DerrNoError` - No error (also, refer to *API Error Codes* on page 5-68)
See Also	`daqAdcGetScan, daq200SetMode`
Program References	None
Used With	Does not apply to DaqBook/100/112/120

`daq200SetScan` configures a scan sequence much like **`daqAdcSetScan`** with the addition of a polarity mode per channel.

A/D Channel Descriptions

A/D Channel	Source
0 to 15	Local channels 0 to 15
16 to 31	Channels 0 to 15 of A/D expansion card 0
32 to 47	Channels 0 to 15 of A/D expansion card 1
48 to 63	Channels 0 to 15 of A/D expansion card 2
64 to 79	Channels 0 to 15 of A/D expansion card 3
80 to 95	Channels 0 to 15 of A/D expansion card 4
96 to 111	Channels 0 to 15 of A/D expansion card 5
112 to 127	Channels 0 to 15 of A/D expansion card 6
128 to 143	Channels 0 to 15 of A/D expansion card 7
144 to 159	Channels 0 to 15 of A/D expansion card 8
160 to 175	Channels 0 to 15 of A/D expansion card 9
176 to 191	Channels 0 to 15 of A/D expansion card 10
192 to 207	Channels 0 to 15 of A/D expansion card 11
208 to 223	Channels 0 to 15 of A/D expansion card 12
224 to 239	Channels 0 to 15 of A/D expansion card 13
240 to 255	Channels 0 to 15 of A/D expansion card 14
256 to 271	Channels 0 to 15 of A/D expansion card 15
272	High speed digital I/O (DaqBook/100, DaqBook/200, DaqBoard/100A or DaqBoard/200A)
Note: In differential mode, only (sub) channels 0 to 7 are valid.	

Thermocouple Types

Description	Value		Description	Value
`Dbk14TCTypeJ`	`0`		`Dbk14TCTypeK`	`1`
`Dbk14TCTypeT`	`2`		`Dbk14TCTypeE`	`3`
`Dbk14TCTypeN28`	`4`		`Dbk14TCTypeN14`	`5`
`Dbk14TCTypeS`	`6`		`Dbk14TCTypeR`	`7`
`Dbk14TCTypeB`	`8`		`Dbk19TCTypeJ`	`9`
`Dbk19TCTypeK`	`10`		`Dbk19TCTypeT`	`11`
`Dbk19TCTypeE`	`12`		`Dbk19TCTypeN28`	`13`
`Dbk19TCTypeN14`	`14`		`Dbk19TCTypeS`	`15`
`Dbk19TCTypeR`	`16`		`Dbk19TCTypeB`	`17`

A/D Trigger Source Definitions

Definition	Value	Trigger
`DtsPacerClock`	`00h`	8254 Pacer Clock
`DtsSoftware`	`10h`	Software
`DtsTTLFall`	`20h`	External TTL falling edge
`DtsTTLRise`	`30h`	External TTL rising edge
`DtsAnalogFallNeg`	`40h`	Falling below a negative setpoint
`DtsAnalogRiseNeg`	`50h`	Rising above a negative setpoint
`DtsAnalogRisePos`	`60h`	Rising above a positive setpoint
`DtsAnalogFallPos`	`70h`	Falling below positive setpoint

A/D Gain Definitions

BASE UNIT	
Description	Value
DgainX1	00h
DgainX2	01h
DgainX4	02h
DgainX8	03h

DBK12	
Description	Value
Dbk12X1	00h
Dbk12X2	01h
Dbk12X4	02h
Dbk12X8	03h
Dbk12X16	13h
Dbk12X32	23h
Dbk12X64	33h

DBK13	
Description	Value
Dbk13X1	00h
Dbk13X2	10h
Dbk13X4	20h
Dbk13X8	30h
Dbk13X10	01h
Dbk13X20	11h
Dbk13X40	21h
Dbk13X80	31h
Dbk13X100	02h
Dbk13X200	12h
Dbk13X400	22h
Dbk13X800	32h
Dbk13X1000	03h
Dbk13X2000	13h
Dbk13X4000	23h
Dbk13X8000	33h

DBK14		
Description	Bipolar Value	Unipolar Value
Dbk14BiGainCJC	10h	20h
Dbk14BiGainJ	02h	12h
Dbk14BiGainK	31h	12h
Dbk14BiGainT	12h	22h
Dbk14BiGainE	21h	02h
Dbk14BiGainN28	22h	32h
Dbk14BiGainN14	02h	12h
Dbk14BiGainS	12h	22h
Dbk14BiGainR	12h	22h
Dbk14BiGainB	22h	32h

DBK15		
Description	Bipolar Value	Unipolar Value
Dbk15BiX1	00h	02h
Dbk15BiX2	01h	03h

DBK16	
Description	Value
Dbk16ReadBridge	00h
Dbk16SetOffse	01h
Dbk16SetInputGain	02h
Dbk16SetScalingGain	03h

DBK19		
Description	Bipolar Value	Unipolar Value
Dbk19BiCJC	00h	01h
Dbk19BiTypeJ	01h	02h
Dbk19BiTypeK	01h	02h
Dbk19BiTypeT	02h	03h
Dbk19BiTypeE	00h	01h
Dbk19BiTypeN28	03h	03h
Dbk19BiTypeN14	01h	02h
Dbk19BiTypeS	03h	03h
Dbk19BiTypeR	02h	03h
Dbk19BiTypeB	03h	03h

DBK42	
Description	Value
Dbk42X1	00h

DBK43	
Description	Value
Dbk43ReadBridge	00h
Dbk43SetOffset	01h
Dbk43SetInputGain	02h
Dbk43SetScalingGain	03h

DBK44	
Description	Value
Dbk44X1	00h

DBK50	
Description	Value
Dbk50Range1	00h
Dbk50Range10	01h
Dbk50Range100	02h
Dbk50Range300	03h

Digital I/O Port Connection

Base Unit		
Description	**Value**	**Address Select Jumper Location**
Ddp4BitIO	**83h**	Connector P1
DdpLocalA	**10h**	Connector P2 Port A
DdpLocalB	**11h**	Connector P2 Port B
DdpLocalC	**12h**	Connector P2 Port C
DdpLocalCHigh	**B2h**	Connector P2 Port C High Nibble
DdpLocalCLow	**92h**	Connector P2 Port C Low Nibble
Expansion Unit Address A		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21**
DdpExp0A	**60h**	Dig Exp Chan 0 Port A / (P2 A)
DdpExp0B	**61h**	Dig Exp Chan 0 Port B / (P2 A)
DdpExp0C	**62h**	Dig Exp Chan 0 Port C / (P2 A)
DdpExp0High	**E2h**	Dig Exp Chan 0 Port C High Nibble / (P2 A)
DdpExp0Low	**C2h**	Dig Exp Chan 0 Port C Low Nibble / (P2 A)
DdpExp1A	**64h**	Dig Exp Chan 1 Port A / (P3 A)
DdpExp1B	**65h**	Dig Exp Chan 1 Port B / (P3 A)
DdpExp1C	**66h**	Dig Exp Chan 1 Port C / (P3 A)
DdpExp1CHigh	**E6h**	Dig Exp Chan 1 Port C High Nibble / (P3 A)
DdpExp1Low	**C6h**	Dig Exp Chan 1 Port C Low Nibble / (P3 A)
Expansion Unit Address B		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
DdpExp2A	**68h**	Dig Exp Chan 2 Port A / (P2 B)
DdpExp2B	**69h**	Dig Exp Chan 2 Port B / (P2 B)
DdpExp2C	**6Ah**	Dig Exp Chan 2 Port C / (P2 B)
DdpExp2CHigh	**EAh**	Dig Exp Chan 2 Port C High Nibble / (P2 B)
DdpExp2Low	**CAh**	Dig Exp Chan 2 Port C Low Nibble / (P2 B)
DdpExp3A	**6Ch**	Dig Exp Chan 3 Port A / (P3 B)
DdpExp3B	**6Dh**	Dig Exp Chan 3 Port B / (P3 B)
DdpExp3C	**6Eh**	Dig Exp Chan 3 Port C / (P3 B)
DdpExp3CHigh	**EEh**	Dig Exp Chan 3 Port C High Nibble / (P3 B)
DdpExp3Low	**CEh**	Dig Exp Chan 3 Port C Low Nibble / (P3 B)
Expansion Unit Address C		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
DdpExp4A	**70h**	Dig Exp Chan 4 Port A / (P2 C)
DdpExp4B	**71h**	Dig Exp Chan 4 Port B / (P2 C)
DdpExp4C	**72h**	Dig Exp Chan 4 Port C / (P2 C)
DdpExp4CHigh	**F2h**	Dig Exp Chan 4 Port C High Nibble / (P2 C)
DdpExp4Low	**D2h**	Dig Exp Chan 4 Port C Low Nibble / (P2 C)
DdpExp5A	**74h**	Dig Exp Chan 5 Port A / (P3 C)
DdpExp5B	**75h**	Dig Exp Chan 5 Port B / (P3 C)
DdpExp5C	**76h**	Dig Exp Chan 5 Port C / (P3 C)
DdpExp5CHigh	**F6h**	Dig Exp Chan 5 Port C High Nibble / (P3 C)
DdpExp5Low	**D6h**	Dig Exp Chan 5 Port C Low Nibble / (P3 C)
Expansion Unit Address D		
Description	**Value**	**Address Select Jumper Location / (DBK20 & 21)**
DdpExp6A	**78h**	Dig Exp Chan 6 Port A / (P2 D)
DdpExp6B	**79h**	Dig Exp Chan 6 Port B / (P2 D)
DdpExp6C	**7Ah**	Dig Exp Chan 6 Port C / (P2 D)
DdpExp6CHigh	**FAh**	Dig Exp Chan 6 Port C High Nibble / (P2 D)
DdpExp6Low	**DAh**	Dig Exp Chan 6 Port C Low Nibble / (P2 D)
DdpExp7A	**7Ch**	Dig Exp Chan 7 Port A / (P3 D)
DdpExp7B	**7Dh**	Dig Exp Chan 7 Port B / (P3 D)
DdpExp7C	**7Eh**	Dig Exp Chan 7 Port C / (P3 D)
DdpExp7CHigh	**FEh**	Dig Exp Chan 7 Port C High Nibble / (P3 D)
DdpExp7Low	**DEh**	Dig Exp Chan 7 Port C Low Nibble / (P3 D)

API Error Codes

Error Name	Code # hex - dec	Description
DerrNoError	00h - 0	No error
DerrBadChannel	01h - 1	Specified LPT channel was out-of-range
DerrNotOnLine	02h - 2	Requested DaqBook is not online
DerrNoDaqbook	03h - 3	DaqBook is not on the requested channel
DerrBadAddress	04h - 4	Bad function address
DerrFIFOFull	05h - 5	FIFO Full detected, possible data corruption
DerrInvChan	10h - 16	Invalid analog input channel
DerrInvCount	11h - 17	Invalid count parameter
DerrInvTrigSource	12h - 18	Invalid trigger source parameter
DerrInvLevel	13h - 19	Invalid trigger level parameter
DerrInvGain	14h - 20	Invalid channel gain parameter
DerrInvDacVal	15h - 21	Invalid DAC output parameter
DerrInvExpCard	16h - 22	Invalid expansion card parameter
DerrInvPort	17h - 23	Invalid port parameter
DerrInvChip	18h - 24	Invalid chip parameter
DerrInvDigVal	19h - 25	Invalid digital output parameter
DerrInvBitNum	1Ah - 26	Invalid bit number parameter
DerrInvClock	1Bh - 27	Invalid clock parameter
DerrInvTod	1Ch - 28	Invalid time-of-day parameter
DerrInvCtrNum	1Dh - 29	Invalid counter number
DerrInvCntSource	1Eh - 30	Invalid counter source parameter
DerrInvCtrCmd	1Fh - 31	Invalid counter command parameter
DerrInvGateCtrl	20h - 32	Invalid gate control parameter
DerrInvOutputCtrl	21h - 33	Invalid output control parameter
DerrInvInterval	22h - 34	Invalid interval parameter
DerrTypeConflict	23h - 35	An integer was passed to a function requiring a character
DerrMultBackXfer	24h - 36	A second background transfer was requested
DerrInvDiv	25h - 37	Invalid Fout divisor
DerrTCE_TYPE	26h - 38	TC type out-of-range
DerrTCE_TRANGE	27h - 39	Temperature out-of-CJC-range
DerrTCE_VRANGE	28h - 40	Voltage out-of-TC-range
DerrTCE_PARAM	29h - 41	Unspecified parameter value error
DerrTCE_NOSETUP	2Ah - 42	**dacTCConvert** called before **dacTCSetup**
DerrNotCapable	2Bh - 43	DaqBook is incapable of function
DerrOverrun	2Ch - 44	A buffer overrun occurred
DerrNoPreTActive	32h - 50	No pretrigger configured
DerrInvDacChan	33h - 51	DAC channel does not exist
DerrInvDacParam	34h - 52	DAC parameter is invalid
DerrInvBuf	35h - 53	Buffer point to NULL or size
DerrMemAlloc	36h - 54	Could not allocate the needed memory
DerrUpdateRate	37h - 55	Could not achieve the specified update rate
DerrInvDacWave	38h - 56	Could not start waveforms because of missing or invalid parameters
DerrInvBackDac	39h - 57	Could not start waveforms with background transfers
DerrInvPredWave	3Ah - 58	Predefined waveform not supported
DerrRtdValue	3Bh - 59	**rtdValue** out-of-range
DerrRtdNoSetup	3Ch - 60	**rtdConvert** called before **rtdSetup**
DerrRtdArraySize	3Dh - 61	Temperature array not large enough
DerrRtdParam	3Eh - 62	Incorrect RTD parameter
DerrInvBankType	3Fh - 63	Invalid bank type specified
DerrBankBoundary	40h - 64	Simultaneous writes to DBK cards in different banks not allowed
DerrInvFreq	41h - 65	Invalid scan frequency specified
DerrNoDaq	42h - 66	No Daq112B/216B installed
DerrInvOptionType	43h - 67	Invalid option type parameter
DerrInvOptionValue	44h - 68	Invalid option value parameter
DerrInvParam	45h - 69	Invalid parameter
DerrNoSetup	46h - 70	A …**convert** function was called before …**setup**
DerrArraySize	47h - 71	The array size is too small to hold converted data

Visual Basic VBX Support 6

Overview

Five VBX files provide access to all Daq* hardware functions including: analog input, analog output, digital I/O, counter/timers, and communications. Each VBX tool has its own icon.

To use the VBX tools, add the DBK.VBX tool to your project by selecting *Add File...* from the File item in the VB menu. A file selection box will allow you to select the VBX tool, then click OK. In the same way, add as many of the other VBX tools as you need. For example, if your application requires analog input only, just add the ADC.VBX (must also include dbk.bas). The following table describes the five included VBXs and related drivers.

VBX	Filename	Description
DBK	dbk.vbx	Performs Daq* configuration and opening and closing the driver; must be included
ADC	adc.vbx	Performs A/D functions, selectable
CTR	ctr.vbx	Performs Counter/Timer Functions, selectable
DAC	dac.vbx	Performs D/A functions, selectable
DIO	dio.vbx	Performs Digital I/O Functions, selectable
	daqbook.dll	DaqBook driver V1.7 or greater
	dbk.bas	Must be included if adc.vbx or ctr.vbx are used

To use the Daq* VB controls, place the DBK control on one of your forms with any or all of the controls. Selecting a control on your application form will display its design-time properties in the Properties window.

This chapter has 6 sections: one for each VBX and one of example programs.

DBK VBX

The following table lists general configuration properties of the VBXs.

- The “R/W” in the column heading stands for Read/Write. An “R” in this column means that the property can be Read or assigned to a variable as follows: **processStatus = ADC1.Active.** A “W” in this column means that the property can be Written to or assigned a value as follows: **ADC1.Arm = True.** A “R/W” in this column means that the property possesses both read and write characteristics.
- The “Access” column signifies whether the property is accessible in the Properties window. “Run” in this column means the property can only be accessed in code at run-time (it is not visible in the Properties window). If “Design” is in this column, the property is visible and accessible in the Properties window at Design-time.

VBX General Configuration Properties				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
IntLevel	Specifies the LPT interrupt level prior to assigning the Open property	R/W	Design/Run	0 - 7 (DaqBook) 10-15 (DaqBoard)
LptPort	Specifies the LPT port number prior to assigning the Open property	R/W	Design/Run	(DaqBook Only:) 0 - LPT1 1 - LPT2 2 - LPT3 3 -LPT4
Open	After the LptPort, IntLevel, and Protocol properties have been set, this method, if set to True, initializes the driver and establishes communication with the Daq*. If set to False, this method performs a close function	W	Run	True to open False to close
Protocol	Specifies which type of parallel-port implementation and protocol is available to the computer	R/W	Design/Run	(DaqBook Only:) 0 - 8 bit I/O 1 - 4 bit I/O 2-Far Point F/Port EPP Interface 3 - 82360SL EPP Interface
Version	Return the hardware version number of the Daq* being used	R	Run	100 - DaqBook/100 112 -DaqBook/112 200-DaqBook/200 216-DaqBook/216 1100-DaqBoard/100 1112-DaqBoard/112 1200-DaqBoard/200 1216-DaqBoard/216
Type	Specifies the type of hardware prior to assigning the Open Property	R/W	Design/Run	0 -DaqBook 1 - DaqBoard
ISA Address	Specifies the address of the base port for a DaqBoard prior to assigning the open property	R/W	Design/Run	0 -Port_0300 1 - Port_0304 2 -Port_0308... 15 -Port_033C
DM Channel	Specified the DMA channel for the DaqBoard prior to assigning the Open Property	R/W	Design/Run	0 -DMA None 1 - DMA 5 2 - DMA 6 3 - DMA 7

Event Routines- DBK:

None

DBK Properties

IntLevel

Access:	Read and write
Valid settings:	0 - 7 for DaqBook; 10 - 15 for DaqBoard
Syntax:	Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 Dbk1.Open = True 'This line usually appears in the Form_Load subroutineDbk1. Open = False Closing the driver usually takes place in the Form_Unload subroutine

Specifies the LPT or DaqBoard interrupt level prior to assigning the Open property. The default interrupt level is 7.

LptPort (DaqBook Only)

Access:	Read and write
Valid settings:	0 - LPT1; 1 - LPT2; 2 - LPT3; 3 - LPT4
Syntax:	Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 Dbk1.Open = True 'This line usually appears in the Form_Load subroutine Dbk1.Open = False 'Closing the driver usually takes place in the Form_Unload subroutine

Specifies the LPT port number prior to assigning the Open property. The default LPT port is 1. If you only have one LPT port in the system, it is very likely LPT1.

Open

Access:	Write only
Valid settings:	True to open; False to close
Syntax:	Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 Dbk1.Open = True 'This line usually appears in the Form_Load subroutine Dbk1.Open = False 'Closing the driver usually takes place in the Form_Unload subroutine

After the LptPort or ISA Bus Address, IntLevel, and Protocol properties have been set, this method, if set to True, initializes the driver and establishes communication with the Daq*. If set to False, this method performs a close function.

Protocol (DaqBook Only)

Access:	Read and write
Valid settings:	0 - 8 bit I/O 1 - 4 bit I/O 2 - Far Point F/Port EPP Interface 3 - 82360 SL EPP Interface 4 - Quatech SMC666 EPP Interface 5 - EPP Bios (Draft Revision 3)
Syntax:	Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 'Set to 8-bit protocol Dbk1.Open = True 'This line usually appears in the Form_Load subroutine Dbk1.Open = False 'Closing the driver usually takes place in the Form_Unload subroutine

Specifies which type of parallel-port implementation and protocol is available to the computer. EPP (Enhanced Parallel Port) will allow the fastest data transfer, if it is supported by your LPT port. 8-bit operation is 2nd fastest, while 4-bit is the slowest, but compatible with virtually all LPT ports.

Version

Access:	Read only
Syntax:	Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 'Set to 8-bit protocol Dbk1.Open = True 'This line usually appears in the Form_Load subroutine ersionNumber = DBK1.Version.Dbk1.Open = False 'Closing the driver usually takes place in the Form_Unload subroutine

Returns the hardware version of the Daq DAS Family being used, allowing the user to properly interpret data and issue commands.

Type

Valid Settings:	0 - DaqBook; 1 - DaqBoard
Syntax:	Dbk1.Type = 0 'for a DaqBook product Dbk1.IntLevel = 7 Dbk1.LptPort = 0 Dbk1.Protocol = 0 Dbk1.Open = TRUE

Specifies the type of hardware to being attached to prior to assigning the Open property. Some of the DBK VBX's properties are only relevant for one hardware type or the other. The default value is 0 - DaqBook.

ISA Address

Valid Settings:	0 - Port_0300 1 - Port_0304 2 - Port_0308... 15 - Port_033C
Syntax:	Dbk1.Type = 1 'for a DaqBoard product Dbk1.IntLevel = 8 ' for interrupt 10 Dbk1.IsaAddress = PORT_0300 Dbk1.DmaChannel = DMA5 Dbk1.Open = TRUE

Specifies the address of the base port for a DaqBoard product prior to assigning the Open property. The base port address are set on hardware switches. The addresses start at &H300 and are spaced every 4 addresses from there. e.g. &H300, &H304, &H308 &H30C, ... The default is PORT_0300.

DMA Channel

Valid Settings:	0 - DMA None 1 - DMA 5 2 - DMA 6 3 - DMA 7
Syntax:	Dbk1.Type = 1 'for a DaqBoard product Dbk1.IntLevel = 8 ' for interrupt 10 Dbk1.IsaAddress = PORT_0300 Dbk1.DmaChannel = DMA5 Dbk1.Open = TRUE

Specifies the DMA channel to be used for a DaqBoard product prior to assigning the Open property. The default is DMANONE (no DMA used).

ADC VBX

ADC VBX Property Summary				
Property	**Description**	**R/ W**	**Access**	**Valid Settings**
Active	A status flag showing if the acquisition is active	R	Run	True or False
Arm	After all of the acquisition parameters are set, this property arms the acquisition	W	Run	True to Arm False to Disarm
BipolarArray	When UseChanArray is true, BipolarArray holds the pole configuration values for the associated channel in the ChanArray. (DaqBook/200 and DaqBoards only)	W	Run	An integer array of elements assigned to either True or False.
Buffer	Points to the user buffer for the incoming data. All data is collected directly into a VB integer buffer in the background	W	Run	The 0th element of a user-dimensioned integer array.
Buffered	Represents how many scans have been collected and placed in the buffer.	R	Run	0-4000000
BufferLength	Represents the usable length of the user-allocated integer array.	R/ W	Design/ Run	1 to 32767
BufferOverrun	Indicates whether or not a buffer overrun condition exists.	R	Run	True or False
ChanArray	When UseChanArray is true, ChanArray holds the array of channels used in the scan	W	Run	An integer array of channel numbers, each with a value of 0 to 272 .
EndChan	When UseChanArray is false, StartChan represents the first channel in a channel range that ends with EndChan	R/ W	Design/ Run	0 - 272 See channel table.
Frequency	Sets the scan rate for acquisitions containing more than one scan	R/ W	Design/ Run	100000.0 to 0.0002
GainArray	When UseChanArray is true, GainArray holds the gain values for the associated channels in the ChanArray	W	Run	An integer array of gain values. See gain table.
GlobalGain	When UseChanArray is false, GlobalGain represents the gain to be used on all of the channels specified in the channel range StartChan to EndChan	R/ W	Design/ Run	See gain table.
GlobalBipolar	When UseChanArray is false, specifies the bipolar or unipolar inputs for all channels in the scan range specified by StartChan through EndChan. (DaqBook/200 only	R/ W	Design/ Run	True or False
NumChannels	When UseChanArray is true, NumChannels holds an integer representing the number of channels in the channel array	R/ W	Design/ Run	1 - 512
NumScans	The number of scans to collect	R/ W	Design/ Run	1 - 32767, or -1 for infinite cycle
OneShot	If true enables one-shot trigger mode	R/ W	Design/ Run	True or False
GlobalSE	Specifies Single Ended or Differential inputs (DaqBook/200 only	R/ W	Design/ Run	True or False
SoftTrig	If trigger source is Software, provides the trigger condition	W	Run	True to trigger
StartChan	When UseChanArray is false, StartChan represents the first channel in a channel range that ends with EndChan	R/ W	Design/ Run	0 - 272 See channel table in chapter 11.
TrigLevel	The analog trigger setpoint	R/ W	Design/ Run	-10.0 to +10.0 volts
TrigRefVoltage	If analog trigger is the source, this represents the external reference voltage of D/A channel 1	R/ W	Design/ Run	-10.0 to 0.0 volts
TrigSource	The source of the trigger	R/ W	Design/ Run	0 - Software 1 - TTL 2 - Analog
TrigSourceRising	Specifies a rising or a falling trigger (TTL and Analog only	R/ W	Design/ Run	True or False
UseChanArray	Specifies use of StartChan/EndChan or ChanArray to specify the desired channels in the scans.	R/ W	Design/ Run	True or False

Event Routines- ADC

When the ADC.VBX tool is placed on an application form, 2 subroutine stubs are automatically created. They are:

```
Sub ADC1_Triggered( )
Sub ADC1_AcquisitionComplete( )
```

When the system trigger has been satisfied by either an internal or external event, the subroutine ADC1_Triggered is automatically called. Code required to post status or begin the data transfer process can be located in this routine.

The trigger event is monitored in the hardware and passed on the custom control in the background during the transfer of the first block of data that has been buffered in the external hardware. If the sample frequency is slow, the external buffer may take a few seconds to fill, delaying the notification of the custom control that the trigger has been satisfied.

When the acquisition is completely finished, the Active property will become false and the subroutine ADC1_AcquisitionComplete will automatically be called.

ADC.VBX Note

When using the analog input control, the file DBK.BAS must be added to your application. This file contains the function declaration **addressof** which gets the address of a VB integer array. The Buffer property accepts this pointer.

Example:

```
Dim MyData(1000)As Integer
        'The Following line would typically be placed in the Form_Load subroutine.
ADC1.Buffer = addressof(MyData(0))
```

ADC VBX Properties

The gain definitions are shown in the table.

VBX Gain Entry Table		
0 - Base Unit X1	20 - Dbk13 ×200	40 - Dbk16 Set Offset
1 - Base Unit ×2	21 - Dbk13 ×400	41 - Dbk16 Input Gain
2 - Base Unit ×4	22 - Dbk13 ×800	42 - Dbk16 Scaling Gain
3 - Base Unit ×8	23 - Dbk13 ×1000	43 - 00 Hex (reserved)
4 - Dbk12 ×1	24 - Dbk13 ×2000	44 - 01 Hex (reserved)
5 - Dbk12 ×2	25 - Dbk13 ×4000	45 - 02 Hex (reserved)
6 - Dbk12 ×4	26 - Dbk13 ×8000	46 - 03 Hex (reserved)
7 - Dbk12 ×8	27 - Dbk14 Bipolar CJC	47 - 10 Hex (reserved)
8 - Dbk12 ×16	28 - Dbk14 Bipolar Type J	48 - 11 Hex (reserved)
9 - Dbk12 ×32	29 - Dbk14 Bipolar Type K	49 - 12 Hex (reserved)
10 - Dbk12 ×64	30 - Dbk14 Bipolar Type T	50 - 13 Hex (reserved)
11 - Dbk13 ×1	31 - Dbk14 Unipolar CJC	51 - 20 Hex (reserved)
12 - Dbk13 ×2	32 - Dbk14 Unipolar Type J	52 - 21 Hex (reserved)
13 - Dbk13 ×4	33 - Dbk14 Unipolar Type K	53 - 22 Hex (reserved)
14 - Dbk13 ×8	34 - Dbk14 Unipolar TypeT	54 - 23 Hex (reserved)
15 - Dbk13 ×10	35 - Dbk15 Bipolar ×1	55 - 30 Hex (reserved)
16 - Dbk13 ×20	36 - Dbk15 Bipolar ×2	56 - 31 Hex (reserved)
17 - Dbk13 ×40	37 - Dbk15 Unipolar ×1	57 - 32 Hex (reserved)
18 - Dbk13 ×80	38 - Dbk15 Unipolar ×2	58 - 33 Hex (reserved)
19 - Dbk13 ×100	39 - Dbk16 Read Bridge	

Active

Access:	Read only
Valid setting:	True or False
Syntax:	If ADC1.Active = False then MsgBox "The acquisition is inactive"

The Active property serves as a status flag to show the state of the armed acquisition. At run-time the Active property returns True to signify that the acquisition is still active, and false if inactive. This property is useful when the state of the acquisition is in question. Upon completion of any acquisition, the Acquisition_Complete routine is automatically called. If polling the acquisition is preferred, use the Active property.

Arm

Access:	Write Only
Valid settings:	True to Arm, False to Disarm
Syntax:	Adc1.Arm = True 'Start the acquisition Adc1.Arm = False 'Stop the acquisition

After all of the acquisition parameters are set, this property arms the acquisition. Setting this property to True arms the acquisition, setting it to False stops data collection and disarms the acquisition.

The ADC VBX "ARM" code only uses "Bipolar Array" or "GlobalBipolar" when utilizing the following components: DaqBook/200, DaqBoard/100/200.

The "ARM" code uses "GlobalSE" (single-ended) only if "UseChanArray" is true, when using the above components.

The "BopolarArray" & "GlobalBipolar" properties have no effect with the following components: DaqBook/216, DaqBoard/112/216

BipolarArray (DaqBook/200 & DaqBoards Only)

Access:	Write only
Valid settings:	True for Bipolar and False for Unipolar
Syntax:	Adc1.UseChanArray = True For i = 0 to 15 Adc1.chanArray(i) = i Adc1.gainArray(i) = 0 Adc1.BipolarArray(i) = TRUE 'Set all chans in scan to bipolar Next i Adc1.NumChannels = 16

When UseChanArray is true, BipolarArray holds the pole configuration values for the associated channel in the ChanArray. Up to 512 array elements can be assigned a value of True or False. The NumChannels property is used to tell the custom control which element in the ChanArray holds the last valid channel.

Buffer

Access:	Write only
Valid settings:	A pointer to the 0th element of a user-dimensioned integer array.
Syntax:	Dim arrayBuffer(1000) as integer Adc1.Buffer = addressof(arrayBuffer(0)) Adc1.BufferLength = 1000

The ADC VBX collects all readings in the background under interrupt control. As the data is acquired, it is placed directly into a user-dimensioned VB integer array. Assign the Buffer property the value of the pointer to the integer array. Once dimensioned, the pointer to the integer array is yielded from the function call "addressof", supplied in the file DBK.BAS. The data in the integer array can be accessed concurrently with the acquisition. The number of valid scans in the integer array can be queried using the Buffered property.

Before any analog input operations are performed, the Buffer property must be assigned a pointer to a valid, dimensioned integer array. The dimensioned array must remain valid during the entire background acquisition. If the array is dimensioned within a subroutine using the ReDim command, this array will be de-allocated as the program leaves the subroutine. If the acquisition is still active, the acquisition will write over an undefined area of memory. For this reason, it is recommended that the array be dimensioned as a Global variable. The assignment of the buffer pointer to the Buffer property is typically done in the Form_Load subroutine.

Buffered

Access:	Read only
Valid settings:	0 - 4000000

Syntax:	Static ScansProcessed as Long
	If ScansProcessed Adc1.Buffered then
	Call moveNewScan
	End if

During the acquisition, the buffer is filled with incoming scans. The Buffered property holds the number of buffered scans that are presently valid in the integer array. For applications that need to act on the data as it is coming in, the Buffered property provides the number of valid scans in the buffer. To calculate the array index for any one sample, the number of the scan should be multiplied by the number of channels in the scan. For example, if there are 4 channels in the scan, and the Buffered property show 100 scans, the number of valid values in the integer array is 400.

If the NumScans property is set to -1, the Daq DAS Family is in Cycle mode, collecting an infinite number of scans. In this mode, the integer array will be modeled as a circular buffer, starting at the beginning as the end is reached. The program is responsible for moving the data to a new location, to disk for example, before old data is overwritten. When in Cycle mode, the Buffered property can exceed the size of the buffer by many times since it is keeping track of the total number of scans that have been collected, not just the number that are presently in the buffer. Your program should keep track of the number of scans processed and compare that number with the value of the Buffered property to see if new data is present.

BufferLength

Access:	Read and Write
Valid settings:	1 to 32767
Syntax:	Dim arrayBuffer(1000) as integer Adc1.Buffer = addressof(arrayBuffer(0)) Adc1.BufferLength = 1000

Represents the usable length of the user-allocated integer array that was assigned to the Buffer property. Assigning the correct value to BufferLength keeps the acquisition from accidently overrunning the end of the dimensioned array. When the NumScans property is set to -1 (infinite cycle mode), the BufferLength property is used by the control to know when to wrap to the beginning of the buffer.

BufferOverrun

Access:	Read only
Valid setting:	True or False
Syntax:	If Adc1.BufferOverrun = True then MsgBox "The FIFO has overrun, data may be missing"

The Daq* has a FIFO (first in first out) buffer on the A/D converter. The ADC.VBX control monitors the amount of data in the FIFO and automatically transfers it into the VB integer array in the background. If the speed of the acquisition is greater than the speed at which the computer is able to upload the data, the FIFO will eventually overrun, setting the BufferOverrun property to True.

ChanArray

Access:	Write only
Valid settings:	Each element can be assigned a channel number from 0 to 272.
Syntax:	Adc1.UseChanArray = True For i = 0 to 15 Adc1.chanArray(i) = i : Adc1.gainArray(i) = 0 Next i Adc1.NumChannels = 16

When UseChanArray is true, ChanArray holds the array of channels used in the scan. Up to 512 array elements can be loaded with any channel number in any order. The sample data in the buffer will be in the same order as the channels in the ChanArray. The NumChannels property is used with the ChanArray property to tell the custom control which element in the ChanArray holds the last valid channel.

EndChan

Access:	Read and write.
Valid settings:	0 - 272. StartChan must be less than or equal to EndChan.
Syntax:	Adc1.StartChan = 0 Adc1.EndChan = 3 Adc1.GlobalGain = 0

When UseChanArray is false, StartChan represents the first channel in a channel range that ends with EndChan. When StartChan and EndChan are used, it is not possible to assign individual gains to the channels, the GlobalGain property is used to assign a gain to all channels in the scan.

Frequency

Access:	Read and write
Valid settings:	100000.0 to 0.0002
Syntax:	Adc1.Frequency = 1000 'Set scan rate to 1kHz

Sets the scan rate, in hertz, for acquisitions containing more than one scan.

GainArray

Access:	Write only
Valid settings:	Any valid gain value.
Syntax:	Adc1.UseChanArray = True For i = 0 to 15 Adc1.chanArray(i) = i : Adc1.gainArray(i) = 0 Next i Adc1.NumChannels = 16

When UseChanArray is true, GainArray holds the gain values for the associated channels in the ChanArray. A scan can consist of as many as 512 channels, in any order. The property NumChannels is used to tell the custom control which element in the ChanArray holds the last valid channel number.

GlobalGain

Access:	Read and write
Valid settings:	Any valid gain value.
Syntax:	Adc1.StartChan = 0 Adc1.EndChan = 3 Adc1.GlobalGain = 0

When UseChanArray is false, GlobalGain represents the gain to be used on all of the channels specified in the channel range StartChan to EndChan. When StartChan and EndChan are used, it is not possible to assign individual gains to the channels; the GlobalGain property is used to assign a gain to all channels in the scan.

GlobalBipolar (DaqBook/200 & DaqBoards Only)

Access:	Read and write
Valid settings:	True for bipolar, False for unipolar
Syntax:	Adc1.StartChan = 0 Adc1.EndChan = 3 Adc1.GlobalGain = 0 Adc1.GlobalBipolar = True 'Set all channels in scan to Bipolar

When UseChanArray is false, specifies the bipolar or unipolar inputs for all channels in the scan range specified by StartChan through EndChan. When StartChan and EndChan are used, it is not possible to assign individual pole values to the channels; the GlobalBipolar property is used to assign bipolar or unipolar to all channels in the scan.

NumChannels

Access:	Read and write
Valid settings:	1 - 512
Syntax:	Adc1.UseChanArray = True For i = 0 to 15 Adc1.chanArray(i) = i : Adc1.gainArray(i) = 0 Next i Adc1.NumChannels = 16

When UseChanArray is true, NumChannels holds an integer representing the number of channels in the ChanArray properties. The NumChannels property is used in conjunction with the ChanArray property to tell the custom control which element in the ChanArray holds the last valid channel.

NumScans

Access:	Read and write
Valid settings:	1 - 32767, or -1 for infinite cycle
Syntax:	Adc1.NumScans = 1000 'collect 1000 scans.

The number of scans to collect. If NumScans is set to -1, the acquisition will continue until it is disarmed by setting the Arm property to false. When the BufferLength is reached, the buffer will wrap around to the beginning, overwriting the oldest scans. In this case, your application should monitor the Buffered property and move the data to other destinations before it is overwritten.

OneShot

Access:	Read and write
Valid settings:	True to enable one-shot trigger mode False to disable one-shot trigger mode.
Syntax:	Adc1.OneShot = True 'Take a scan every trigger event.

When set to true, this property enables one-shot trigger mode, taking one scan on every occurrence of the specified trigger event. When in this mode, the Frequency and NumScans properties are ignored. One-shot trigger mode is useful for synchronizing scans with external events rather than a timebase.

GlobalSE (DaqBook/200 & DaqBoards Only)

Access:	Read and write
Valid settings:	True for SE False for DIFF
Syntax:	Adc1.GlobalSE = True 'Set input configuration to SE

The DaqBook/200 provides programmable control of the single-ended/differential input circuitry. This property sets the system-wide input configuration.

SoftTrig

Access:	Write
Valid settings:	True to trigger
Syntax:	Adc1.TrigSource = 0 'Set source to Software Adc1.Arm = True 'Arm the acquisition Adc1.SoftTrig = True 'Trigger the acquisition

When TrigSource is set to Software, and Arm is set to True, set this property to True to initiate the acquisition trigger. If TrigSource is not set to Software when SoftTrig is set to True, the message "Software Trigger Source Not Selected" appears. No acquisition can be triggered unless the acquisition is first armed by setting the Arm property to True.

StartChan

Access:	Read and write.
Valid settings:	0 - 272. StartChan must be less than or equal to EndChan.
Syntax:	Adc1.StartChan = 0 Adc1.EndChan = 3 Adc1.GlobalGain = 0

When UseChanArray is false, StartChan represents the first channel in a channel range that ends with EndChan. When StartChan and EndChan are used, it is not possible to assign individual gains to the channels; instead the GlobalGain property is used to assign a gain to all channels in the scan.

TrigLevel

Access:	Read and write
Valid settings:	-10.0 to +10.0 volts
Syntax:	Adc1.TrigLevel = 2.0 'Trigger on 2 volts Adc1.TrigSourceRising = True 'Trigger on rising edge Adc1.TrigRefVoltage = -5.0 'Set to default value Adc1.Arm = True 'Arm the acquisition

When the TrigSource is set to Analog, this property provides the voltage level at which the acquisition will be triggered. To trigger as the voltage rises through the setpoint, set the TrigSourceRising property to true, set it to false to trigger on a falling edge. DAC1 is used to provide the analog comparator voltage for the analog trigger source. When the analog trigger source is used, DAC1 becomes unavailable for other purposes.

TrigRefVoltage

Access:	Read and write
Valid settings:	-10.0 to 0.0 volts
Syntax:	Adc1.TrigSource = 2 'Set the trigger source to Analog Adc1.TrigLevel = 2.0 'Trigger on 2 volts Adc1.TrigSourceRising = True 'Trigger on rising edge Adc1.TrigRefVoltage = -5.0 'Set to default value Adc1.Arm = True 'Arm the acquisition

DAC1 is used as the analog trigger comparator input. When the TrigSource property is set to Analog, the value of DAC1 is set by the TrigLevel property in volts. To calculate the binary value required by the 12-bit DAC, the DAC reference voltage must be known. The factory default, internal reference is -5 volts; but when set to External, voltages from 0 to -10 can be applied.

TrigSource

Access:	Read and write
Valid settings:	0 - Software, 1 - TTL , 2 - Analog
Syntax:	Adc1.TrigSource = 2 'Set the trigger source to Analog Adc1.TrigLevel = 2.0 'Trigger on 2 volts Adc1.TrigSourceRising = True 'Trigger on rising edge Adc1.TrigRefVoltage = -5.0 'Set to default value Adc1.Arm = True 'Arm the acquisitionor Adc1.TrigSource = 1 'Set the trigger source to TTL Adc1.TrigSourceRising = False 'Trigger on falling edge Adc1.Arm = True 'Arm the acquisitionor Adc1.TrigSource = 0 'Set the trigger source to Software Adc1.Arm = True 'Arm the acquisition Adc1.SoftTrig = True 'Trigger the acquisition

The TrigSource property specifies the source of the trigger event. When the trigger source is Software, the property SoftTrig must be set to true to trigger the acquisition. When set to TTL or Analog, external events are required to trigger the acquisition. Setting the SoftTrig property to True without having set the TrigSource property to Software Trigger will generate the message “Software Trigger Source Not Selected.”

The property TrigSourceRising should be used in conjunction with the TTL and Analog source to select which edge of the external signal should trigger the system. The properties TrigLevel and TrigRefVoltage should be set when using the Analog trigger source to select the level of the analog input voltage on which to trigger.

TrigSourceRising

Access:	Read and write
Valid settings:	True for rising, False for falling
Syntax:	Adc1.TrigSource = 1 'Set the trigger source to TTL Adc1.TrigSourceRising = False 'Trigger on falling edge Adc1.Arm = True 'Arm the acquisition

When the TrigSource property is set to TTL or Analog, the TrigSourceRising property specifies which edge of the external trigger signal will cause the trigger.

UseChanArray

Access:	Read and write
Valid settings:	True to use ChanArray False to use StartChan and EndChan
Syntax:	Adc1.UseChanArray = True For i = 0 to 15 Adc1.chanArray(i) = i : Adc1.gainArray(i) = 0 Next i Adc1.NumChannels = 16 or Adc1.UseChanArray = False Adc1.StartChan = 0 Adc1.EndChan = 3 Adc1.GlobalGain = 0

When UseChanArray is true, ChanArray holds the array of channels used in the scan. When UseChanArray is false, the channels in the scan are defined by the StartChan and EndChan properties. If your application requires a channel scan of non-consecutive channels, or channels repeated in a scan, or if you need to set individual channels to different gains, the ChanArray property should be used instead of the StartChan and EndChan properties.

Up to 512 array elements can be loaded with any channel number in any order. The sample data in the buffer will be in the same order as the channels in the ChanArray. The NumChannels property is used in conjunction with the ChanArray property to tell the custom control which element in the ChanArray holds the last valid channel.

CTR VBX

Counter/Timer Properties				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
Active	A status flag showing if the acquisition is active.	R	Run	True or False
Alarm1	Writes a value to the alarm register #1	W	Run	0 - 65535
Alarm2	Writes a value to the alarm register #2	W	Run	0 - 65535
Arm	Arm the enabled counters to start counting.	W	Run	True to Arm
Buffered	Represents how many scans have been collected and placed in the buffer.	R	Run	0 -4000000
BufferLength	Represents the usable length of the user-allocated arrays.	R/W	Design/Run	1 to 32767
Comp1Enable	Enables/Disables Comparator #1	R/W	Design/Run	True or False
Comp2Enable	Enables/Disables Comparator #2	R/W	Design/Run	True or False
Disarm	Disable the enabled counters from counting	W	Run	True to disarm
DisarmSave	Disable the enabled counters and move their current values to their respective hold registers	W	Run	True to disarm and save
FoutDivider	Selects the divider of the selected source before outputting the signal on fout	R/W	Design/Run	0 -Divide by 16 1 -Divide by 1 2-Divide by 2 3-Divide by 3 4-Divide by 4 5-Divide by 5 6-Divide by 6 7-Divide by 7 8-Divide by 8 9-Divide by 9 10-Divide by 10 11-Divide by 11 12-Divide by 12 13-Divide by 13 14-Divide by 14 15-Divide by 15
FoutSource	Specifies the frequency output source	R/W	Design/Run	0 - Fout Disabled 1-Counter 1 Input 2-Counter 2 Input 3-Counter 3 Input 4-Counter 4 Input 5-Counter 5 Input 6-Counter 1 Gate 7-Counter 2 Gate 8-Counter 3 Gate 9-Counter 4 Gate 10-Counter 5 Gate 11-1 MHz Clk 12-100 kHz Clk 13-10 kHz Clk 14-1 kHz Clk 15 -100 Hz Clk
FreqCnt	Specifies the number of counts accumulated in the gating interval	R	Run	0 - 65535
FreqCntSource	Specifies which external input to read frequency from	R/W	Design/Run	1-Counter 1 Input 2-Counter 2 Input 3-Counter 3 Input 4-Counter 4 Input 5-Counter 5 Input 6-Counter 1 Gate 7-Counter 2 Gate 8-Counter 3 Gate 9 -Counter 4 Gate
FreqInterval	Specifies the gating interval in which to compute frequency	R/W	Design/Run	1 - 32767 milliseconds
Load	Load the initial counter values of the enabled counters with their respective load or hold registers	W	Run	True to activate
LoadArm	Load the initial counter values of the enabled counters with their respective	W	Run	True to activate

Counter/Timer Properties				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
	load or hold registers and enable the counters to start counting.			
NumScans	The number of scans to collect	R/W	Design/Run	1 - 32767, or -1 for infinite cycle
ReadCounters	Initiate reading of the values of the specified counters in the background using interrupts or stop the current background reading.	W	Run	True to initiate False to stop
Save	Transfer the current counter values of the enabled counters to their respective hold registers.	W	Run	True to activate
SetMasterMode	Set the counters master mode register with the values previously specified in the master mode properties	W	Run	True to activate
SetCounterMode	Set the 9513's Mode register for the specified counter with values previously specified in the set counter properties	W	Run	1-5 signifying the counter number.
TimeOfDay	Enables or Disables the time of day operation	R/W	Design/Run	0 - Disabled 1 - Divide by 5 2 - Divide by 6 3 - Divide by 10
CxBuffer x = 1-5 for counters 1-5	Points to the user buffer for the incoming data from ctr #x or 0 if ctr #x is not to be read	W	Run	The 0th element of a user-dimensioned integer array or 0
CxCntDir x = 1-5 for counters 1-5	Selects whether ctr #x will count up or down	R/W	Design/Run	0 - down 1 - up
CxCntEdge x = 1-5 for counters 1-5	Selects whether ctr #x will count when it receives a rising or falling edge on its count source	R/W	Design/Run	0 - neg. count edge 1 - pos. count edge
CxCntRepeat x = 1-5 for counters 1-5	Enables/Disables rearming ctr #x after terminal count occurs	R/W	Design/Run	True or False
CxCntSource x = 1-5 for counters 1-5	Selects the source used as input to ctr #x	R/W	Design/Run	0 - TC toggled output of last ctr 1 - Counter 1 Input 2 - Counter 2 Input 3 - Counter 3 Input 4 - Counter 4 Input 5 -Counter 5 Input 6 - Counter 1 Gate 7 - Counter 2 Gate 8 - Counter 3 Gate 9 - Counter 4 Gate 10 -Counter 5 Gate 11 - 1 MHz Clk 12 - 100 kHz Clk 13 - 10 kHz Clk 14 - 1 kHz Clk 15 - 100 Hz Clk
CxCntType x = 1-5 for counters 1-5	Select binary or BCD counting for ctr #x	R/W	Design/Run	0 - Binary 1 -BCD
CxEnable x = 1-5 for counters 1-5	Enables/Disables ctr #x to respond to the Arm, Disarm, DisarmSave, Load, LoadArm and Save properties	R/W	Design/Run	True or False
CxGateCtrl x = 1-5 for counters 1-5	Selects how ctr #x will use its gate input or another counter's gate input	R/W	Design/Run	0 - Gating Disabled 1 - Level Hi of TC toggled output of last ctr 2 - Level Hi of gate of next ctr 3 -Level Hi of gate of last ctr 4 - Level Hi of gate of this ctr 5 - Level Lo of gate of this ctr 6 - Rising edge of gate of this ctr 7 - Falling edge of gate of this ctr
CxHold x = 1-5 for counters 1-5	Reads the value of or writes a value to the hold register of ctr #x	R/W	Run	0 - 65535

Counter/Timer Properties				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
CxLoad x = 1-5 for counters 1-5	Writes a value to the load register of ctr #x	W	Run	0 - 65535
CxOutput x = 1-5 for counters 1-5	Specifies the state of ctr #x's output	R/W	Design/Run	0 - Inactive, always low 1 - High pulse on terminal count 2 - Toggled on terminal count 3 - Inactive, High impedance 4 - Low pulse on terminal count
CxReload x = 1-5 for counters 1-5	Programs ctr #x to reload from its load register or reload from either its hold register or load register	R/W	Design/Run	0 - Reload from Load 1 - Reload from Load or Hold
CxSpecialGate x = 1-5 for counters 1-5	Enables/Disables the special gate for ctr #x	R/W	Design/Run	True or False

Event Routines - CTR

When the CTR.VBX tool is placed on an application form, 2 subroutine stubs are automatically created. They are:

```
Sub CTR1_Triggered( )
Sub CTR1_AcquisitionComplete( )
```

When the system trigger has been satisfied by either an internal or external event, the subroutine CTR1_Triggered is automatically called. Code required to post status or begin the data transfer process can be located in this routine.

The trigger event is monitored in the hardware. It is then passed to the custom control (in the background) during the transfer of the first block of data which has been buffered in the external hardware.

When the acquisition is completely finished, the Active property will become false and the subroutine CTR1_AcquisitionComplete will automatically be called.

CTR.VBX Note

When using the counter/timer control, the file DBK.BAS must be added to your application. This file contains the function declaration, "addressof", which gets the address of a VB integer array. The Buffer property accepts this pointer.

Example:

```
Dim MyData(1000)As Integer
        'The Following line would typically be placed in the Form_Load subroutine.
Ctr1.C1Buffer = addressof(MyData(0))
```

CTR VBX Properties

Active

Access:	Read only
Valid setting:	True or False
Syntax:	If Ctr1.Active = False then MsgBox “The acquisition is inactive”

The Active property serves as a status flag to show the state of the armed acquisition. At run-time the Active property returns True to signify that the acquisition is still active, and false is inactive. This property is useful when the state of the acquisition is in question. Upon completion of any acquisition, the Acquisition_Complete routine is automatically called. If polling the acquisition is preferred, use the Active property.

Alarm1, Alarm2

Access:	Write only
Valid setting:	0 - 65535
Syntax:	Ctr1.Comp1Enable = True ’ Enable comparator #1 Ctr1.Alarm1 = 1000 ’ Set the Alarm register #1 to 1000 Ctr1.SetMasterMode = True

Specifies the value to write to the alarm registers 1 or 2, respectively. These alarm registers are only used if the corresponding comparators are enabled with the Comp1Enable and Comp2Enable properties. The operation of these registers is described under the SetMasterMode property.

Arm

Access:	Write only
Valid setting:	True to arm, False does nothing
Syntax:	Ctr1.C1Enable = True Ctr1.C3Enable = True Ctr1.Arm = True ’ Arm counters 1 and 3

Arm enables one or more counters to start counting. Setting this property to true will simultaneously start all counters that have the CnEnable (n=1 to 5) property set to true. This is usually done after all the other properties of the corresponding counters have been set. Setting this property to false does nothing.

Buffered

Access:	Read only
Valid settings:	0 - 4000000

Syntax:	Static ScansProcessed as Long
	If ScansProcessed Ctr1.Buffered then
	    Call moveNewScan
	End if

During an acquisition, the buffer specified by the CxBuffer (x=1 to 5) property is filled with incoming data. The Buffered property holds the number of buffered scans that are presently valid in the integer array. A scan of the counters consists of one reading from each of the counters configured with a non-zero CxBuffer property.

If the NumScans property is set to -1, the acquisition is in Cycle mode, collecting an infinite number of scans. In this mode, the integer array will be modeled as a circular buffer, starting at the beginning as the end is reached. The program is responsible for moving the data to a new location, to disk for example, before old data is overwritten. When in Cycle mode, the Buffered property can exceed the size of the buffer by many times since it is keeping track of the total number of scans that have been collected, not just the number that are presently in the buffer. Your program should keep track of the number of scans processed and compare that number with the value of the Buffered property to see if new data is present.

BufferLength

Access:	Read and Write
Valid setting:	1 to 32767
Syntax:	Dim arrayBuffer(1000) as integer Ctr1.C1Buffer = addressOf(arrayBuffer(0)) Ctr1.BufferLength = 1000

Represents the usable length of the user-allocated integer array that was assigned to the CxBuffer (x=1 to 5) property. Assigning the correct value to BufferLength keeps the acquisition from accidently overrunning the end of the dimensioned array. If more than one buffer has been defined by the CxBuffer property and the buffers are not dimensioned to be the same size, the BufferLength should be set to the size of the smallest buffer. When the NumScans property is set to -1 (infinite cycle mode), the BufferLength property is used by the control to know when to wrap to the beginning of the buffer.

Comp1Enable, Comp2Enable

Access:	Read and Write
Valid setting:	True or False
Syntax:	Ctr1.Comp1Enable = True ’ Enable comparator #1 Ctr1.Alarm1 = 1000 ’ Set the Alarm register #1 to 1000 Ctr1.SetMasterMode = True

Enables/disables the use of comparators 1 or 2, respectively. The operation of the comparators is described under the SetMasterMode property.

Disarm

Access:	Write only
Valid setting:	True to disarm, False does nothing
Syntax:	Ctr1.C1Enable = True Ctr1.C3Enable = True Ctr1.Disarm = True ’ Disarm counters 1 and 3

Disarm stops one or more counters from counting. Setting this property to true will simultaneously stop all counters that have the CxEnable (x=1 to 5) property set to true. Setting this property to false does nothing.

DisarmSave

Access:	Write only
Valid setting:	True to disarm and save, False does nothing
Syntax:	Ctr1.C1Enable = True Ctr1.C3Enable = True Ctr1.DisarmSave = True ’ Disarm and save counters 1 and 3 count% = Ctr1.C1Hold ’ Read the saved value from counter 1

Disarm stops one or more counters from counting and saves the count values of the counters in the hold register. Setting this property to true will simultaneously stop all counters that have the CxEnable (x=1 to 5) property set to true. This property also saves the count values of those counters to the hold register which can be read using the Hold property. Setting this property to false does nothing.

FoutDivider

Access:	Read and Write			
Valid setting:	0- Divide by 16 1- Divide by 1 2- Divide by 2 3- Divide by 3	4- Divide by 4 5- Divide by 5 6- Divide by 6 7- Divide by 7	8- Divide by 8 9- Divide by 9 10- Divide by 10 11- Divide by 11	12- Divide by 12 13- Divide by 13 14- Divide by 14 15- Divide by 15
Syntax:	Ctr1.FoutSource = 15 ' Select 100Hz internal clock as the Fout source Ctr1.FoutDivider = 5 ' Select a divider of 5 for a 20Hz Fout signal Ctr1.SetMasterMode = True ' Configure the master mode register			

The FoutDivider property selects a divider of the FoutSource property. The source defined by the FoutSource will be divided by this value before outputting the signal on the FOUT line (pin 30 of P3).

FoutSource

Access:	Read and Write		
Valid setting:	Value	Source	P3 Pin
	0	Fout Disabled	N/A
	1	Counter 1 Input	36
	2	Counter 2 Input	19
	3	Counter 3 Input	17
	4	Counter 4 Input	15
	5	Counter 5 Input	13
	6	Counter 1 Gate	37
	7	Counter 2 Gate	18
	8	Counter 3 Gate	16
	9	Counter 4 Gate	14
	10	Counter 5 Gate	12
	11	1 MHz Clk	Internal
	12	100 kHz Clk	Internal
	13	10 kHz Clk	Internal
	14	1 kHz Clk	Internal
	15	100 Hz Clk	Internal
Syntax:	Ctr1.FoutSource = 15 ' Select 100 Hz internal clock as the Fout source Ctr1.FoutDivider = 5 ' Select a divider of 5 for a 20 Hz Fout signal Ctr1.SetMasterMode = True ' Configure the master mode register		

The FoutSource property selects the source of the FOUT line (pin 30 of P3). This source will then be divided by the value set by the FoutDivider property and output onto the FOUT line. Setting this property to 0 will disable the FOUT signal. The possible inputs include each of the 5 counter input or gate lines or one of 5 internal clock frequencies. See the following table for a complete listing of sources and pin numbers.

FreqCnt

Access:	Read only
Valid setting:	0 - 65535
Syntax:	Ctr1.FreqSource = 1 ' Select counter 1 input as the frequency source Ctr1.FreqInterval = 100 ' Set the frequency interval to 100ms count% = Ctr1.FreqCnt ' Read the number of counts during this interval freq! = count% * 1000 / Ctr1.FreqInterval ' Calculate the frequency

Contains the number of counts accumulated from the frequency source during the gating interval. The frequency source is specified by the FreqCntSource property and the gating interval is specified in milliseconds by the FreqInterval property. The actual frequency can be calculated by multiplying 1000 times the value of FreqCnt and dividing by the gating interval in milliseconds. **Note**: The counter 4 output (pin 32 of P3) must be externally connected to the counter 5 gate (pin 12 of P3). Reading this property will reconfigure counters 4 and 5.

FreqCntSource

Access:	Read and Write
Valid setting:	Value Source P3 Pin 1 Counter 1 Input 36 2 Counter 2 Input 19 3 Counter 3 Input 17 4 Counter 4 Input 15 5 Counter 5 Input 13 6 Counter 1 Gate 37 7 Counter 2 Gate 18 8 Counter 3 Gate 16 9 Counter 4 Gate 14
Syntax:	Ctr1.FreqSource = 1 ’ Select counter 1 input as the frequency source Ctr1.FreqInterval = 100 ’ Set the frequency interval to 100ms count% = Ctr1.FreqCnt ’ Read the number of counts during this interval freq! = count% * 1000 / Ctr1.FreqInterval ’ Calculate the frequency

Specifies which external input will be used when the FreqCnt property is read. See the FreqCnt property for a complete description of reading frequencies using the FreqCntSource property.

FreqInterval

Access:	Read and Write
Valid setting:	1 - 32767 milliseconds
Syntax:	Ctr1.FreqSource = 1 ’ Select counter 1 input as the frequency source Ctr1.FreqInterval = 100 ’ Set the frequency interval to 100ms count% = Ctr1.FreqCnt ’ Read the number of counts during this interval freq! = count% * 1000 / Ctr1.FreqInterval ’ Calculate the frequency

Specifies the gating interval in milliseconds that will be used when the FreqCnt property is read. See the FreqCnt property for a complete description of reading frequencies using the FreqInterval property.

Load

Access:	Write only
Valid setting:	True to arm, False does nothing
Syntax:	Ctr1.C1Enable = True Ctr1.C3Enable = True Ctr1.Load = True ’ Load the count values of counters 1 and 3

Load initializes the count value of one or more counters. Setting this property to true will simultaneously load the initial count of all counters that have the CxEnable (x=1 to 5) property set to true. This initial count value is set using the CxLoad property. Setting this property to false does nothing.

LoadArm

Access:	Write only
Valid setting:	True to arm, False does nothing
Syntax:	Ctr1.C1Enable = True Ctr1.C3Enable = True Ctr1.LoadArm = True ’ Load and arm counters 1 and 3

LoadArm initializes the count value of one or more counters and enables them to start counting. Setting this property to true will simultaneously initialize the count values and start all counters that have the CxEnable (x=1 to 5) property set to true. The initial count value is set using the CxLoad property. This property is usually used after all the other properties of the corresponding counters have been set. Setting this property to false does nothing.

NumScans

Access:	Read and Write
Valid setting:	1 - 32767 -1 for infinite cycle
Syntax:	Ctr1.NumScans = 1000 ’ collect 1000 scans

The number of scans to collect when the ReadCounters property is set to true. If NumScans is set to -1, the acquisition will continue until it is disarmed by setting the ReadCounters property to false. When the BufferLength is reached, the buffer will wrap around to the beginning, overwriting the oldest scans. In this case, your application should monitor the Buffered property and move the data to other destinations before it is overwritten.

ReadCounters

Access:	Write only
Valid setting:	True to start reading the counters False to stop reading the counters
Syntax:	Ctr1.ReadCounters = True ’ Start the acquisition Ctr1.ReadCounters = False ’ Stop the acquisition

ReadCounters enables/disables reading the values of one or more counters in the background using interrupts. Setting this property to true will enable the background acquisition on interrupts. An interrupts will occur on a rising transition on the interrupt input (pin 1 of P3) if the interrupt enable line (pin 2 of P3) is pulled low. When an interrupt occurs the count value of all counters that have a data buffer configured using the CxBuffer (x=1 to 5) property will be stored in the next available location of the data buffer.

This acquisition uses the NumScans and BufferLength properties to define the size of the data buffer and the CxBuffer to define the location of the data buffer and which counter to read from. The Buffered and Active properties can be used to monitor the state of the acquisition. The CTR1_Triggered subroutine will be called after the first scan is read from the counters and the CTR1_AcquisitionComplete subroutine will be called when the acquisition is finished.

Save

Access:	Write only
Valid setting:	True to save, False does nothing
Syntax:	Ctr1.C1Enable = TrueCtr1.C3Enable = TrueCtr1.Save = True ’ Save counters 1 and 3count% = Ctr1.C1Hold ’ Read the saved value from counter 1

Save transfers the count values of one of more counters to their corresponding hold registers. Setting this property to true will simultaneously save the count value to the hold register for all counters that have the CxEnable (x=1 to 5) property set to true. The hold register which can be read using the CxHold property. Setting this property to false does nothing.

SetMasterMode

Access:	Write only
Valid setting:	True to set the master mode register, False does nothing
Syntax:	Ctr1.FoutSource = 15 ’ Select 100 Hz internal clock as the Fout source Ctr1.FoutDivider = 5 ’ Select a divider of 5 for a 20 Hz Fout signal Ctr1.SetMasterMode = True ’ Configure the master mode register

Set the master mode register with the values previously set by the FoutDivider, FoutSource, Comp1Enable, Comp2Enable and TimeOfDay properties.

SetCounterMode

Access:	Write only
Valid setting:	1-5 signifying the counter number.
Syntax:	Ctr1.SetCounterMode = 1 ' Set counter 1 mode register

Set the specified counter's mode register with values previously set by the CxCntDir (x=1 to 5), CxCntEdge, CxCntRepeat, CxCntSource, CxCntType, CxGateCtrl, CxOutput, CxReload, and CxSpecialGate properties.

TimeOfDay

Access:	Read/Write
Valid setting:	Value Description 0 Disabled 1 Divide by 5 2 Divide by 6 3 Divide by 10
Syntax:	Ctr1.TimeOfDay = 3 ' Select divide by 10 Ctr1.SetMasterMode = True ' Configure master mode register

The TimeOfDay property enables/disables the time of day operation of the counters. This operation is a special mode which causes counters 1 and 2 to turn over at counts that generate 24-hour time-of-day accumulations. (See figure below.) The resolution of the time-of-day operation is 0.1 seconds. To use the time-of-day mode, counter 1 must be configured for a 100Hz, 60Hz or 50Hz source (internal or external) and the TimeOfDay property must be set to Divide by 10, 6 or 5 respectively. This will produce the 10Hz clock source needed to drive the time-of-day clock. The hold registers of counters 1 and 2 will hold the 24-hour time.

Counter 2															
C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0
(2)				(3)				(5)				(9)			
Hours								Minutes							

Counter 1															
C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0
(5)				(9)				(9)							
Hours								1/10 second				+5,6,10			

The following steps must be performed to use the time-of-day mode:

1. Set the TimeOfDay property to Divide by 5, 6 or 10.
2. Set the SetMasterMode property to true to configure the master mode register with the properties set in step 1.
3. Set C1GateCtrl=0 (no gating), C1CntEdge=1 (rising edge), C1SpecialGate=False, C1Reload=0 (reload from load), C1CntRepeat=True (count repetitively), C1CntType=1 (BCD), C1CntDir=1 (count up).
4. Set the SetCounterMode to 1 to configure counter 1 with the properties set in step 3.
5. Set up counter 2 the same as counter 1 except that C2CntSource=0 (TC output of last counter).
6. Set the SetCounterMode to 2 to configure counter 2 with the properties set in step 5.
7. Set C1Load and C2Load to 0
8. Initialize the current 24-hour time-of-day setting according to the figure above by setting C1Load and C2Load again.
9. Load and arm counters 1 and 2 using the C1Enabled, C2Enabled and LoadArm properties.

C1Buffer, C2Buffer, C3Buffer, C4Buffer, C5Buffer

Access:	Write only
Valid settings:	A pointer to the 0th element of a user-dimensioned integer array to read from the counter or 0 to disable reading from the counter.
Syntax:	Dim arrayBuffer(1000) as integer Ctr1.C1Buffer = addressOf(arrayBuffer(0)) ’ Read from counter 1 into arrayBuffer Ctr1.C2Buffer = 0 ’ Do not read from counter 2

The CTR VBX collects all readings in the background under interrupt control. Acquired data is placed directly into a user-dimensioned VB integer array. For each counter to be read in the background, the CxBuffer property should be assigned the value of the pointer to the integer array. For other counters which will not be read from, the CxBuffer property should be set to 0. Once dimensioned, the pointer to the integer array is yielded from the function call “addressOf”, supplied in the file DBK.BAS.

The data in the integer array can be accessed concurrently with the acquisition. The number of valid scans in the integer array can be queried using the Buffered property.

Before the ReadCounters property is enabled, the CxBuffer properties must be assigned pointers to a valid, dimensioned integer array. The dimensioned array must remain valid during the entire background acquisition. If the array is dimensioned within a subroutine using the ReDim command, this array will be deallocated as the program leaves the subroutine. If the acquisition is still active, the acquisition will write over an undefined area of memory. For this reason, the array should be dimensioned as a Global variable.

The dimensioned size of the array should be configured using the BufferLength property.

C1CntDir, C2CntDir, C3CntDir, C4CntDir, C5CntDir

Access:	Read and Write
Valid setting:	0 for counting down, 1 for counting up
Syntax:	Ctr1.C1CntDir = 1 ’ Configure counter 1 to count up Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxCntDir (x=1 to 5) property selects whether the counter will count up or down. The counter is normally configured for down counting when generating a pulse or square wave. The CxLoad property would be set to a positive value which will decrement to zero, defining the duration or width of the waveform. In event counting, the counter would initially be set to 0 and configured to count up. The CxHold property in this case would then contain the number of events received. The SetCounterMode property must be used after setting CxCntDir to configure the desired counters in the mode register.

C1CntEdge, C2CntEdge, C3CntEdge, C4CntEdge, C5CntEdge

Access:	Read and Write
Valid setting:	0 for counting on a falling edge 1 for counting on a rising edge
Syntax:	Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxCntEdge (x=1 to 5) property selects whether the desired counter will count when it receives a rising or falling edge on the source specified by CxCntSource. The SetCounterMode property must be used after setting CxCntEdge to configure the desired counters in the mode register.

C1CntRepeat, C2CntRepeat, C3CntRepeat, C4CntRepeat, C5CntRepeat

Access:	Read and Write
Valid setting:	True enables repetitive counting False disables repetitive counting
Syntax:	Ctr1.C1CntRepeat = True ’ Enable repetitive counting Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxCntRepeat (x=1 to 5) property enables/disables rearming the specified counter after a terminal count (TC) occurs. A terminal count occurs when a down counter reaches 0 or an up counter counts past 65535 in binary count mode or 9999 in BCD count mode. When this TC occurs, the counter can reset the value of the counter to the value contained in the load or hold register and start counting again, or it can disarm itself. Applications such as software retriggerable one-shots would set CxCntRepeat to false so that the one-shot pulse only occurs once after the Arm property is set. Other applications such as rate generators, hardware retriggerable one-shots and square waves would set CxCntRepeat to true so that the counter runs until it is disarmed. The SetCounterMode property must be used after setting CxCntRepeat to configure the desired counters in the mode register.

C1CntSource, C2CntSource, C3CntSource, C4CntSource, C5CntSource

Access:	Read and Write		
Valid setting:	**Value**	**Source**	**P3 Pin #**
	0	TC output of last counter	N/A
	1	Counter 1 Input	36
	2	Counter 2 Input	19
	3	Counter 3 Input	17
	4	Counter 4 Input	15
	5	Counter 5 Input	13
	6	Counter 1 Gate	37
	7	Counter 2 Gate	18
	8	Counter 3 Gate	16
	9	Counter 4 Gate	14
	10	Counter 5 Gate	12
	11	1 MHz Clk	Internal
	12	100 kHz Clk	Internal
	13	10 kHz Clk	Internal
	14	1 kHz Clk	Internal
	15	100 Hz Clk	Internal
Syntax:	Ctr1.C1CntSource = 15 ’ Select the 100Hz internal clock sourceCtr1.SetCounterMode = 1 ’ Program counter 1 mode		

The CxCntSource (x=1 to 5) selects the source which the specified counter will count. This source can be any one of the counter input or gate pins or one of five internal clocks including 1MHz, 100kHz, 10kHz, 1kHz and 100Hz. The source can also be configured to be the terminal count (TC) of the previous counter. The input or gate pins are commonly used for counting events from an external source. The internal clock can be used to generate square wave and rate generators. The TC of the previous counter occurs when the previous counter reaches 0 (down counting), 65535 (binary up counting) or 9999 (BCD up counting). This allows counters to be concatenated internally rather than externally. For example, counter 1 could be configured to count an external input and counter 2 could be configured to count counter 1. This would make the combination of counters 1 and 2 appear to be a single 32-bit counter without any external connection. Counter 5 in this mode is adjacent to counter 1.

The SetCounterMode property must be used after setting CxCntSource to configure the desired counters the mode register.

C1CntType, C2CntType, C3CntType, C4CntType, C5CntType

Access:	Read and Write
Valid setting:	0 for binary counting 1 for BCD counting
Syntax:	Ctr1.C1CntType = 1 ’ Configure binary counting Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxCntType selects binary or BCD counting for the specified counter. Binary counting uses a single 16-bit integer ranging from 0 to 65535. BCD (binary coded decimal) counting uses four 4-bit numbers each of which ranges from 0 to 9, so that the whole16-bit integer will have a range of 0 to 9999. The SetCounterMode property must be used after setting CxCntType to configure the desired counters the mode register.

C1Enable, C2Enable, C3Enable, C4Enable, C5Enable

Access:	Read and Write
Valid setting:	True or False
Syntax:	Ctr1.C1Enable = True Ctr2.C2Enable = False Ctr1.C3Enable = True Ctr2.C4Enable = False Ctr2.C5Enable = False Ctr1.Arm = True ’ Arm counters 1 and 3

The CxEnable (x=1 to 5) enables/disables the specified counter to respond to the Arm, Disarm, DisarmSave, Load, LoadArm and Save properties.

C1GateCtrl, C2GateCtrl, C3GateCtrl, C4GateCtrl, C5GateCtrl

Access:	Read/Write
Valid setting:	Value Gate Control 0 Gating Disabled 1 Level Hi of TC toggled output of last ctr 2 Level Hi of gate of next ctr 3 Level Hi of gate of last ctr 4 Level Hi of gate of this ctr 5 Level Lo of gate of this ctr 6 Rising edge of gate of this ctr 7 Falling edge of gate of this ctr
Syntax:	Ctr1.C1GateCtrl = 0 ’ Disable gating Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxGateCtrl (x=1 to 5) property selects how the specified counter uses its gate pin or the gate pin of the previous or next counter. If gating is disabled, the counter will count when armed regardless of the state of the gate pins. If a level gate is selected, the counter will only count while it is armed and the desired level is applied to the proper gate pin. These settings include a level high on the specified counter, the next counter or the previous counter, or a level low on specified counter. If a falling or rising edge on this counter is selected, the counter will not operate until it receives the desired transition on its corresponding gate pin while it is armed. The final gate control is level high on TC toggled output of last counter. In this mode the current counter will run only when the TC toggled output of the previous counter is high. See the CxOutput property for a complete description of TC toggled output. Counter 1 is adjacent to counter 5 when using the previous or next counter.

The SetCounterMode property must be used after setting CxCntGateCtrl to configure the desired counters in the mode register.

C1Hold, C2Hold, C3Hold, C4Hold, C5Hold

Access:	Read and Write
Valid setting:	0 - 65535
Syntax:	Ctr1.C1Enabled = True Ctr1.Save = True ’ Save count value to the hold register count% = Ctr1.C1Hold ’ Read the hold register of counter 1 Ctr1.C1Hold = 1000 ’ Set the hold register of counter 1 to 1000

The CxHold (x=1 to 5) reads or sets the value of the hold register of the specified counter. When event counting, this register can be used to view the current count value of the counter without disturbing the counting in progress. This is done using the Save or DisarmSave property. This register can also be used to initialize the value of the counter when the CxReload property is set to reload from load or hold. See the CxReload for a complete description of when this register is used.

C1Load, C2Load, C3Load, C4Load, C5Load

Access:	Write only
Valid setting:	0 - 65535
Syntax:	Ctr1.C1Hold = 1000 ’ Set the hold register of counter 1 to 1000

The CxLoad (x=1 to 5) sets the value of the load register of the specified counter. This register is used to initialize the value of the counter when the Load or LoadArm property is set to true. It is also used when the CxCntRepeat property is set for repetitive counting and the counter reaches its terminal count.

C1Output, C2Output, C3Output, C4Output, C5Output

Access:	Read and Write
Valid setting:	Value Output 0 Inactivealways low 1 High pulse on terminal count 2 Toggled on terminal count 3 Inactive, High impedance 4 Low pulse on terminal count
Syntax:	Ctr1.C1Output = 2 ’ Select the output to be TC toggled Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxOutput (x=1 to 5) property controls the output line of the specified counter. The output line of each counter can be disabled and either forced low or put into a high impedance state. It can also be configured to go high or low during a terminal count (TC). A terminal count occurs when a counter reaches 0 by counting down past 1 or up counts past 65535 in binary count mode or 9999 in BCD count mode. Finally, the output can be configured to toggle after a TC. This mode is used to generate variable duty cycle square waves.

The SetCounterMode property must be used after setting CxOutput to configure the desired counters in the mode register.

C1Reload, C2Reload, C3Reload, C4Reload, C5Reload

Access:	Read and Write
Valid setting:	0 for Reload from Load 1 for Reload from Load or Hold
Syntax:	Ctr1.C1Reload = 0 ’ Reload from load only Ctr1.SetCounterMode = 1 ’ Program counter 1 mode

The CxReload (x=1 to 5) property selects whether the specified counter reloads its count value from just the load register or from either the load or hold register. The actual reloading of the counter is related to the values of the CxSpecialGate and CxGateCtrl properties. If the reload property is set to reload from load, the counter will always use the load register when it needs to reload the counter. This usually occurs when the counter is configured for repetitive counting and it reaches a count of 0. If the reload property is set to reload from load or hold, the counter will sometimes use the load register and sometimes the hold register for reloading depending on the CxSpecialGate setting. See the CxSpecialGate setting for a description of this.

The SetCounterMode property must be used after setting CxReload to configure the desired counters in the mode register.

C1SpecialGate, C2SpecialGate, C3SpecialGate, C4SpecialGate, C5SpecialGate

Access:	Read and Write
Valid setting:	True or False
Syntax:	Ctr1.SpecialGate = 0 ’ Disable the special gate Ctr1.SetCounterMode = 1’ Program counter 1 mode

The CxSpecialGate (x=1 to 5) property enables/disables the special gating operation of the specified counter. If the special gate is disabled and the CxReload is set to reload from load, the counter will reload itself from the load register. If the CxReload is set to reload from load and hold, the counter will toggle between reloading from the load and hold registers. If the special gate is enabled and the CxReload property is set to reload from load, an active edge on the gate will cause the counter to save the count value in the hold register and reload the counter with the load register. If the CxReload property is set to reload from load or hold, the gate will control which register is used. If the gate is low during a terminal count the load register will be used to reload the counter, and if the gate is high the hold register will be used.

The SetCounterMode property must be used after setting CxSpecialGate to configure the desired counters in the mode register.

DAC VBX

D/A PROPERTIES				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
ChVoltage(0	Specifies the voltage value to output to D/A channel 0	W	Run	0 -4095 (0 - 5 Volts
ChVoltage(1	Specifies the voltage value to output to D/A channel 1.	W	Run	0 -4095 (0 - 5 Volts

Event Routines - DAC

None.

DAC Properties

ChVoltage(i)

Access:	Write only
Valid settings:	0 - 4095, representing 0 - 5 Volts when the reference is set to -5.
Syntax:	Dac1.ChVoltage(0) = 2000 Dac1.ChVoltage(1) = 1000

Specifies the voltage value to output to D/A channel i. The integer value from 0 to 4095 is used to vary the output voltage of the 12-bit D/As. The voltage varies from 0 to 5 volts when the on-board jumper is set to Internal Reference. The following equation converts volts to counts.

counts = (4095 / (-voltageRef)) * desiredVoltage

For example, if the internal voltage reference of -5 volts is used and 3 volts is required on the output, the count value to assign to the property would be calculated per the following equation:

counts = (4095 / (-(-5))) * 3

Dac1.ChVoltage(0) = counts

DIO VBX

DIGITAL I/O PROPERTIES				
Property	**Description**	**R/W**	**Access**	**Valid Settings**
Local	Specifies whether the local chip is available. (not available means expansion chips are available)	R/W	Design/Run	True = local False = expansion
LocalxSetAsInput x = A or B	Specifies whether local port x is to be configured as input or output	R/W	Design/Run	True = input False = output
LocalxByte x = A or B	Set or read the byte on local port x	R/W	Design/Run	0 - 255
LocalxBit(i) x = A or B i = 0 to 7	Set or read one of the 8 bits on local port x	R/W	Run	0 or non-0
LocalCHiSetAsInput	Specifies whether local port C - High nibble is to be configured as input or output	R/W	Design/Run	True = input False = output
LocalCHiNibble	Set or read the 4 bits on local port C, high nibble	R/W	Design/Run	0 - 15
LocalCHiBit(i) i = 0 to 3	Set or read one of the 4 bits on local port C, high nibble	R/W	Run	0 or non-0
LocalCLoSetAsInput	Specifies whether local port C - Low nibble is to be configured as input or output	R/W	Design/Run	True = input False = output
LocalCLoNibble	Set or read the 4 bits on local port C, low nibble	R/W	Design/Run	0 - 15
LocalCLoBit(I) I = 0 to 3	Set or read one of the 4 bits on local port C, low nibble	R/W	Run	0 or non 0
ExpASetAsInput(i) i = 0 to 7	Specifies whether expansion port A(i) is to be configured as input or output	R/W	Run	True = input False = output
ExpAByte(i) i = 0 to 7	Set or read the byte on expansion port A(i	R/W	Run	0 -255
ExpBSetAsInput(i) i = 0 to 7	Specifies whether expansion port B(i) is to be configured as input or output	R/W	Run	True = input False = output
ExpBByte(i) i = 0 to 7	Set or read the byte on expansion port B(i	R/W	Run	0 -255
ExpCHiSetAsInput(i) i = 0 to 7	Specifies whether expansion port C(i) - High nibble is to be configured as input or output	R/W	Run	True = input False = output
ExpCHiNibble(i) i = 0 to 7	Set or read the 4 bits on expansion port C(i), high nibble	R/W	Run	0 - 15
ExpCLoSetAsInput(i) i = 0 to 7	Specifies whether expansion port C(i) - Low nibble is to be configured as input or output	R/W	Run	True = input False = output
ExpCLoNibble(i) i = 0 to 7	Set or read the 4 bits on expansion port C(i), low nibble	R/W	Run	0 - 15
InitLocalPorts	Configures the local I/O ports then outputs the values specified in the properties window to their corresponding output ports. Note that this is only necessary if the local output ports are expected to be valid when switching from design to run mode.	W	Run	True to activate

Event Routines - DIO

None.

DIO Properties

Local

Access:	Read and write
Valid settings:	True = local False = expansion
Syntax:	Dio1.Local = True 'The local ports on P2 are being used.

Specifies whether the local ports or expansion ports are being used. If a DBK20 or 21 is being used, set this property to false and use the Exp___ properties to control the ports. If the local ports are being used (no DBK20 or 21s), then set this property to true and use the Local___ properties to control the ports.

LocalASetAsInput, LocalBSetAsInput, LocalCHiSetAsInput, LocalCLoSetAsInput

Access:	Read or write
Valid settings:	True = input False = output
Syntax:	Dio1.LocalASetAsInput = True ’Configure local port A as input theValue = Dio1.LocalAByte ’Get the byte value from input port A

Specifies whether the specified local port is to be configured as input or output. The setting of any of these properties automatically sets the output value of all of the ports to zero. These properties should only be used when the local ports are being exercised. If expansion cards DBK20 or DBK21 are being used, the property Local should be set to False, and the Exp___ properties should be used.

LocalAByte, LocalBByte, LocalCHiNibble, LocalCLoNibble

Access:	Read or write
Valid settings:	0 to 255 for byte properties 0 to 15 for nibble properties.
Syntax:	Dio1.LocalASetAsInput = True ’Configure local port A as input Dio1.LocalBSetAsInput = False ’Configure local port B as output theValue = Dio1.LocalAByte ’Get the byte value from input port A Dio1.LocalBByte = 255 ’Set all bits high on port B

These properties set or read the bytes on the local digital I/O on P2. The C port is configured as 2 independent 4-bit nibbles. Each nibble can be independently set as an input or output. The byte values set or read should be between 0 and 255, the nibble values can be between 0 and 15. Bits above the 8th for byte values and bits above the 4th for nibble values will be ignored when assigning values to these properties. These properties should only be used when the local ports are being exercised. If expansion cards DBK20 or DBK21 are being used, the property Local should be set to False, and the Exp___ properties should be used.

LocalABit(i), LocalBBit(i), LocalCHiBit(i), LocalCLoBit(i)

Access:	Read or write
Valid settings:	0 for TTL low non-zero for TTL high
Syntax:	Dio1.LocalASetAsInput = True ’Configure local port A as input Dio1.LocalBSetAsInput = False ’Configure local port B as output theBit = Dio1.LocalABit(0) ’Get the bit value of bit 0 on port A Dio1.LocalBBit(0) = 1 ’Set bit 0 on port B to low Dio1.LocalBBit(1) = 0 ’Set bit 1 on port B to high

These properties set or read the bits on the local digital I/O on P2. The C port is configured as 2 independent 4-bit nibbles. Each nibble can be independently set as an input or output. The bit values set or read are zero or non-zero. The i index which ranges from 0 to 7 (or 0 to 3) indicates what bit of the port is to be read or set. These properties should only be used when the local ports are being exercised. If expansion cards DBK20 or DBK21 are being used, the property Local should be set to False, and the Exp___ properties should be used.

ExpASetAsInput(i), ExpBSetAsInput(i), ExpCHiSetAsInput(i), ExpCLoSetAsInput(i)

Access:	Read and write
Valid settings:	True = input, False = output
Syntax:	Dio1.ExpASetAsInput(0) = True 'Configure port A on 1st connector of DBK20 at address A as input Dio1.ExpBSetAsInput(2) = False 'Configure port B on 1st connector of DBK20 at address B as output Dio1.ExpASetAsInput(5) = False 'Configure port A on 2nd connector of DBK20 at address C as output theValue = Dio1.ExpAByte(0) 'Get the value from port A on 1st connector of DBK20 at address A Dio1.ExpBSetAsInput(2) = 255 'Set all bits high on port B on 1st connector of DBK20 at address B Dio1.ExpASetAsInput(5) = 0 'Set all bits low on port A on 2nd connector of DBK20 at address C

Specifies whether the specified expansion port is to be configured as input or output. The setting of any of these properties automatically sets the output value of all of the ports to zero. The i index, set from 0 to 7, specifies the expansion section on the DBK20 and 21s connected. As many as 4 cards can be connected, each with 2 connectors. The jumper on the expansion cards marked "A-B-C-D" allows the user to assign a unique address to each card as follows.

i Index	Card Affected
0	A 1st connector
1	A 2nd connector
2	B 1st connector
3	B 2nd connector
4	C 1st connector
5	C 2nd connector
6	D 1st connector
7	D 2nd connector

These properties should only be used when DBK20 or DBK21 digital expansion cards are used and the Local property is set to false. If no expansion cards used, the property Local should be set to True, and the Local___ properties should be used.

ExpAByte(i), ExpBByte(i), ExpCHiNibble(i), ExpCLoNibble(i)

Access:	Read and write
Valid settings:	0 to 255 for byte properties, 0 to 15 for nibble properties.
Syntax:	Dio1.ExpASetAsInput(0) = True 'Configure port A on 1st connector of DBK20 at address A as input Dio1.ExpBSetAsInput(2) = False 'Configure port B on 1st connector of DBK20 at address B as output Dio1.ExpASetAsInput(5) = False 'Configure port A on 2nd connector of DBK20 at address C as output theValue = Dio1.ExpAByte(0) 'Get the value from port A on 1st connector of DBK20 at address A Dio1.ExpBSetAsInput(2) = 255 'Set all bits high on port B on 1st connector of DBK20 at address B Dio1.ExpASetAsInput(5) = 0 'Set all bits low on port A on 2nd connector of DBK20 at address C

These properties set or read the bytes on the expansion digital I/O from the DBK20 and 21s. The C ports on each card are configured as 2 independent 4-bit nibbles. Each nibble can be independently set as an input or output. The byte values set or read should be between 0 and 255, the nibble values can be between 0 and 15. Bits above the 8th for byte values and bits above the 4th for nibble values will be ignored when assigning values to these properties. The i index, set from 0 to 7, specifies the expansion section on the DBK20 and 21s connected. As many as 4 cards can be connected, each with 2 connectors. The jumper on the expansion cards marked "A-B-C-D" allows the user to assign a unique address to each card as follows.

i IndexCard Affected0A 1st connector1A 2nd connector2B 1st connector3B 2nd connector4C 1st connector5C 2nd connector6D 1st connector7D 2nd connector

These properties should only be used when DBK20 or DBK21 digital expansion cards are used and the Local property is set to false. If no expansion cards are used, the property Local should be set to True, and the Local___ properties should be used.

InitLocalPorts

Access:	Write only
Valid settings:	True to activate
Syntax:	Dio1.InitLocalPorts = True

This property configures the local I/O ports then outputs the values specified in the properties window to their corresponding output ports. Note that this is only necessary if the local output ports are expected to be valid when switching from design to run mode. If the Local byte properties are set in the properties window, these values will not automatically appear on the output ports until the property InitLocalPorts is set to true. This is unnecessary if the local byte properties are assigned in code.

Programming Examples

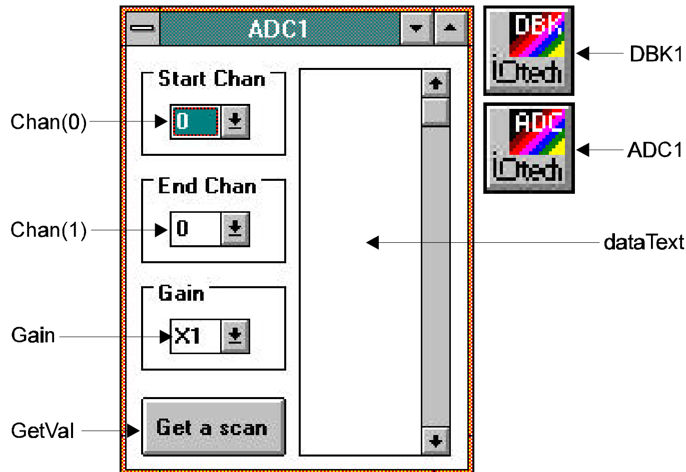
These programming examples the proper use of custom VBX properties. The user interface and elegance are minimized for the sake of clarity. Each example is preceded by a graphic of the application form and listing of the design-time properties of the controls set in the Properties window.

Example Summary

Example	Description	Controls Used	Featured Properties		Page #
ADC1	Analog input of one scan using start and end channel parameters	DBK, ADC	Arm BufferedLength GlobalGain SoftTrig	Buffer EndChan Open StartChan	9-33
ADC2	Analog input of multiple scans using start and end channel, and trigger parameters	DBK, ADC	Active Buffer Buffered EndChan GlobalGain Open StartChan TrigLevel TrigSource	Arm BufferedLength Frequency NumScans SoftTrig TrigRefVoltage TrigSourceRising	9-35
ADC3	Analog input of multiple scans using ChanArray and trigger parameters	DBK, ADC	Arm Buffer BufferLength Frequency NumChannels Open UseChanArray	BipolarArray Buffered ChanArray GainArray NumScans SoftTrig	9-39
ADC4	Analog input direct-to-disk program that uses the input buffer in a continuous circular fashion	DBK, ADC	Arm Buffered BufferOverrun Frequency NumScans (as infinite) SoftTrig TrigLevel TrigSource	Buffer BufferLength EndChan GlobalGain Open StartChan TrigSourceRising	9-42
ADC5	Analog input using expansion cards. Converts counts to volts	DBK, ADC	Active Buffer EndChan Open StartChan	Arm BufferLength GlobalGain SoftTrig	9-47
DAC1	Analog output. Controls both DACs.	DBK, DAC	ChVoltage	Open	9-51
DIO1	Digital I/O. Provides byte-wise I/O to local and expansion ports	DBK, DIO	ExpAByte ExpBByte ExpCHiNibble ExpCLoNibble Local LocalASetAsInput LocalBSetAsInput LocalCHiSetAsInput LocalCLoSetAsInput	ExpASetAsInput ExpBSetAsInput ExpCHiSetAsInput ExpCLoSetAsInput LocalAbyte LocalBbyte LocalCHiNibble LocalCLoNibble Open	9-52
DIO2	Digital I/O. Provides bit-wise I/O for local ports.	DBK, DIO	Local LocalASetAsInput LocalBSetAsInput LocalCHiSetAsInput LocalCLoSetAsInput	LocalABit LocalBBit LocalCHiBit LocalCLoBit Open	9-54

Example	Description	Controls Used	Featured Properties		Page #
CTR1	Counter/Timer. Output variable duty-cycle waveforms.	DBK, CTR	CxCntDir CxCntRepeat CxCntType CxGateCtrl CxLoad CxReload Disarm Open	CxCntEdge CxCntSource CxEnable CxHold CxOutputCtrl CxSpecialGate, LoadArm SetCounterMode	9-59
CTR2	Counter/Timer. Totalize events on the counter inputs.	DBK, CTR	CxCntDir CxCntRepeat CxCntType CxGateCtrl CxLoad CxReload Disarm Load Save	CxCntEdge CxCntSource CxEnable CxHold CxOutputCtrl CxSpecialGate LoadArm Open SetCounterMode	9-62
CTR3	Counter/Timer. Read the frequency of each counter input using low-level counter properties.	DBK, CTR	CxCntDir CxCntRepeat CxCntType CxGateCtrl CxLoad CxReload Disarm Open SetCounterMode	CxCntEdge CxCntSource CxEnable CxHold CxOutputCtrl CxSpecialGate LoadArm Save	9-66
CTR4	Counter/Timer. Configure the source and divider of the Fout pin.	DBK, CTR	FoutDivider Open	FoutSource SetMasterMode	9-68
CTR5	Counter/Timer. Display the elapsed time from the start if the program using the time-of-day operation of the counter.	DBK, CTR	CxCntDir CxCntRepeat CxCntType CxGateCtrl CxReload CxLoad Disarm Open SetCounterModeRSet TimeOfDay	CxCntEdge CxCntSource CxEnable CxOutputCtrl CxSpecialGate CxHold LoadArm Save MasterMode	9-70
CTR6	Counter/Timer. Read the frequency of each counter using the built-in frequency properties.	DBK, CTR	FreqCnt FreqInterval	FreqCntSource Open	9-72
CTR7	Counter/Timer. Totalize events on the counter inputs using a background transfer and write totalized values to disk.	DBK, CTR	Active BufferLength CxCntDir CxCntRepeat CxCntType CxGateCtrl CxOutputCtrl CxSpecialGate LoadArm Open SetCounterMode	Buffered CxBuffer CxCntEdge CxCntSource CxEnable CxLoad CxReload Disarm NumScans ReadCounters	9-75

ADC1

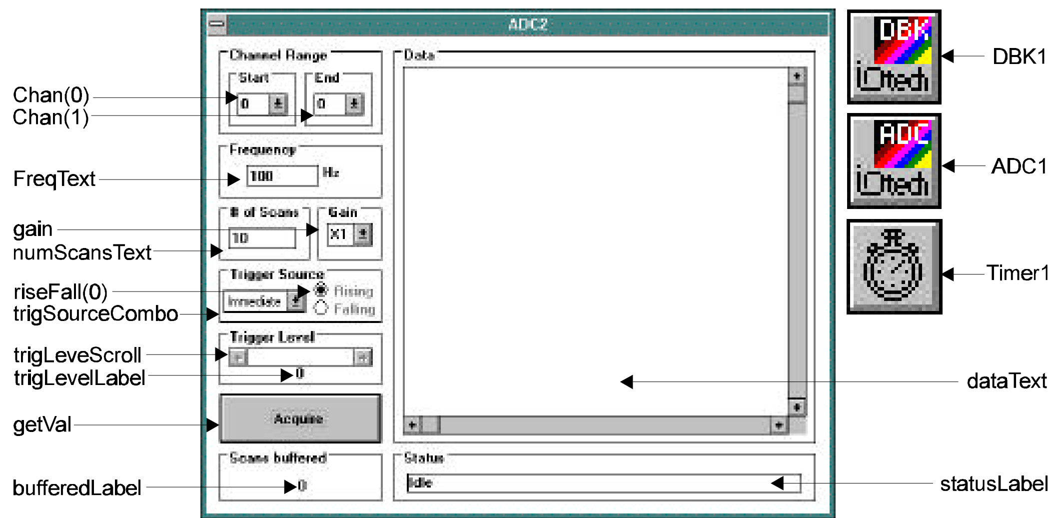


ADC1 Form

```
Begin ADC Adc1
            BufferLength        =       1
            EndChan             =       0
            Frequency           =       10000
            GlobalBipolar       =       0       'False
            GlobalGain          =       0       ' 0 - Base Unit X1
            GlobalSE            =       0       'False
            Left                =       30
            NumChannels         =       1
            NumScans            =       1
            OneShot             =       0       'False
            StartChan           =       0
            Top                 =       165
            TrigLevel           =       0
            TrigRefVoltage      =       -5
            TrigSource          =       0        'Software
            TrigSourceRising    =       -1      'True
            UseChanArray        =       0       'False
End
Begin DBK Dbk1
            IntLevel            =       7
            Left                =       30
            LptPort             =       0       'LPT1
            Protocol            =       1       '4 Bit I/O
            Top                 =       810
End
Const MAXBUF = 100          'The size of my data buffer
Const STARTCH = 0           'Mnemonic to identify the index of the control array
Const ENDCH = 1             'Mnemonic to identify the index of the control array
Dim NL As String            'Used to separate the channels in the text box
Dim dataBuffer(MAXBUF) As Integer
                            'This is the data buffer for all of the analog input data
Sub Adc1_AcquisitionComplete ()
Dim i As Integer
Dim NumberOfChannels As Integer

            'At this point, the scan is already in the array dataBuffer.
            'The following code extracts the data from the integer array,
            'dataBuffer, and places into the TextBox dataText.  So that each
            'channel value occupies one line in the TextBox, a Newline (NL)
            'is placed between each reading.
NumberOfChannels = chan(ENDCH).ListIndex - chan(STARTCH).ListIndex
dataText.Text = Format$(dataBuffer(0))
            For i = 1 To NumberOfChannels
                        dataText.Text = dataText.Text + NL + Format$(dataBuffer(i))
            Next i
End Sub

Sub Chan_Click (Index As Integer)
            'Adjust the StartChan and EndChan properties using the
```

```
        'ListIndex property of the combobox.
        If Index = STARTCH Then
                adc1.StartChan = chan(STARTCH).ListIndex
        Else
                adc1.EndChan = chan(ENDCH).ListIndex
        End If
End Sub

Sub Form_Load ()
        Dim i As Integer

        'Create a NEW LINE string to separate the channels
        NL = Chr$(13) + Chr$(10)

        'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
        adc1.Buffer = addressOf(dataBuffer(0)) 'dataBuffer is has global scope
        adc1.BufferLength = MAXBUF

        'Put channel choices in combos
        For i = 0 To 15
                chan(STARTCH).AddItem Format$(i)
                chan(ENDCH).AddItem Format$(i)
        Next i
        chan(STARTCH).ListIndex = 0
        chan(ENDCH).ListIndex = 0

        'Put gain choices in combo
        gain.AddItem "X1"
        gain.AddItem "X2"
        gain.AddItem "X4"
        gain.AddItem "X8"
        gain.ListIndex = 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub

Sub gain_Click ()
        'Use the combo's lisIndex property to set the GlobalGain.
        'GlobalGain will be used on all the channels in the scan.  To
        'assign independent gains to each channel, use the ChanArray
        'property rather than StartChan and EndChan.
        adc1.GlobalGain = gain.ListIndex
End Sub

Sub GetVal_Click ()
        'Start the acquisition
        adc1.Arm = True

        'Since the trigger source is Software, issue the software
        'trigger command.  If the trigger source is external, this
        'command should not be issued.
        adc1.SoftTrig = True
End Sub
```

ADC2



ADC2 Form

```
Begin ADC Adc1
        BufferLength        =       1
        EndChan             =       0
        Frequency           =       10000
        GlobalBipolar       =       0       'False
        GlobalGain          =       0       ' 0 - Base Unit X1
        GlobalSE            =       0       'False
        Left                =       60
        NumChannels         =       1
        NumScans            =       1
        OneShot             =       0       'False
        StartChan           =       0
        Top                 =       90
        TrigLevel           =       0
        TrigRefVoltage      =       -5
        TrigSource          =       0        'Software
        TrigSourceRising    =       -1      'True
        UseChanArray        =       0       'False
End
Begin DBK Dbk1
        IntLevel            =       7
        Left                =       45
        LptPort             =       0       'LPT1
        Protocol            =       1       '4 Bit I/O
        Top                 =       540
End

Const MAXBUF = 32000        'Set the analog input buffer length
Const STARTCH = 0           'Mnemonic to identify the index in the control array
Const ENDCH = 1             'Mnemonic to identify the index in the control array
Const IMMEDIATE = 0         'Mnemonic to identify trigger source from combo listIndex
Const TTL = 1               'Mnemonic to identify trigger source from combo listIndex
Const ANALOG = 2            'Mnemonic to identify trigger source from combo listIndex

Dim NL As String            '(Newline) used to separate the channels in the text box
Dim dataBuffer(MAXBUF) As Integer 'The data buffer for all analog input operations
Dim bigString As String     'Intermediate string for the data destined for the text box.  It's
                            'faster to manipulate the string data in a string variable
                            'than in a textbox.

Sub Adc1_AcquisitionComplete ()
        'At this point, the collected data is in the integer array, dataBuffer.
        'This subroutine organizes the data in  rows and columns for printing
        'in the textbox.
```

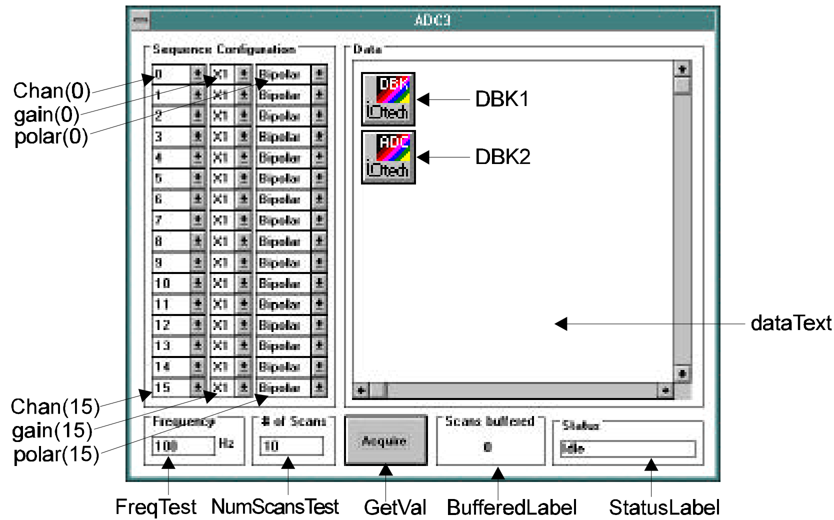
```
        Dim i As Integer
        Dim scan As Integer
        Dim chans As Integer
        Dim unsigned As Long

        'All of the string manipulation takes place in bigString, then
        'bigString is transferred in to the textbox.  This is much faster
        'than manipulating the string directly in the textbox.

        'Put the channel labels across the top
        bigString = Chr$(9)
        For i = chan(STARTCH).ListIndex To chan(ENDCH).ListIndex
                bigString = bigString + "CH" + Format$(i) + Chr$(9)
        Next i
        bigString = bigString + NL

        'Put each scan in a single row, separating the channels with a tab
        'character.  Separate each scan with a newline.
        chans = chan(ENDCH).ListIndex -chan(STARTCH).ListIndex + 1
        For scan = 0 To adc1.Buffered - 1
                bigString = bigString + Format$(scan + 1) + Chr$(9)
                For i = 0 To chans - 1
                        'Since the A/D converter is a full 16 bits, VB's integer type
                        'incorrectly interprets the MSB as a sign bit.  The next
                        'two lines create an unsigned value from 0 to 65535.
                        unsigned = dataBuffer(i + scan * chans)
                        If unsigned  0 Then unsigned = unsigned + 65536
                        bigString = bigString + Format$(unsigned) + Chr$(9)
                Next i
                bigString = bigString + NL
        Next scan
        dataText.Text = bigString

        'Update status textbox
        statusLabel.Caption = "Idle"
End Sub

Sub Adc1_Triggered ()
        'Update the status textbox
        statusLabel.Caption = "Triggered"
End Sub

Sub Chan_Click (Index As Integer)
        'Update the startChan and endChan properties of ADC1.

        'Keep the startChan from being higher that endChan.
        If chan(STARTCH).ListIndex  chan(ENDCH).ListIndex Then
                MsgBox "End channel must be greater than Start channel"
                chan(STARTCH).ListIndex = 0
                chan(ENDCH).ListIndex = 0
                Exit Sub
        End If
        'Start and End combos are indices 0 and 1 in a control array.  Depending
        'on the index argument, update the ADC1 property.
        If Index = STARTCH Then
                adc1.StartChan = chan(STARTCH).ListIndex
        Else
                adc1.EndChan = chan(ENDCH).ListIndex
        End If
End Sub

Sub Form_Load ()
        Dim i As Integer

        'Create a NEW LINE string to separate the channels
        NL = Chr$(13) + Chr$(10)

        'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
        adc1.Buffer = addressOf(dataBuffer(0))

        'Set bufferLength so the ADC VBX can perform boundary checking
        adc1.BufferLength = MAXBUF
```

```
        'Set the default frequency and number of scans
        numScansText.Text = "10"
        freqText = "100"

        'Put channel choices in combos
        For i = 0 To 15
                chan(STARTCH).AddItem Format$(i)
                chan(ENDCH).AddItem Format$(i)
        Next i
        chan(ENDCH).ListIndex = 0
        chan(STARTCH).ListIndex = 0

        'Put gain choices in combo
        gain.AddItem "X1"
        gain.AddItem "X2"
        gain.AddItem "X4"
        gain.AddItem "X8"
        gain.ListIndex = 0

        'Put trigger source choices in combo
        trigSourceCombo.AddItem "Immediate"
        trigSourceCombo.AddItem "TTL"
        trigSourceCombo.AddItem "Analog"
        trigSourceCombo.ListIndex = 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub

Sub FreqText_Change ()
        'Set the frequency property of ADC1
        If freqText.Text  "" Then
                adc1.Frequency = Int(Val(freqText.Text))
        End If
End Sub

Sub FreqText_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39       'All numbers
                Case &H8                'Back space
                Case Else
                        keyascii = 0    'Reject all other characters
        End Select
End Sub

Sub gain_Click ()
        'Set the GlobalGain of ADC1.  To set the gain of each channel independently,
        'use the ChanArray property rather than the startChan and endChan properties.
        adc1.GlobalGain = gain.ListIndex
End Sub

Sub GetVal_Click ()
        'Start the acquisition.

        If getVal.Caption = "Acquire" Then
                'Turn the button into a Disarm button for the duration of the acquisition
                getVal.Caption = "Disarm"

                'Update the status textbox
                statusLabel.Caption = "Waiting for trigger"
                statusLabel.Refresh

                'Arm the system
                adc1.Arm = True

                'If the trigger source in Software, issue the software trigger.
                If trigSourceCombo.ListIndex = 0 Then adc1.SoftTrig = True

                'Enable a timer to read the number of scans that have been collected.
                statusTimer.Enabled = True
        Else
                'Disarm the acquisition.  The statusTimer will sense that the acquisition
```

```
                'has been disabled and will set the caption of the button back to "Acquire".
                adc1.Arm = False
        End If
End Sub

Sub NumScansText_Change ()
        'Set the numberScans property of ADC1.
        If numScansText.Text  "" Then
                adc1.NumScans = Int(Val(numScansText))
        End If
End Sub

Sub NumScansText_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39           'All numbers
                Case &H8                    'Back space
                Case Else
                        keyascii = 0        'Reject all other characters
        End Select
End Sub

Sub RiseFall_Click (Index As Integer)
'Set the trigger edge to rising or falling.

        If riseFall(0).Value = True Then
                adc1.TrigSourceRising = True
        Else
                adc1.TrigSourceRising = False
        End If
End Sub

Sub StatusTimer_Timer ()
        'Post the number of scans that have been collected.

        'If the acquisition is no longer active, disable this timer.
        If adc1.Active = 0 Then
                statusTimer.Enabled = False
                getVal.Caption = "Acquire"
                statusLabel.Caption = "Idle"
        End If

        'Use the ADC1's Buffered property to get the number of buffered scans,
        'then post the number in the label.
        BufferedLabel.Caption = Str$(adc1.Buffered)
End Sub

Sub TrigLevelScroll_Change ()
        'The DaqBook trigger level can range from -5 to +5 volts.
        'The scrollbar min and max are -50 to +50.
        'Dividing the scroll bar value by 10 allows the user 0.1 volt resolution
        'in setting the trigger value.

        'As the scrollbar is operated, post the value in the label below.
        trigLevelLabel.Caption = Format$(trigLevelScroll.Value / 10) + "V"

        'Set the trigLevel property.
        adc1.TrigLevel = trigLevelScroll.Value / 10
End Sub

Sub TrigLevelScroll_Scroll ()
        Call TrigLevelScroll_Change
End Sub

Sub TrigSourceCombo_Click ()
        'Set the trigSource property.
        adc1.TrigSource =  trigSourceCombo.ListIndex

        'If the trigger source is analog, enable the trigger level scrollbar.
        If trigSourceCombo.ListIndex = IMMEDIATE Then
                trigLevelScroll.Enabled = False
                riseFall(0).Enabled = False
                riseFall(1).Enabled = False
        ElseIf trigSourceCombo.ListIndex = ANALOG Then
                trigLevelScroll.Enabled = True
```

```
                    riseFall(0).Enabled = True
                    riseFall(1).Enabled = True
          Else
                    riseFall(0).Enabled = True
                    riseFall(1).Enabled = True
          End If
End Sub
```

ADC3



ADC3 Form

```
Begin ADC Adc1
          BufferLength        =         1
          EndChan             =         0
          Frequency           =         10000
          GlobalBipolar       =         0         'False
          GlobalGain          =         0         ' 0 - Base Unit X1
          GlobalSE            =         0         'False
          Left                =         3360
          NumChannels         =         1
          NumScans            =         1
          OneShot             =         0         'False
          StartChan           =         0
          Top                 =         600
          TrigLevel           =         0
          TrigRefVoltage      =         -5
          TrigSource          =         0          'Software
          TrigSourceRising    =         -1        'True
          UseChanArray        =         -1        'True
End
Begin DBK Dbk1
          IntLevel            =         7
          Left                =         3360
          LptPort             =         0         'LPT1
          Protocol            =         1         '4 Bit I/O
          Top                 =         1080
End
Const MAXBUF = 32000        'The analog input buffer size
Dim NL As String  '(Newline), used to separate channel readings in textbox
Dim dataBuffer(MAXBUF) As Integer
                    'The analog input data buffer integer array
Dim bigString As String        'The string destined for the textbox is manipulated in
                               'bigString.  This is faster than performing character
                               'manipulation in a textbox.
Sub Adc1_AcquisitionComplete ()
          'At this point, the analog input data is in the buffer, dataBuffer.  This subroutine pulls the values
            out of the integer array, formats them, then places them into the textbox for viewing.  Since
```

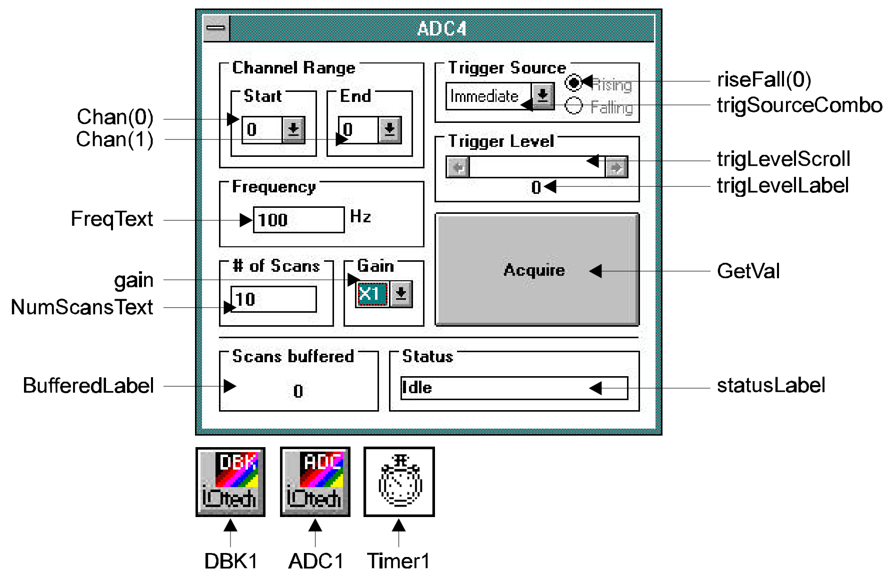
```
            the data destination in this program is a string, the number of scans is limited by the
            maximum string size of VB.
Dim i As Integer
Dim scan As Integer
Dim chans As Integer
  'Put up the hour glass.  Moving the data into the string is time-consuming.
mousePointer = 11
DoEvents
        'Put tabs between channels and newlines between scans, then put entire
        'string into the textbox.
bigString = ""
chans = adc1.NumChannels
For scan = 0 To adc1.Buffered - 1
        bigString = bigString + Format$(scan + 1) + Chr$(9)
        For i = 0 To chans - 1
                bigString = bigString + Format$(dataBuffer(i + scan * chans)) + Chr$(9)
        Next i
                bigString = bigString + NL
        Next scan
        dataText.Text = bigString
                'Update status box
        StatusLabel.Caption = "Idle"
                'Get rid of the hourglass
        mousePointer = 0
End Sub

Sub Adc1_Triggered ()
        'Update the status box
        StatusLabel.Caption = "Triggered"
End Sub

Sub Chan_Click (Index As Integer)
                'Make sure there are no holes in the channel list.
        Dim i As Integer
        'If a channel combo is set to "none", then all channels below should be "none" also.
        If chan(Index).ListIndex = 0 Then
                For i = Index To 15
                        chan(i).ListIndex = 0
                Next i
        Else
                For i = Index To 0 Step -1
                        If chan(i).ListIndex = 0 Then
                                MsgBox "Fill in empty sequencer locations above first"
                                chan(Index).ListIndex = 0
                                Exit Sub
                        End If
                Next i
        End If
End Sub

Sub Form_Load ()
        Dim i As Integer
        Dim l As Integer
                'Create a NEW LINE string to separate the channels
        NL = Chr$(13) + Chr$(10)
                'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
        adc1.Buffer = addressOf(dataBuffer(0))
                'Set bufferLength to all adc1 to check buffer boundaries
        adc1.BufferLength = MAXBUF
                'Set the default frequency and number of scans
        numScansText.Text = "10"
        freqText = "100"
                'Load the combo boxes with channel, gain and pole choices
        For l = 0 To 15
                chan(l).AddItem "None"
                For i = 0 To 15
                        chan(l).AddItem Format$(i)
                Next i
                gain(l).AddItem "X1"
                gain(l).AddItem "X2"
                gain(l).AddItem "X4"
                gain(l).AddItem "X8"
                polar(l).AddItem "Bipolar"
```

```
                polar(I).AddItem "Unipolar"
        Next I
                'Initialize the combos
        For I = 0 To 15
                chan(I).ListIndex = I + 1
                gain(I).ListIndex = 0
                polar(I).ListIndex = 0
        Next I
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub

Sub FreqText_Change ()
                'Set the frequency property of ADC1.
        If freqText.Text  "" Then
                adc1.Frequency = Int(Val(freqText.Text))
        End If
End Sub

Sub FreqText_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39   'All numbers
                Case &H8                        'Back space
                Case Else
                        keyascii = 0            'Reject all other characters
        End Select
End Sub

Sub GetVal_Click ()
'Arm the system
        Dim i As Integer
        If getVal.Caption = "Acquire" Then
                'Set the button caption to "Disarm" for the duration of the acquisition.
                getVal.Caption = "Disarm"
                'Set the chanArray, gainArray, and  BipolarArray properties of ADC1.
                For i = 0 To 15
                'Find out how many channels are configured (not set to NONE).
                        If chan(i).ListIndex = 0 Then
                                If i = 0 Then Exit Sub
                                 'If the 1st channel combo is set to NONE, exit
                                Exit For
                        End If
                        adc1.ChanArray(i) = chan(i).ListIndex - 1
                        adc1.GainArray(i) = gain(i).ListIndex
                        adc1.BipolarArray(i) = polar(i).ListIndex
                Next i
                adc1.NumChannels = i
                        'When NONE is detected, set the NumChannels property
                        'Update the status box
                StatusLabel.Caption = "Waiting for trigger"
                StatusLabel.Refresh

                        'Arm the acquisition
                adc1.Arm = True
                        'Send the software trigger.
                adc1.SoftTrig = True
                        'Enable the timer that checks how many scans have been collected.
                statusTimer.Enabled = True
        Else
        'Disarm the acquisition.  The status timer will detect the disarm and set the button caption back
          to "Acquire"
                adc1.Arm = False
        End If
End Sub

Sub NumScansText_Change ()
'Set the numScans property.
        If numScansText.Text  "" Then
                adc1.NumScans = Int(Val(numScansText))
        End If
End Sub
```

```
Sub NumScansText_KeyPress (keyascii As Integer)
    Select Case keyascii
        Case &H30 To &H39       'All numbers
        Case &H8                    'Back space
        Case Else
            keyascii = 0        'Reject all other characters
    End Select
End Sub

Sub StatusTimer_Timer ()
    If adc1.Active = 0 Then
        statusTimer.Enabled = False
        getVal.Caption = "Acquire"
    End If
    BufferedLabel.Caption = Str$(adc1.Buffered)
        End Sub
```

ADC4

```
Begin ADC Adc1
    BufferLength    =   1
    EndChan         =   0
    Frequency       =   10000
    GlobalBipolar   =   0       'False
    GlobalGain      =   0       ' 0 - Base Unit X1
    GlobalSE        =   0       'False
    Left            =   2580
    NumChannels     =   1
    NumScans        =   -1
    OneShot         =   0       'False
    StartChan       =   0
    Top             =   2520
    TrigLevel       =   0
    TrigRefVoltage  =   -5
    TrigSource      =   0        'Software
    TrigSourceRising =  -1      'True
    UseChanArray    =   0       'False
```



ADC4 Form

```
End
Begin DBK Dbk1
    IntLevel        =   7
    Left            =   3060
    LptPort         =   0       'LPT1
    Protocol        =   1       '4 Bit I/O
    Top             =   2520
End
```

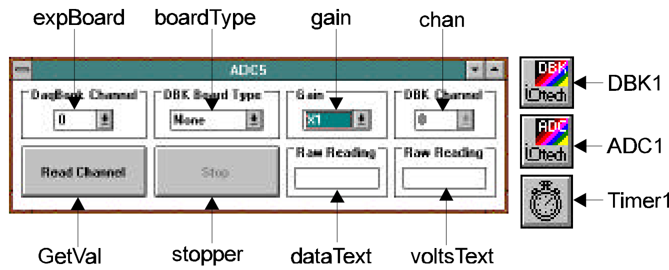
```
Const MAXBUF = 16000        'Maximum buffer size
Const STARTCH = 0           'Index of start channel in control array
Const ENDCH = 1             'Index of end channel in control array
Const IMMEDIATE = 0         'Trigger source  mnemonic
Const TTL = 1               'Trigger source mnemonic
Const ANALOG = 2            'Trigger source mnemonic

Dim dataBuffer(MAXBUF) As Integer   'The data buffer for all incoming data
Dim scansProcessed As Long'keeps track of how many scans in dataBuffer have been sent to disk
Dim scanSize As Integer     'the number of bytes in a scan
Dim MaxBufIndex As Integer  'the max scan index in the buffer
Dim readIndex As Integer    'the array index from which to read the next value
Dim fHandle As Integer      'the file handle of the data destination file
Dim er As Integer           'error return variable
        'fWrite is a Windows API function that allows us to write an integer array to disk quickly
Declare Function fWrite Lib "kernel" Alias "_lwrite" (ByVal hFile As Integer, lpBuff As Any, ByVal wBytes As Integer) As Integer

Sub Adc1_Triggered ()
        'Update the status box.
        statusLabel.Caption = "Triggered"
        'Enable the timer that checks for new data and stores it away.
        collectDataTimer.Enabled = True
End Sub

Sub Chan_Click (Index As Integer)
'Set the startChan and endChan properties of ADC1.
        If chan(STARTCH).ListIndex  chan(ENDCH).ListIndex Then
                MsgBox "End channel must be greater than Start channel"
                chan(STARTCH).ListIndex = 0
                chan(ENDCH).ListIndex = 0
                Exit Sub
        End If
        If Index = STARTCH Then
                adc1.StartChan = chan(STARTCH).ListIndex
        Else
                adc1.EndChan = chan(ENDCH).ListIndex
        End If
End Sub

Sub CollectDataTimer_Timer ()
                'If there is new data, append it to the disk file.
        Dim cnt As Long
        Dim unprocessed As Long
        Dim ints As Long
        Dim firstBufSize As Long
        Dim secondBufSize As Long
                'Check for buffer overrun
        If adc1.BufferOverrun Then
                disarmAcq
                MsgBox "DaqBook FIFO buffer overrun"
                Exit Sub
        End If
                'Get the number of scans collected
        cnt = adc1.Buffered
        'If more scans collected than processed, process the new scans.
        If cnt  scansProcessed Then
                unprocessed = cnt - scansProcessed
                'Calculate the number of unprocessed scans
                'Calculate the number of integers are unprocessed
                ints = unprocessed * scanSize

                'Check to see if integer array has overflowed
                If ints = MAXBUF Then
                        disarmAcq
                        MsgBox "Internal buffer overrun"
                        Exit Sub
                End If
                'If ints + readIndex  MaxBufIndex, the buffer has wrapped around and we have to
                   process the data in two chunks — from the present readIndex to the end of the
                   buffer, and from the beginning of the buffer until all of the unprocessed scans are
                   processed. The readIndex keeps track of our read pointer in the data buffer.
                If ints + readIndex  MaxBufIndex Then
                        'The buffer has wrapped around, so 2 buffers must be transferred.
```

```
                'Calculate the size of the 1st chunk.
                firstBufSize = MaxBufIndex -readIndex + 1
                'Write the chunk to disk
                er = fWrite(fHandle, dataBuffer(readIndex), firstBufSize * 2)
                'Init the readIndex to the beginning.
                readIndex = 1
                'Calculate the size of the 2nd chunk.
                secondBufSize = ints -firstBufSize
                'Write the chunk to disk
                er = fWrite(fHandle, dataBuffer(readIndex), secondBufSize * 2)
                'Set the new readIndex
                readIndex = readIndex + secondBufSize
            Else
                'The buffer has not wrapped around, so only one buffer must be transferred.
                'Write the buffer to disk.
                er = fWrite(fHandle, dataBuffer(readIndex), ints * 2)
                'Set the new readIndex
                readIndex = readIndex + ints
            End If
        End If
            'Record the new number of scans processed.
        scansProcessed = cnt
        'If we've processed more scans that specifed by the user, then quit.
        If scansProcessed = Val(numScansText.text) Then
            disarmAcq
        End If
        'Post the number of processed scans.
        BufferedLabel.Caption = Str$(cnt)
End Sub

Sub disarmAcq ()
'Disarm the acquisition
        adc1.Arm = False                        'Stop sampling
        statusLabel.Caption = "idle"     'Update the status box
        Close #1                                'Close the output file
        getVal.Caption = "Acquire"      'Update the caption of the button
        collectDataTimer.Enabled = False         'Disable the data collection timer
End Sub

Sub Form_Load ()
        Dim i As Integer
        'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
        adc1.Buffer = addressOf(dataBuffer(0))
        'Set bufferLength to all adc1 to check buffer boundaries
        adc1.BufferLength = MAXBUF
        'Set the default frequency and number of scans
        numScansText.text = "10"
        freqText.text = "100"
        'Put channel choices in combos
        For i = 0 To 15
            chan(STARTCH).AddItem Format$(i)
            chan(ENDCH).AddItem Format$(i)
        Next i
        chan(ENDCH).ListIndex = 0
        chan(STARTCH).ListIndex = 0
        'Put gain choices in combo
        gain.AddItem "X1"
        gain.AddItem "X2"
        gain.AddItem "X4"
        gain.AddItem "X8"
        gain.ListIndex = 0
        'Put trigger source choices in combo
        trigSourceCombo.AddItem "Immediate"
        trigSourceCombo.AddItem "TTL"
        trigSourceCombo.AddItem "Analog"
        trigSourceCombo.ListIndex = 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub
```

```
Sub FreqText_Change ()
'Set the frequency property of ADC1
        If Val(freqText.text)  0 And Val(freqText.text)  100000 Then
                adc1.Frequency = Int(Val(freqText.text))
        End If
End Sub

Sub FreqText_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39               'All numbers
                Case &H8                        'Back space
                Case Else
                        keyascii = 0            'Reject all other characters
        End Select
End Sub

Sub gain_Click ()
        'Set the globalGain property of ADC1.  To set independent gains for each channel, use the
          ChanArray property rather than the startChan and endChan properties.
        adc1.GlobalGain = gain.ListIndex
End Sub

Sub GetVal_Click ()
                'Arm the system
        Dim MaxScansInBuf As Integer
        Dim bytesInFile As Long
        Dim i As Long
                'Disarm the acquisition if the button is labeled "Abort"
        If getVal.Caption = "Abort" Then
                disarmAcq
                Exit Sub
        End If
        'Update the status box
        statusLabel.Caption = "Waiting for trigger"
        statusLabel.Refresh
        'Initialize the acquisition variables
        scansProcessed = 0
        getVal.Caption = "Abort"
        readIndex = 1
        scanSize = adc1.EndChan - adc1.StartChan + 1
        MaxScansInBuf = Int(MAXBUF / scanSize)
        MaxBufIndex = MaxScansInBuf * scanSize
                'Open the data destination file, and get its handle
        Open "c:\ADCDATA.BIN" For Output As #1
        fHandle = FileAttr(1, 2)
                'Pre-write the file so that all of the required disk blocks
                'are allocated before the acquisition begins.  This allows the
                'file output to be performed faster.
        bytesInFile = (scanSize * 2) * Val(numScansText.text)
        For i = 1 To bytesInFile / 256 + 1
                er = fWrite(fHandle, dataBuffer(0), 256)
        Next i
        Seek 1, 1   'Set the file pointer back to the beginning
                'Start the acquisition
        adc1.Arm = True
                'If the trigger source is Software, send the software trigger.
        If trigSourceCombo.ListIndex = 0 Then adc1.SoftTrig = True
End Sub

Sub NumScansText_KeyPress (keyascii As Integer)
                'Filter non-numeric keystrokes.
        Select Case keyascii
                Case &H30 To &H39               'All numbers
                Case &H8                                        'Back space
                Case Else
                        keyascii = 0            'Reject all other characters
        End Select
End Sub

Sub RiseFall_Click (Index As Integer)
                'Set the trigger edge to rising or falling.
        If riseFall(0).Value = True Then
                adc1.TrigSourceRising = True
        Else
```

```
                adc1.TrigSourceRising = False
        End If
End Sub

Sub TrigLevelScroll_Change ()
        'Set the trigger level for analog triggering.  The DaqBook allows a trigger value of -5 to +5.  The
          scrollbar has a min and max of -50 to +50 to allow for 0.1 volt resolution when divided by 10.
        trigLevelLabel.Caption = Format$(trigLevelScroll.Value / 10) + "V"
        adc1.TrigLevel = trigLevelScroll.Value / 10
End Sub

Sub TrigLevelScroll_Scroll ()
        Call TrigLevelScroll_Change
End Sub

Sub TrigSourceCombo_Click ()
                'Set the trigSource property of ADC1.
        adc1.TrigSource = trigSourceCombo.ListIndex
        If trigSourceCombo.ListIndex = IMMEDIATE Then
                trigLevelScroll.Enabled = False
                riseFall(0).Enabled = False
                riseFall(1).Enabled = False
        ElseIf trigSourceCombo.ListIndex = ANALOG Then
                trigLevelScroll.Enabled = True
                riseFall(0).Enabled = True
                riseFall(1).Enabled = True
        Else
                trigLevelScroll.Enabled = False
                riseFall(0).Enabled = True
                riseFall(1).Enabled = True
        End If
End Sub
```

ADC5



ADC5 Form

```
Begin ADC Adc1
            BufferLength        =       1
            EndChan             =       0
            Frequency           =       10000
            GlobalBipolar       =       0           'False
            GlobalGain          =       0           ' 0 - Base  Unit X1
            GlobalSE            =       0           'False
            Left                =       30
            NumChannels         =       1
            NumScans            =       1
            OneShot             =       0           'False
            StartChan           =       0
            Top                 =       165
            TrigLevel           =       0
            TrigRefVoltage      =       -5
            TrigSource          =       0            'Software
            TrigSourceRising    =       -1          'True
            UseChanArray        =       0           'False
End
Begin DBK Dbk1
            IntLevel            =       7
            Left                =       30
            LptPort             =       0           'LPT1
            Protocol            =       1           '4 Bit I/O
            Top                 =       810
End

Const MAXBUF = 10                       'The size of my data buffer
            'For code readability, these are used to identify the index of the
            'two dimensional card information arrays below.
Const SETTING = 0
Const BITWEIGHT = 1
Const BIPOLAROFFSET = 2
  'For code readability, these are used to identify the selected expansion card.
Const A_BASEUNIT = 0
Const A_DBK12 = 1
Const A_DBK13 = 2
Const A_DBK14 = 3
            'This is the data buffer for all of the analog input data
Dim dataBuffer(MAXBUF) As Integer
'These arrays hold expansion card information.  See the subroutine, loadGainArrays for more information.
Dim BASEUNIT(3, 10) As Single
Dim DBK12(3, 10) As Single
Dim DBK13(3, 10) As Single
Dim DBK14(3, 10) As Single

Sub boardType_click ()
            'Depending of the board type selected,  update the gain combo with the available gains.
            Select Case boardType.ListIndex
                        Case 0                          'None
                                    gain.Clear
                                    gain.AddItem "X1"
                                    gain.AddItem "X2"
                                    gain.AddItem "X4"
                                    gain.AddItem "X8"
                                    chan.Enabled = False
                        Case 1                          'DKB12
```

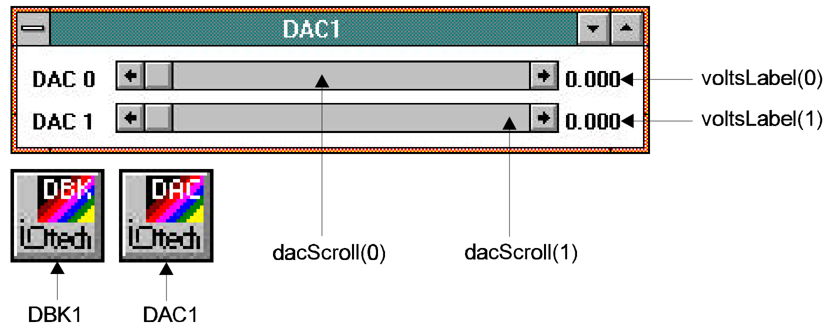
```
                    gain.Clear
                    gain.AddItem "X1"
                    gain.AddItem "X2"
                    gain.AddItem "X4"
                    gain.AddItem "X8"
                    gain.AddItem "X16"
                    gain.AddItem "X32"
                    gain.AddItem "X64"
                    chan.Enabled = True
                Case 2                      'DBK13
                    gain.Clear
                    gain.AddItem "X1"
                    gain.AddItem "X2"
                    gain.AddItem "X4"
                    gain.AddItem "X8"
                    gain.AddItem "X10"
                    gain.AddItem "X100"
                    gain.AddItem "X1000"
                    chan.Enabled = True
                Case 3                      'DBK14
                    gain.Clear
                    gain.AddItem "X1"
                    gain.AddItem "X2"
                    gain.AddItem "X4"
                    gain.AddItem "X8"
                    gain.AddItem "X10"
                    gain.AddItem "X100"
                    gain.AddItem "X1000"
                    chan.Enabled = True
        End Select
        gain.ListIndex = 0
        Call chan_click
End Sub

Sub chan_click ()
                'Set the startChan and endChan to the desired channel
                'If the listindex = 0, no expansion boards are attached
        If boardType.ListIndex = 0 Then
                'Set the channel equal to the base unit channel number
                adc1.StartChan =  expBoard.ListIndex
                adc1.EndChan = adc1.StartChan
        Else
                'Calculate then set the channel number for the expansion board.
                adc1.StartChan = (expBoard.ListIndex + 1) * 16 + chan.ListIndex
                adc1.EndChan = adc1.StartChan
        End If
End Sub

Sub expBoard_Click ()
'When the DaqBook base unit channel has changed, set the board type back to NONE.
        If boardType.ListIndex = 0 Then
                Call chan_click
        Else
                boardType.ListIndex = 0
        End If
End Sub

Sub Form_Load ()
        Dim i As Integer
                'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
        adc1.Buffer = addressOf(dataBuffer(0)) 'dataBuffer has global scope
        adc1.BufferLength = MAXBUF
        Call loadGainArrays
                'Put channel choices in combos
        For i = 0 To 15
                chan.AddItem Format$(i)
                expBoard.AddItem Format$(i)
        Next i
                'Put DBK boards in combo
        boardType.AddItem "None"
        boardType.AddItem "DBK12"
        boardType.AddItem "DBK13"
        boardType.AddItem "DBK14"
```

```
        stopper.Value = True            'Click the Stop button
        chan.ListIndex = 0   'Set the expansion channel to 0
        expBoard.ListIndex = 0          'Set the base unit channel to 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub

Sub gain_Click ()
        'Use the gain tables to set the gain for the desired expansion board
        Select Case boardType.ListIndex
                Case A_BASEUNIT
                        adc1.GlobalGain = BASEUNIT(SETTING, gain.ListIndex)
                Case A_DBK12
                        adc1.GlobalGain = DBK12(SETTING, gain.ListIndex)
                Case A_DBK13
                        adc1.GlobalGain = DBK13(SETTING, gain.ListIndex)
                Case A_DBK14
                        adc1.GlobalGain = DBK14(SETTING, gain.ListIndex)
        End Select
End Sub

Sub GetVal_Click ()
'Enable sampling.  Disable all of the controls so parameters can not be adjusted during sampling.
        getVal.Enabled = False
        boardType.Enabled = False
        expBoard.Enabled = False
        gain.Enabled = False
        chan.Enabled = False
        timer1.Enabled = True
        stopper.Enabled = True
End Sub

Sub loadGainArrays ()
'Load the 2-dimensional arrays with gain information for each expansion card.  Each card has an array of
  available gains.  The gain combo listindex identifies the gain chosen.  Associated with each listindex is a
  SETTING which is assigned to the globalGain property of the ADC1 control, and a BITWEIGHT and a
  BIPOLAROFFSET, which is used later to convert the raw binary values to volts.
        'The BITWEIGHT and BIPOLAROFFSET are calculated as follows:
        '   BITWEIGHT = 65536 / inputVoltageRange
        '   BIPOLAROFFSET = bipolarRange
        '   For example: X2 gain, bipolar mode, yields a range of +/-2.5 volts.
        '   BITWEIGHT = 65536 / 5volts = 13107.2
        '   BIPOLAROFFSET = 2.5
        '  X1, X2, X4, X8 available
        BASEUNIT(SETTING, 0) = &H0
        BASEUNIT(SETTING, 1) = &H1
        BASEUNIT(SETTING, 2) = &H2
        BASEUNIT(SETTING, 3) = &H3
        BASEUNIT(BITWEIGHT, 0) = 6553.6
        BASEUNIT(BITWEIGHT, 1) = 13107.2
        BASEUNIT(BITWEIGHT, 2) = 26214.4
        BASEUNIT(BITWEIGHT, 3) = 52428.8
        BASEUNIT(BIPOLAROFFSET, 0) = 5
        BASEUNIT(BIPOLAROFFSET, 1) = 2.5
        BASEUNIT(BIPOLAROFFSET, 2) = 1.25
        BASEUNIT(BIPOLAROFFSET, 3) = .625
                'X1, X2, X4, X8, X16, X32, X64 available
        DBK12(SETTING, 0) = &H0
        DBK12(SETTING, 1) = &H1
        DBK12(SETTING, 2) = &H2
        DBK12(SETTING, 3) = &H3
        DBK12(SETTING, 4) = &H13
        DBK12(SETTING, 5) = &H23
        DBK12(SETTING, 6) = &H33
        DBK12(BITWEIGHT, 0) = 6553.6
        DBK12(BITWEIGHT, 1) = 13107.2
        DBK12(BITWEIGHT, 2) = 26214.4
        DBK12(BITWEIGHT, 3) = 52428.8
        DBK12(BITWEIGHT, 4) = 104857.6
        DBK12(BITWEIGHT, 5) = 209715.2
        DBK12(BITWEIGHT, 6) = 419430.4
```

```
        DBK12(BIPOLAROFFSET, 0) = 5
        DBK12(BIPOLAROFFSET, 1) = 2.5
        DBK12(BIPOLAROFFSET, 2) = 1.25
        DBK12(BIPOLAROFFSET, 3) = .625
        DBK12(BIPOLAROFFSET, 4) = .3125
        DBK12(BIPOLAROFFSET, 5) = .15625
        DBK12(BIPOLAROFFSET, 6) = .078125
                'X1, X2, X4, X8, X10, X100, X1000 available
        DBK13(SETTING, 0) = &H00
        DBK13(SETTING, 1) = &H10
        DBK13(SETTING, 2) = &H20
        DBK13(SETTING, 3) = &H30
        DBK13(SETTING, 4) = &H01
        DBK13(SETTING, 5) = &H02
        DBK13(SETTING, 6) = &H03
        DBK13(BITWEIGHT, 0) = 6553.6
        DBK13(BITWEIGHT, 1) = 13107.2
        DBK13(BITWEIGHT, 2) = 26214.4
        DBK13(BITWEIGHT, 3) = 52428.8
        DBK13(BITWEIGHT, 4) = 65536
        DBK13(BITWEIGHT, 5) = 655360
        DBK13(BITWEIGHT, 6) = 6553600
        DBK13(BIPOLAROFFSET, 0) = 5
        DBK13(BIPOLAROFFSET, 1) = 2.5
        DBK13(BIPOLAROFFSET, 2) = 1.25
        DBK13(BIPOLAROFFSET, 3) = .625
        DBK13(BIPOLAROFFSET, 4) = .5
        DBK13(BIPOLAROFFSET, 5) = .05
        DBK13(BIPOLAROFFSET, 6) = .005
                'X1, X2, X4, X8, X10, X100, X1000 available
        DBK14(SETTING, 0) = &H0
        DBK14(SETTING, 1) = &H1
        DBK14(SETTING, 2) = &H2
        DBK14(SETTING, 3) = &H3
        DBK14(SETTING, 4) = &H1
        DBK14(SETTING, 5) = &H2
        DBK14(SETTING, 6) = &H3
        DBK14(BITWEIGHT, 0) = 6553.6
        DBK14(BITWEIGHT, 1) = 13107.2
        DBK14(BITWEIGHT, 2) = 26214.4
        DBK14(BITWEIGHT, 3) = 52428.8
        DBK14(BITWEIGHT, 4) = 65536
        DBK14(BITWEIGHT, 5) = 655360
        DBK14(BITWEIGHT, 6) = 6553600
        DBK14(BIPOLAROFFSET, 0) = 5
        DBK14(BIPOLAROFFSET, 1) = 2.5
        DBK14(BIPOLAROFFSET, 2) = 1.25
        DBK14(BIPOLAROFFSET, 3) = .625
        DBK14(BIPOLAROFFSET, 4) = .5
        DBK14(BIPOLAROFFSET, 5) = .05
        DBK14(BIPOLAROFFSET, 6) = .005
End Sub

Sub Stopper_Click ()
                'Enable the acquisition parameter controls, and disable the stop button.
        getVal.Enabled = True
        boardType.Enabled = True
        expBoard.Enabled = True
        gain.Enabled = True
        chan.Enabled = True
        timer1.Enabled = False
        stopper.Enabled = False
End Sub

Sub Timer1_Timer ()
                'Get a reading and post it in the GUI in raw counts and volts
        Dim bitsPerVolt As Single
        Dim offset As Single
        Dim goodInt As Long
                'Start the acquisition and wait for one scan to be collected
        adc1.Arm = True
        adc1.SoftTrig = True
        While adc1.Active: Wend
```

```
'Convert the 16 bit value in the integer array to a long.  VB will interpret the value as a signed number with
   the MSB as the sign bit.  In VB's integer format, a 0 from the A/D converter looks like -32767, and
   65535 from the A/D looks like 32766.  The following lines convert the signed integer value into a
   continuous value ranging from 0 - 65535.
        goodInt = dataBuffer(0)
        If goodInt  0 Then goodInt = goodInt + 65536
                'Put the raw count into the textbox dataText.
        dataText.Text = Format$(goodInt)
                'Scale the raw data into volts
        Select Case boardType.ListIndex
                Case A_BASEUNIT
                        bitsPerVolt = BASEUNIT(BITWEIGHT, gain.ListIndex)
                        offset = BASEUNIT(BIPOLAROFFSET, gain.ListIndex)
                Case A_DBK12
                        bitsPerVolt = DBK12(BITWEIGHT, gain.ListIndex)
                        offset = DBK12(BIPOLAROFFSET, gain.ListIndex)
                Case A_DBK13
                        bitsPerVolt = DBK13(BITWEIGHT, gain.ListIndex)
                        offset = DBK13(BIPOLAROFFSET, gain.ListIndex)
                Case A_DBK14
                        bitsPerVolt = DBK14(BITWEIGHT, gain.ListIndex)
                        offset = DBK14(BIPOLAROFFSET, gain.ListIndex)
        End Select
                'Put the volts value into the textbox voltsText.
        voltsText.Text = Format$(goodInt / bitsPerVolt - offset, "0.000000")
End Sub
```

DAC1

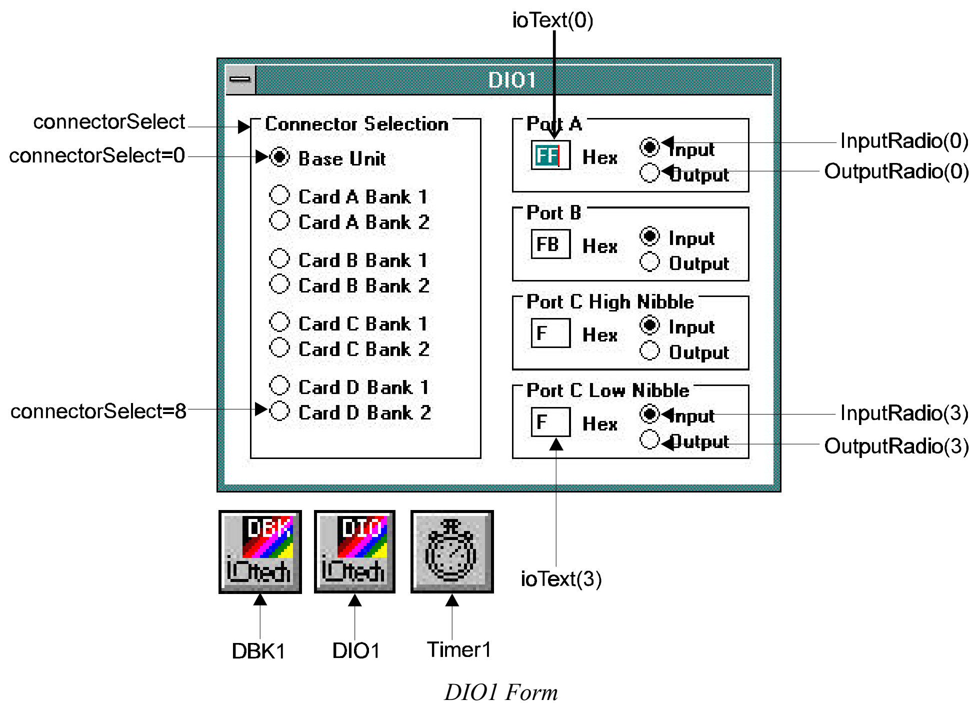


DAC1 Form

```
Begin DBK Dbk1
        IntLevel        =       7
        Left            =       2700
        LptPort         =       0       'LPT1
        Protocol        =       1       '4 Bit I/O
        Top             =       180
End
Begin DAC Dac1
        Left            =       2100
        Top             =       180
End
```

```
Sub DacScroll_Change (Index As Integer)
                'Scroll bar max is set to 4096, min to 0 in prop window
        dac1.ChVoltage(Index) = dacScroll(Index).Value
        voltsLabel(Index).Caption = Format$(dacScroll(Index).Value * .0012207, "0.000")
End Sub

Sub DacScroll_Scroll (Index As Integer)
        Call DacScroll_Change(Index)
End Sub

Sub Form_Load ()
        'Open DaqBook driver
        dbk1.Open = True
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub
```

DIO1



DIO1 Form

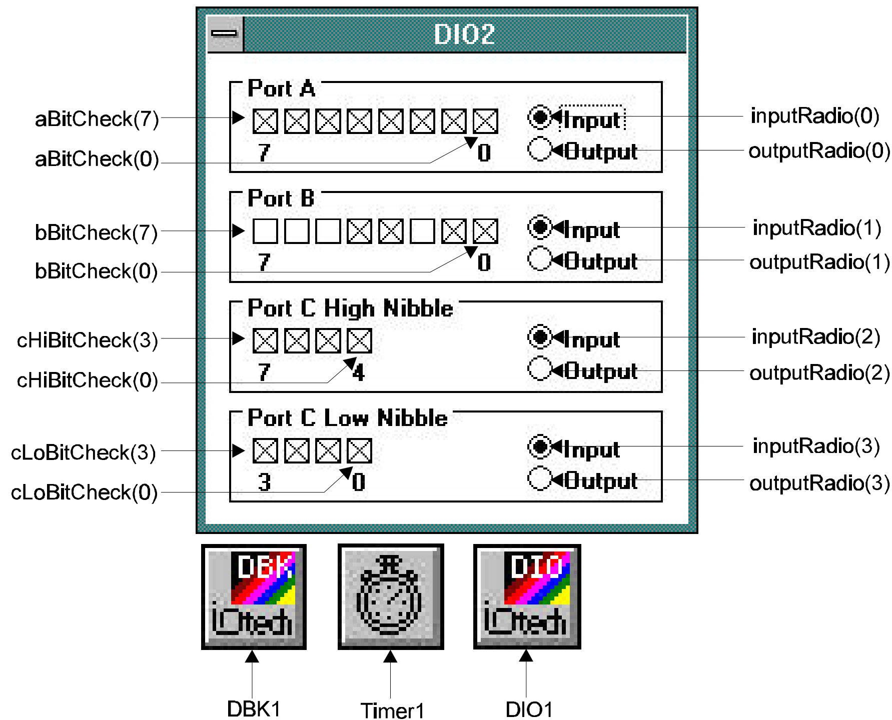
```
Begin DIO Dio1
        ByteIn             =     0
        ByteOut            =     0
        IndexIn            =     0
        IndexOut           =     0
        Left               =     2160
        Local              =     -1        'True
        LocalAByte         =     0
        LocalASetAsInput   =     -1        'True
        LocalBByte         =     0
        LocalBSetAsInput   =     -1        'True
        LocalCHiNibble     =     0
        LocalCHiSetAsInput=      -1        'True
        LocalCLoNibble     =     0
        LocalCLoSetAsInput=      -1        'True
        Top                =     660
End
```

```
Begin DBK Dbk1
        IntLevel        =       7
        Left            =       2160
        LptPort         =       0       'LPT1
        Protocol        =       1       '4 Bit I/O
        Top             =       1140
End

Dim connectorSelect As Integer

Const INPUTMODE = True
Const OUTPUTMODE = False
Const BASEUNIT = 0
Const A1 = 1
Const A2 = 2
Const B1 = 3
Const B2 = 4
Const C1 = 5
Const C2 = 6
Const D1 = 7
Const D2 = 8
Const PORTA = 0
Const PORTB = 1
Const PORTCHI = 2
Const PORTCLO = 3

Sub Connector_Click (index As Integer)
        Dim i As Integer
        If index = 0 Then
                dio1.Local = True
        Else
                dio1.Local = False
        End If
        connectorSelect = index
        For i = 0 To 3
                inputRadio(i).Value = True
        Next i
End Sub

Sub Form_Load ()
        Dim i As Integer
                'Open DaqBook driver
        dbk1.Open = True

                'Select the base unit connector
        connector(0).Value = True

                'Set all of the ports as inputs
        For i = 0 To 3
                inputRadio(i).Value = True
        Next i
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
        End
End Sub

Function getByteFromPort (whichPort As Integer) As Integer
        Dim theVal As Integer
        If connectorSelect = BASEUNIT Then
                Select Case whichPort
                        Case PORTA
                                theVal = dio1.LocalAByte
                        Case PORTB
                                theVal = dio1.LocalBByte
                        Case PORTCHI
                                theVal = dio1.LocalCHiNibble
                        Case PORTCLO
                                theVal = dio1.LocalCLoNibble
                End Select
        Else
                Select Case whichPort
                        Case PORTA
```

```
                                theVal = dio1.ExpAByte(connectorSelect - 1)
                        Case PORTB
                                theVal = dio1.ExpBByte(connectorSelect - 1)
                        Case PORTCHI
                                theVal = dio1.ExpCHiNibble(connectorSelect - 1)
                        Case PORTCLO
                                theVal = dio1.ExpCLoNibble(connectorSelect - 1)
                End Select
        End If
        getByteFromPort = theVal
End Function

Sub GetInputDataTimer_Timer ()
        Dim i As Integer
        For i = 0 To 3
                If inputRadio(i).Value = True Then
                        ioText(i).text = Hex$(getByteFromPort(i))
                End If
        Next i
End Sub

Function hexVal (hexString As String) As Integer
        Dim hiChar As Integer
        Dim loChar As Integer
        If Len(hexString) = 0 Then
                hexVal = 0
                Exit Function
        End If
        If Len(hexString) = 2 Then
                hiChar = Asc(hexString) - &H30
                If hiChar  10 Then hiChar = hiChar -7
        End If
        loChar = Asc(Mid$(hexString, Len(hexString), 1)) - &H30
        If loChar  10 Then loChar = loChar -7
        hexVal = hiChar * 16 + loChar
End Function

Sub InputRadio_click (index As Integer)
        Call setupPortIo(index, INPUTMODE)
End Sub

Sub ioText_Change (index As Integer)
        Dim maxLen As Integer
        Dim byteString As String
        If index = 0 Or index = 1 Then
                maxLen = 2
        Else
                maxLen = 1
        End If
        If Len(ioText(index).text) = maxLen Then
                ioText(index).SelStart = 0
                ioText(index).SelLength = Len(ioText(index).text)
        End If
        If outputRadio(index).Value = True Then
                byteString = ioText(index).text
                Call putByteToPort(index, byteString)
        End If
End Sub

Sub ioText_GotFocus (index As Integer)
        ioText(index).SelStart = 0
        ioText(index).SelLength = Len(ioText(index).text)
End Sub

Sub ioText_KeyPress (index As Integer, keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39       'All numbers
                Case &H8                'Back space
                Case &H61 To &H66       'Chars a to f
                        keyascii = keyascii - &H20      'make upper
                Case &H41 To &H46       'Chars A to F
                Case Else
                        keyascii = 0
        End Select
```

```
End Sub

Sub OutputRadio_Click (index As Integer)
        ioText(index).text = "00"
        Call setupPortIo(index, OUTPUTMODE)
End Sub

Sub putByteToPort (whichPort As Integer, textVal As String)
        Dim outputVal As Integer
        outputVal = hexVal(textVal)
        If connectorSelect = BASEUNIT Then
                Select Case whichPort
                        Case PORTA
                                dio1.LocalAByte = outputVal
                        Case PORTB
                                dio1.LocalBByte = outputVal
                        Case PORTCHI
                                dio1.LocalCHiNibble = outputVal
                        Case PORTCLO
                                dio1.LocalCLoNibble = outputVal
                End Select
        Else
                Select Case whichPort
                        Case PORTA
                                dio1.ExpAByte(connectorSelect - 1) = outputVal
                        Case PORTB
                                dio1.ExpBByte(connectorSelect - 1) = outputVal
                        Case PORTCHI
                                dio1.ExpCHiNibble(connectorSelect - 1) = outputVal
                        Case PORTCLO
                                dio1.ExpCLoNibble(connectorSelect - 1) = outputVal
                End Select
        End If
End Sub

Sub setupPortIo (whichPort As Integer, ioMode As Integer)
        If connectorSelect = BASEUNIT Then
                Select Case whichPort
                        Case PORTA
                                dio1.LocalASetAsInput = ioMode
                        Case PORTB
                                dio1.LocalBSetAsInput = ioMode
                        Case PORTCHI
                                dio1.LocalCHiSetAsInput = ioMode
                        Case PORTCLO
                                dio1.LocalCLoSetAsInput = ioMode
                End Select
        Else
                Select Case whichPort
                        Case PORTA
                                dio1.ExpASetAsInput(connectorSelect - 1) = ioMode
                        Case PORTB
                                dio1.ExpBSetAsInput(connectorSelect - 1) = ioMode
                        Case PORTCHI
                                dio1.ExpCHiSetAsInput(connectorSelect - 1) = ioMode
                        Case PORTCLO
                                dio1.ExpCLoSetAsInput(connectorSelect - 1) = ioMode
                End Select
        End If
End Sub
```

DIO2



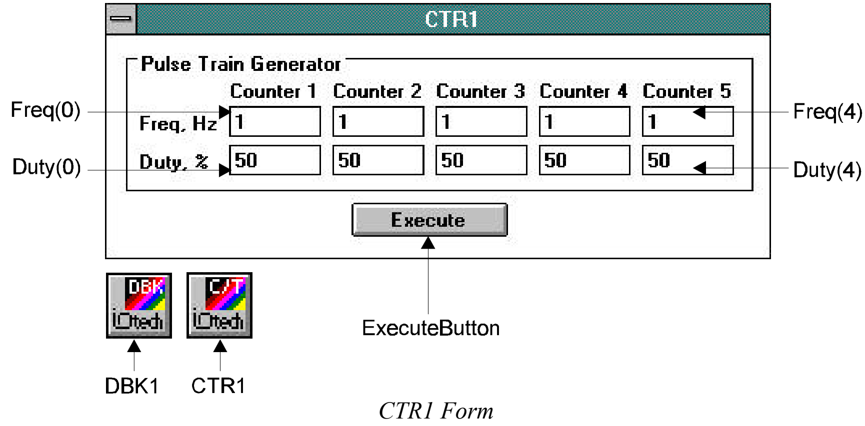
DIO2 Form

```
Begin DIO Dio1
        ByteIn              =       0
        ByteOut             =       0
        IndexIn             =       0
        IndexOut            =       0
        Left                =       60
        Local               =       -1      'True
        LocalAByte          =       0
        LocalASetAsInput    =       -1      'True
        LocalBByte          =       0
        LocalBSetAsInput    =       -1      'True
        LocalCHiNibble      =       0
        LocalCHiSetAsInput=             -1      'True
        LocalCLoNibble      =       0
        LocalCLoSetAsInput=             -1      'True
        Top                 =       3120
End
Begin DBK Dbk1
        IntLevel            =       7
        Left                =       1020
        LptPort             =       0       'LPT1
        Protocol            =       1       '4 Bit I/O
        Top                 =       3120
End

Const INPUTMODE = True
Const OUTPUTMODE = False
Const BASEUNIT = 0
Const A1 = 1
Const A2 = 2
Const B1 = 3
Const B2 = 4
Const C1 = 5
Const C2 = 6
Const D1 = 7
Const D2 = 8
Const PORTA = 0
Const PORTB = 1
Const PORTCHI = 2
```

```
Const PORTCLO = 3

Sub aBitCheck_Click (Index As Integer)
        'If the selected port is an output, put the new bit value on the port.
        Dim theVal As Integer
        If outputRadio(PORTA).Value = False Then Exit Sub
        theVal = aBitCheck(Index).Value
        Call checkChange(PORTA, Index, theVal)
End Sub

Sub bBitCheck_Click (Index As Integer)
        'If the selected port is an output, put the new bit value on the port.
        Dim theVal As Integer
        If outputRadio(PORTB).Value = False Then Exit Sub
        theVal = bBitCheck(Index).Value
        Call checkChange(PORTB, Index, theVal)
End Sub

Sub checkChange (port As Integer, bit As Integer, theValue As Integer)
        'Change the value of the selected output bit on the selected port to the selected value. Only
          change the bit if the port is configured for output
        If outputRadio(port).Value = True Then  'Only change the bit if the
                Select Case port
                        Case 0
                                dio1.LocalABit(bit) = theValue
                        Case 1
                                dio1.LocalBBit(bit) = theValue
                        Case 2
                                dio1.LocalCHiBit(bit) = theValue
                        Case 3
                                dio1.LocalCLoBit(bit) = theValue
                End Select
        End If
End Sub

Sub cHiBitCheck_Click (Index As Integer)
        'If the selected port is an output, put the new bit value on the port.
        Dim theVal As Integer
        If outputRadio(PORTCHI).Value = False Then Exit Sub
        theVal = cHiBitCheck(Index).Value
        Call checkChange(PORTCHI, Index, theVal)
End Sub

Sub cLoBitCheck_Click (Index As Integer)
        'If the selected port is an output, put the new bit value on the port.
        Dim theVal As Integer
        If outputRadio(PORTCLO).Value = False Then Exit Sub
        theVal = cLoBitCheck(Index).Value
        Call checkChange(PORTCLO, Index, theVal)
End Sub

Sub Form_Load ()
        Dim i As Integer
                'Open DaqBook driver and allocate a data buffer
        dbk1.Open = True
                'Set all of the ports as inputs
        For i = 0 To 3
                inputRadio(i).Value = True
        Next i
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub

Sub GetInputDataTimer_Timer ()
        'Every time the timer ticks, test all of the bits on ports configured
                'as inputs and post the results in the associated check boxes.
        Dim i As Integer
        For i = 0 To 3  'Loop through all of the ports
                If inputRadio(i).Value = True Then
                'If configured as an input, post the bit values for this port
                        putBitsInCheckBoxes (i)
                End If
```

```
        Next i
End Sub
Sub InputRadio_click (Index As Integer)
                'Configure the specified port as an input.
        Call setupPortIo(Index, INPUTMODE)
        Call resetCheckBoxes
End Sub
Sub OutputRadio_Click (Index As Integer)
'Configure the specified port as an output.

        Call setupPortIo(Index, OUTPUTMODE)
        Call resetCheckBoxes
End Sub
Sub putBitsInCheckBoxes (whichPort As  Integer)
        'For the selected port, scan the input bits and place the result of each
                'bit test in the associated check box
        Dim i As Integer
        Dim maxBits As Integer
                'Ports A and B have bits 0-7, C hi and C lo have 0-3.
        If whichPort  2 Then
                maxBits = 7
        Else
                maxBits = 3
        End If
                'The DIO property LocalxBits(i) returns a 0 or a -1, the check boxes accept either a 0
                  or a 1 for their value.  That's why the bit test is multiplied by -1.
        Select Case whichPort
                Case PORTA
                        For i = 0 To maxBits
                                aBitCheck(i).Value = dio1.LocalABit(i) * -1
                        Next i
                Case PORTB
                        For i = 0 To maxBits
                                bBitCheck(i).Value = dio1.LocalBBit(i) * -1
                        Next i
                Case PORTCHI
                        For i = 0 To maxBits
                                cHiBitCheck(i).Value = dio1.LocalCHiBit(i) * -1
                        Next i
                Case PORTCLO
                        For i = 0 To maxBits
                                cLoBitCheck(i).Value = dio1.LocalCLoBit(i) * -1
                        Next i
        End Select
End Sub
Sub resetCheckBoxes ()
                'Set all of the check boxes back to 0.
        Dim i As Integer
        For i = 0 To 7
                aBitCheck(i).Value = False
                bBitCheck(i).Value = False
                If i  4 Then
                        cHiBitCheck(i).Value =  False
                        cLoBitCheck(i).Value = False
                End If
        Next i
End Sub

Sub setupPortIo (whichPort As Integer, ioMode As Integer)
        'Configure the selected port as either an input or an output.
        Select Case whichPort
                Case PORTA
                        dio1.LocalASetAsInput = ioMode
                Case PORTB
                        dio1.LocalBSetAsInput = ioMode
                Case PORTCHI
                        dio1.LocalCHiSetAsInput = ioMode
                Case PORTCLO
                        dio1.LocalCLoSetAsInput = ioMode
        End Select
End Sub
```

CTR1



CTR1 Form

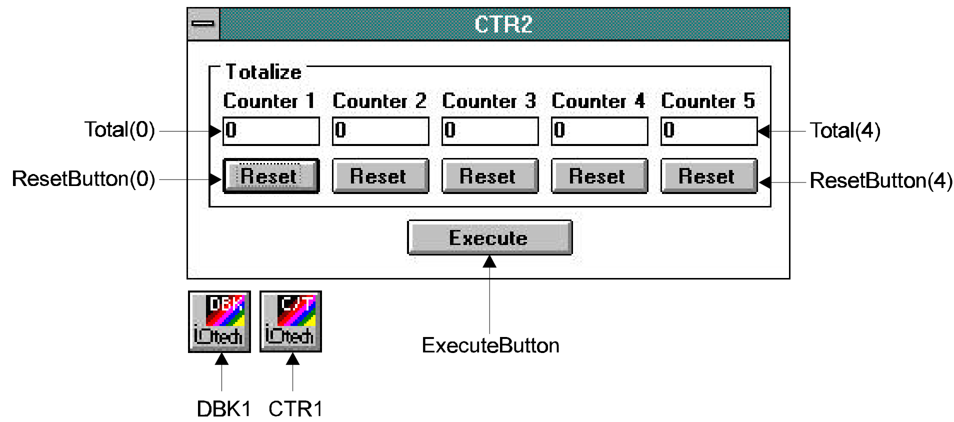
```
Begin CTR Ctrl
        BufferLength      =    1
        C1CntDir          =    0        'Count Down
        C1CntEdge         =    0        'Negative Count Edge
        C1CntRepeat       =    0         'False
        C1CntSource       =    0        ' 0 - TC toggled output of last ctr
        C1CntType         =    0        'Binary Count
        C1Enable          =    0        'False
        C1GateCtrl        =    0        'Gating Disabled
        C1OutputCtrl      =    0        'Inactive -Always low
        C1Reload          =    0        'Reload from Load
        C1SpecialGate     =    0        'False
        C2CntDir          =    0        'Count Down
        C2CntEdge         =    0        'Negative Count Edge
        C2CntRepeat       =    0         'False
        C2CntSource       =    0        ' 0 - TC toggled output of last ctr
        C2CntType         =    0        'Binary Count
        C2Enable          =    0        'False
        C2GateCtrl        =    0        'Gating Disabled
        C2OutputCtrl      =    0        'Inactive -Always low
        C2Reload          =    0        'Reload from Load
        C2SpecialGate     =    0        'False
        C3CntDir          =    0        'Count Down
        C3CntEdge         =    0        'Negative Count Edge
        C3CntRepeat       =    0         'False
        C3CntSource       =    0        ' 0 - TC toggled output of last ctr
        C3CntType         =    0        'Binary Count
        C3Enable          =    0        'False
        C3GateCtrl        =    0        'Gating Disabled
        C3OutputCtrl      =    0        'Inactive -Always low
        C3Reload          =    0        'Reload from Load
        C3SpecialGate     =    0        'False
        C4CntDir          =    0        'Count Down
        C4CntEdge         =    0        'Negative Count Edge
        C4CntRepeat       =    0         'False
        C4CntSource       =    0        ' 0 - TC toggled output of last ctr
        C4CntType         =    0        'Binary Count
        C4Enable          =    0        'False
        C4GateCtrl        =    0        'Gating Disabled
        C4OutputCtrl      =    0        'Inactive -Always low
        C4Reload          =    0        'Reload from Load
        C4SpecialGate     =    0        'False
        C5CntDir          =    0        'Count Down
        C5CntEdge         =    0        'Negative Count Edge
        C5CntRepeat       =    0         'False
        C5CntSource       =    0        ' 0 - TC toggled output of last ctr
        C5CntType         =    0        'Binary Count
        C5Enable          =    0        'False
        C5GateCtrl        =    0        'Gating  Disabled
        C5OutputCtrl      =    0        'Inactive -Always low
        C5Reload          =    0        'Reload from Load
```

```
            C5SpecialGate   =   0   'False
            Comp1Enable     =   0    'False
            Comp2Enable     =   0    'False
            FoutDivider     =   0   ' Divide by 16
            FoutSource      =   0   ' 0 - Fout Disabled
            FreqCntSource   =   1   'Counter 1 Input
            FreqInterval    =   1
            Left            =   600
            NumScans        =   1
            TimeOfDay       =   0   'Disabled
            Top             =   1500
        End
        Begin DBK Dbk1
            IntLevel        =   7
            Left            =   180
            LptPort         =   0   'LPT1
            Protocol        =   1   '4 Bit I/O
            Top             =   1500
        End
Function CvtMinMax (ByVal CvtStr As String, ByVal min As Long, ByVal Max As Long)   As Long
                ' converts a string to a long integer number
                ' and makes sure that the number is within bounds
            If CvtStr = "" Then
                CvtMinMax = min
            Else
                Dim CvtVal As Long
                CvtVal = CLng(CvtStr)
                If CvtVal  Max Then
                    CvtVal = Max
                ElseIf CvtVal  min Then
                    CvtVal = min
                End If
                CvtMinMax = CvtVal
            End If
        End Function

        Sub Duty_KeyPress (Index As Integer, keyascii As Integer)
            Select Case keyascii
                Case &H30 To &H39   'All numbers
                Case &H8                        'Back space
                Case Else
                    keyascii = 0        'Reject all other characters
            End Select
        End Sub

        Sub ExecuteButton_Click ()
            Dim userduty%, userfreq&, srcfreq&, src%
            Dim i As Integer
                    ' Disarm all 5 counters
            ctr1.Disarm = True
            For i = 1 To 5
                    ' Read user input
                userduty% = CvtMinMax(Duty(i -1).Text, 1, 99)
                userfreq& = CvtMinMax(Freq(i -1).Text, 1, 1000000)
                    ' Decide which internal source to use as an input
            If userfreq  20 Then
                ' for faster waveforms, use the 1MHz clock
                src% = 11
                srcfreq& = 1000000
            Else
                ' for slower waveforms, use the 10kHz clock
                src% = 13
                srcfreq& = 10000
            End If
                ' Set hold and load registers and input source
            Select Case i
            Case 1
                ctr1.C1CntSource = src%
                ctr1.C1Hold = Max(CInt((srcfreq& * userduty%) / (100 * userfreq&)), 1)
                ctr1.C1Load = Max(CInt((srcfreq& * (100 - userduty%)) / (100 * userfreq&)), 1)
            Case 2
                ctr1.C2CntSource = src%
                ctr1.C2Hold = Max(CInt((srcfreq& * userduty%) / (100 * userfreq&)), 1)
                ctr1.C2Load = Max(CInt((srcfreq& * (100 - userduty%)) / (100 * userfreq&)), 1)
```

```
        Case 3
                ctr1.C3CntSource = src%
                ctr1.C3Hold =  Max(CInt((srcfreq& * userduty%) / (100 * userfreq&)), 1)
                ctr1.C3Load = Max(CInt((srcfreq& * (100 - userduty%)) / (100 * userfreq&)), 1)
        Case 4
                ctr1.C4CntSource = src%
                ctr1.C4Hold = Max(CInt((srcfreq& * userduty%) / (100 * userfreq&)), 1)
                ctr1.C4Load = Max(CInt((srcfreq& * (100 - userduty%)) / (100 * userfreq&)), 1)
        Case 5
                ctr1.C5CntSource = src%
                ctr1.C5Hold = Max(CInt((srcfreq& * userduty%) / (100 * userfreq&)), 1)
                ctr1.C5Load = Max(CInt((srcfreq& * (100 - userduty%)) / (100 * userfreq&)), 1)
        End Select
                ' Initialize all 5 counters
                ctr1.SetCounterMode = i
        Next i
                ' Start all 5 counters
        ctr1.LoadArm = True
End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.Open = True
                ' Set non-changing properties of each counter. These properties are set here and
                  never change. Alternatively, they could be set in the properties window
        ctr1.C1Enable = True            ' counter enabled for arm/disarm, save/load commands
        ctr1.C1CntDir = 0               ' count down
        ctr1.C1CntEdge = 1              ' count on positive edge
        ctr1.C1CntRepeat = True         ' repeat enabled
        ctr1.C1CntType = 0              ' binary counting
        ctr1.C1GateCtrl = 0             ' no gating
        ctr1.C1OutputCtrl = 2           ' TC toggled output
        ctr1.C1Reload = 1               ' reload from load or hold
        ctr1.C1SpecialGate = 0          ' special gate  disabled
        ctr1.C2Enable = True            ' counter enabled for arm/disarm, save/load commands
        ctr1.C2CntDir = 0               ' count down
        ctr1.C2CntEdge = 1              ' count on positive edge
        ctr1.C2CntRepeat = True         ' repeat enabled
        ctr1.C2CntType = 0              ' binary counting
        ctr1.C2GateCtrl = 0             ' no gating
        ctr1.C2OutputCtrl = 2           ' TC toggled output
        ctr1.C2Reload = 1               ' reload from load or hold
        ctr1.C2SpecialGate = 0          ' special gate disabled
        ctr1.C3Enable = True            ' counter enabled for arm/disarm, save/load commands
        ctr1.C3CntDir = 0               ' count down
        ctr1.C3CntEdge = 1              ' count on positive edge
        ctr1.C3CntRepeat = True         ' repeat enabled
        ctr1.C3CntType = 0              ' binary counting
        ctr1.C3GateCtrl = 0             ' no gating
        ctr1.C3OutputCtrl = 2           ' TC toggled output
        ctr1.C3Reload = 1               ' reload from load or hold
        ctr1.C3SpecialGate = 0          ' special gate disabled
        ctr1.C4Enable = True            ' counter enabled for arm/disarm, save/load commands
        ctr1.C4CntDir = 0               ' count down
        ctr1.C4CntEdge = 1              ' count on positive edge
        ctr1.C4CntRepeat = True         ' repeat enabled
        ctr1.C4CntType = 0              ' binary counting
        ctr1.C4GateCtrl = 0             ' no gating
        ctr1.C4OutputCtrl = 2           ' TC toggled output
        ctr1.C4Reload = 1               ' reload from load or hold
        ctr1.C4SpecialGate = 0          ' special gate disabled
        ctr1.C5Enable = True            ' counter enabled for arm/disarm, save/load commands
        ctr1.C5CntDir = 0               ' count down
        ctr1.C5CntEdge = 1              ' count on positive edge
        ctr1.C5CntRepeat = True         ' repeat enabled
        ctr1.C5CntType = 0              ' binary counting
        ctr1.C5GateCtrl = 0             ' no gating
        ctr1.C5OutputCtrl = 2           ' TC toggled output
        ctr1.C5Reload = 1               ' reload from load or hold
        ctr1.C5SpecialGate = 0          ' special gate disabled
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
```

```
End Sub

Sub Freq_KeyPress (Index As Integer, keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39          'All numbers
                Case &H8                   'Back space
                Case Else
                        keyascii = 0       'Reject all other characters
        End Select
End Sub

Function Max (ByVal a As Long, ByVal b As Long) As Long
        If a  b Then
                Max = a
        Else
                Max = b
        End If
End Function
```

CTR2



CTR2 Form

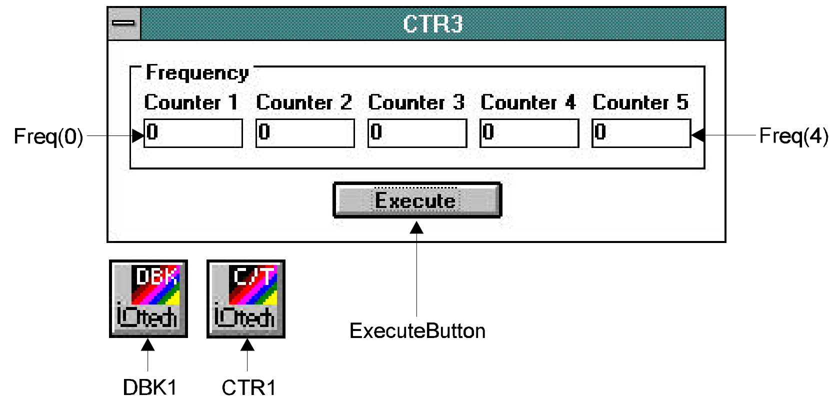
```
Begin CTR Ctr1
        BufferLength    =       1
        C1CntDir        =       0       'Count Down
        C1CntEdge       =       0       'Negative Count Edge
        C1CntRepeat     =       0        'False
        C1CntSource     =       0       ' 0 - TC  toggled output of last ctr
        C1CntType       =       0       'Binary Count
        C1Enable        =       0       'False
        C1GateCtrl      =       0       'Gating Disabled
        C1OutputCtrl    =       0       'Inactive -Always low
        C1Reload        =       0       'Reload from Load
        C1SpecialGate   =       0       'False
        C2CntDir        =       0       'Count Down
        C2CntEdge       =       0       'Negative Count Edge
        C2CntRepeat     =       0        'False
        C2CntSource     =       0       ' 0 - TC toggled output of last ctr
        C2CntType       =       0       'Binary Count
        C2Enable        =       0       'False
        C2GateCtrl      =       0       'Gating Disabled
        C2OutputCtrl    =       0       'Inactive -Always low
        C2Reload        =       0       'Reload from Load
        C2SpecialGate   =       0       'False
        C3CntDir        =       0       'Count Down
        C3CntEdge       =       0       'Negative Count Edge
        C3CntRepeat     =       0        'False
        C3CntSource     =       0       ' 0 - TC toggled output of last ctr
        C3CntType       =       0       'Binary Count
        C3Enable        =       0       'False
        C3GateCtrl      =       0       'Gating Disabled
        C3OutputCtrl    =       0       'Inactive -Always low
```

```
        C3Reload        =    0     'Reload from Load
        C3SpecialGate   =    0     'False
        C4CntDir        =    0     'Count Down
        C4CntEdge       =    0     'Negative Count Edge
        C4CntRepeat     =    0      'False
        C4CntSource     =    0     ' 0 - TC toggled output of last ctr
        C4CntType       =    0     'Binary Count
        C4Enable        =    0     'False
        C4GateCtrl      =    0     'Gating Disabled
        C4OutputCtrl    =    0     'Inactive -Always low
        C4Reload        =    0     'Reload from Load
        C4SpecialGate   =    0     'False
        C5CntDir        =    0     'Count Down
        C5CntEdge       =    0     'Negative Count Edge
        C5CntRepeat     =    0      'False
        C5CntSource     =    0     ' 0 - TC toggled output of last ctr
        C5CntType       =    0     'Binary Count
        C5Enable        =    0     'False
        C5GateCtrl      =    0     'Gating Disabled
        C5OutputCtrl    =    0     'Inactive -Always low
        C5Reload        =    0     'Reload from Load
        C5SpecialGate   =    0     'False
        Comp1Enable     =    0      'False
        Comp2Enable     =    0      'False
        FoutDivider     =    1     ' Divide by  1
        FoutSource      =    15    '100 Hz  Clock
        FreqCntSource   =    1     'Counter 1 Input
        FreqInterval    =    1
        Left            =    600
        NumScans        =    1
        TimeOfDay       =    0     'Disabled
        Top             =    1500
End
Begin DBK Dbk1
        IntLevel        =    7
        Left            =    180
        LptPort         =    0     'LPT1
        Protocol        =    1     '4 Bit I/O
        Top             =    1500
End

Sub ExecuteButton_Click ()
                ' enable all counters to execute the save command
        ctr1.C1Enable = True
        ctr1.C2Enable = True
        ctr1.C3Enable = True
        ctr1.C4Enable = True
        ctr1.C5Enable = True
                ' save the count value to the hold register
        ctr1.Save = True
                ' print the contents of the hold register
        Total(0).Caption = CStr(ctr1.C1Hold)
        Total(1).Caption = CStr(ctr1.C2Hold)
        Total(2).Caption = CStr(ctr1.C3Hold)
        Total(3).Caption = CStr(ctr1.C4Hold)
        Total(4).Caption = CStr(ctr1.C5Hold)
End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.Open = True
        ' enable all counters to execute the disarm/arm and load commands
        ctr1.C1Enable = True
        ctr1.C2Enable = True
        ctr1.C3Enable = True
        ctr1.C4Enable = True
        ctr1.C5Enable = True
                ' Halt all counters
        ctr1.Disarm = True
                ' Set non-changing properties of each counter
                ' These properties are set here and never change
                ' Alternatively, they could be set in the properties window
        ctr1.C1CntDir = 1          ' count up
        ctr1.C1CntEdge = 1         ' count on positive edge
```

```
        ctr1.C1CntRepeat = True      ' repeat enabled
        ctr1.C1CntType = 0           ' binary counting
        ctr1.C1GateCtrl = 0          ' no gating
        ctr1.C1OutputCtrl = 3        ' output disabled, high impedance
        ctr1.C1Reload = 0            ' reload from load
        ctr1.C1SpecialGate = 0       ' special gate disabled
        ctr1.C1CntSource = 1         ' use counter 1 input as source
        ctr1.C1Load = 0              ' initial load register value is 0
        ctr1.C2CntDir = 1            ' count up
        ctr1.C2CntEdge = 1           ' count on positive edge
        ctr1.C2CntRepeat = True      ' repeat enabled
        ctr1.C2CntType = 0           ' binary counting
        ctr1.C2GateCtrl = 0          ' no gating
        ctr1.C2OutputCtrl = 3        ' output disabled, high impedance
        ctr1.C2Reload = 0            ' reload from load
        ctr1.C2SpecialGate = 0       ' special gate disabled
        ctr1.C2CntSource = 2         ' use counter 1 input as source
        ctr1.C2Load = 0              ' initial load register value is 0
        ctr1.C3CntDir = 1            ' count up
        ctr1.C3CntEdge = 1           ' count on positive edge
        ctr1.C3CntRepeat = True      ' repeat enabled
        ctr1.C3CntType = 0           ' binary counting
        ctr1.C3GateCtrl = 0          ' no gating
        ctr1.C3OutputCtrl = 3        ' output disabled, high impedance
        ctr1.C3Reload = 0            ' reload from load
        ctr1.C3SpecialGate = 0       ' special gate disabled
        ctr1.C3CntSource = 3         ' use counter 1 input as source
        ctr1.C3Load = 0              ' initial load register value is 0
        ctr1.C4CntDir = 1            ' count up
        ctr1.C4CntEdge = 1           ' count on positive edge
        ctr1.C4CntRepeat = True      ' repeat enabled
        ctr1.C4CntType = 0           ' binary counting
        ctr1.C4GateCtrl = 0          ' no gating
        ctr1.C4OutputCtrl = 3        ' output disabled, high impedance
        ctr1.C4Reload = 0            ' reload from load
        ctr1.C4SpecialGate = 0       ' special gate disabled
        ctr1.C4CntSource = 4         ' use counter 1 input as source
        ctr1.C4Load = 0              ' initial load  register value is 0
        ctr1.C5CntDir = 1            ' count up
        ctr1.C5CntEdge = 1           ' count on positive edge
        ctr1.C5CntRepeat = True      ' repeat enabled
        ctr1.C5CntType = 0           ' binary counting
        ctr1.C5GateCtrl = 0          ' no gating
        ctr1.C5OutputCtrl = 3        ' output disabled, high impedance
        ctr1.C5Reload = 0            ' reload from load
        ctr1.C5SpecialGate = 0       ' special gate disabled
        ctr1.C5CntSource = 5         ' use counter 1 input as source
        ctr1.C5Load = 0              ' initial load register value is 0
                ' program the counters with the previous set parameters
        ctr1.SetCounterMode = 1
        ctr1.SetCounterMode = 2
        ctr1.SetCounterMode = 3
        ctr1.SetCounterMode = 4
        ctr1.SetCounterMode = 5
                ' initialize the counter values and start the counters
        ctr1.LoadArm = True
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub

Sub ResetButton_Click (Index As Integer)
                ' Reset the corresponding counter's load register
        ctr1.C1Enable = False
        ctr1.C2Enable = False
        ctr1.C3Enable = False
        ctr1.C4Enable = False
        ctr1.C5Enable = False
        Select Case Index
        Case 0
                ctr1.C1Enable = True
        Case 1
                ctr1.C2Enable = True
```

```
        Case 2
                ctr1.C3Enable = True
        Case 3
                ctr1.C4Enable = True
        Case 4
                ctr1.C5Enable = True
        End Select
        ctr1.Load = True
End Sub
```

CTR3



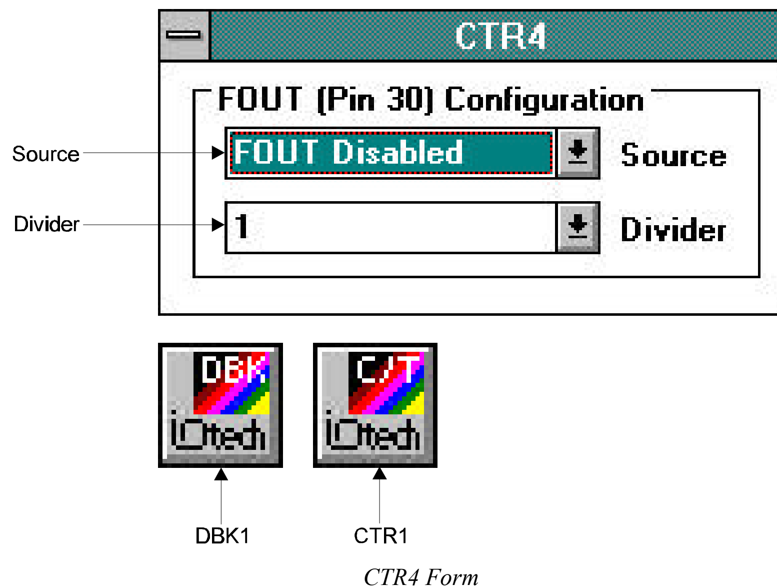
CTR3 Form

```
Begin CTR Ctr1
        BufferLength    =    1
        C1CntDir        =    0    'Count Down
        C1CntEdge       =    0    'Negative Count Edge
        C1CntRepeat     =    0     'False
        C1CntSource     =    0    ' 0 - TC toggled output of last ctr
        C1CntType       =    0    'Binary Count
        C1Enable        =    0    'False
        C1GateCtrl      =    0    'Gating Disabled
        C1OutputCtrl    =    0    'Inactive -Always low
        C1Reload        =    0    'Reload from Load
        C1SpecialGate   =    0    'False
        C2CntDir        =    0    'Count Down
        C2CntEdge       =    0    'Negative Count Edge
        C2CntRepeat     =    0     'False
        C2CntSource     =    0    ' 0 - TC toggled output of last ctr
        C2CntType       =    0    'Binary Count
        C2Enable        =    0    'False
        C2GateCtrl      =    0    'Gating Disabled
        C2OutputCtrl    =    0    'Inactive -Always low
        C2Reload        =    0    'Reload from Load
        C2SpecialGate   =    0    'False
        C3CntDir        =    0    'Count Down
        C3CntEdge       =    0    'Negative Count Edge
        C3CntRepeat     =    0     'False
        C3CntSource     =    0    ' 0 - TC toggled output of last ctr
        C3CntType       =    0    'Binary Count
        C3Enable        =    0    'False
        C3GateCtrl      =    0    'Gating Disabled
        C3OutputCtrl    =    0    'Inactive -Always low
        C3Reload        =    0    'Reload from Load
        C3SpecialGate   =    0    'False
        C4CntDir        =    0    'Count Down
        C4CntEdge       =    0    'Negative Count Edge
        C4CntRepeat     =    0     'False
        C4CntSource     =    0    ' 0 - TC toggled output of last ctr
        C4CntType       =    0    'Binary Count
        C4Enable        =    0    'False
```

```
		C4GateCtrl	=	0	'Gating Disabled
		C4OutputCtrl	=	0	'Inactive -Always low
		C4Reload	=	0	'Reload from Load
		C4SpecialGate	=	0	'False
		C5CntDir	=	0	'Count Down
		C5CntEdge	=	0	'Negative Count Edge
		C5CntRepeat	=	0	 'False
		C5CntSource	=	0	' 0 - TC toggled output of last ctr
		C5CntType	=	0	'Binary Count
		C5Enable	=	0	'False
		C5GateCtrl	=	0	'Gating Disabled
		C5OutputCtrl	=	0	'Inactive -Always low
		C5Reload	=	0	'Reload from Load
		C5SpecialGate	=	0	'False
		Comp1Enable	=	0	 'False
		Comp2Enable	=	0	 'False
		FoutDivider	=	0	' Divide by 16
		FoutSource	=	0	' 0 - Fout Disabled
		FreqCntSource	=	1	'Counter 1 Input
		FreqInterval	=	1
		Left	=	600
		NumScans	=	1
		TimeOfDay	=	0	'Disabled
		Top	=	1140
End
Begin DBK Dbk1
		IntLevel	=	7
		Left	=	180
		LptPort	=	0	'LPT1
		Protocol	=	1	'4 Bit I/O
		Top	=	1140
End

Sub ExecuteButton_Click ()
	Dim i As Integer
	' Setup counter 1 to read a known source (1MHz) that will be used as a timebase for counter 2.
	   Counter 2 will read the frequency of all 5 counter inputs.  This source can be set to
	   frequencies slower than 1MHz to read slower frequencies.
	ctr1.C1CntSource = 11
	ctr1.SetCounterMode = 1
	For i = 1 To 5
			' Halt counters 1 and 2
		ctr1.Disarm = True
			' Program counter 2 to read the current source
		ctr1.C2CntSource = i
		ctr1.SetCounterMode = 2
			' Reset the counters 1 and 2 to 0 and start counting
		ctr1.LoadArm = True
			' Wait for counter 1 to accumulate  10000 counts
			' At 1MHz, this translates to 0.01 seconds
		Do
			' Transfer the count value of counters 1 and 2
			' to the hold register
			ctr1.Save = True
		Loop While ctr1.C1Hold  10000
		' Now use the known timebase (counter 1) to calculate the frequency of the unknown
		   timebase (counter 2).  If an input other than 1MHz is used for counter 1, change the
		   constant 1000000 in the following line to the input frequency of counter 1
		Freq(i - 1).Caption = CStr(ctr1.C2Hold * 1000000 / ctr1.C1Hold)
	Next i
End Sub

Sub Form_Load ()
		' Open the DaqBook driver
	dbk1.Open = True
		'enable counters 1 and 2 to execute the disarm/arm and load commands
	ctr1.C1Enable = True
	ctr1.C2Enable = True
	ctr1.C3Enable = False
	ctr1.C4Enable = False
	ctr1.C5Enable = False
		' Set non-changing properties of each counter
		' These properties are set here and never change
		' Alternatively, they could be set in the properties window
```

```
        ctr1.C1CntDir = 1              ' count up
        ctr1.C1CntEdge = 1             ' count on positive edge
        ctr1.C1CntRepeat = False       ' repeat enabled
        ctr1.C1CntType = 0             ' binary counting
        ctr1.C1GateCtrl = 0            ' no gating
        ctr1.C1OutputCtrl = 3          ' output disabled, high impedance
        ctr1.C1Reload = 0              ' reload from load
        ctr1.C1SpecialGate = 0         ' special gate disabled
        ctr1.C1CntSource = 11          ' use 1MHz clock as source
        ctr1.C1Load = 0                ' initial load register value is 0
        ctr1.C2CntDir = 1              ' count up
        ctr1.C2CntEdge = 1             ' count on positive edge
        ctr1.C2CntRepeat = False       ' repeat enabled
        ctr1.C2CntType = 0             ' binary counting
        ctr1.C2GateCtrl = 0            ' no gating
        ctr1.C2OutputCtrl = 3          ' output disabled, high impedance
        ctr1.C2Reload = 0              ' reload from load
        ctr1.C2SpecialGate = 0         ' special gate disabled
        ctr1.C2Load = 0                ' initial load register value is 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub
```

CTR4



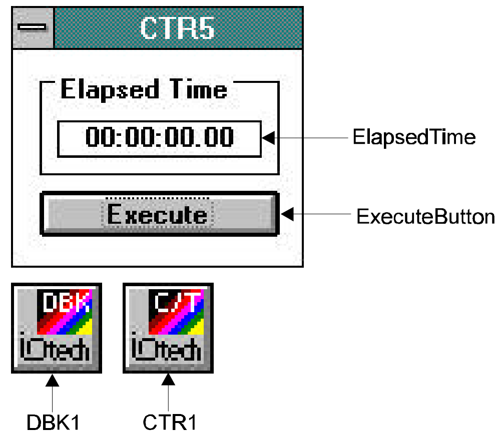
CTR4 Form

```
Begin CTR Ctr1
        BufferLength     =     1
        C1CntDir         =     0      'Count Down
        C1CntEdge        =     0      'Negative Count Edge
        C1CntRepeat      =     0       'False
        C1CntSource      =     0      ' 0 - TC toggled output of last ctr
        C1CntType        =     0      'Binary Count
        C1Enable         =     0      'False
        C1GateCtrl       =     0      'Gating Disabled
        C1OutputCtrl     =     0      'Inactive -Always low
        C1Reload         =     0      'Reload from Load
        C1SpecialGate    =     0      'False
        C2CntDir         =     0      'Count Down
        C2CntEdge        =     0      'Negative Count Edge
        C2CntRepeat      =     0       'False
        C2CntSource      =     0      ' 0 - TC toggled output of last ctr
        C2CntType        =     0      'Binary Count
        C2Enable         =     0      'False
```

```
        C2GateCtrl      =   0     'Gating Disabled
        C2OutputCtrl    =   0     'Inactive -Always low
        C2Reload        =   0     'Reload from Load
        C2SpecialGate   =   0     'False
        C3CntDir        =   0     'Count Down
        C3CntEdge       =   0     'Negative Count Edge
        C3CntRepeat     =   0      'False
        C3CntSource     =   0     ' 0 - TC toggled output of last ctr
        C3CntType       =   0     'Binary Count
        C3Enable        =   0     'False
        C3GateCtrl      =   0     'Gating Disabled
        C3OutputCtrl    =   0     'Inactive -Always low
        C3Reload        =   0     'Reload from Load
        C3SpecialGate   =   0     'False
        C4CntDir        =   0     'Count Down
        C4CntEdge       =   0     'Negative Count Edge
        C4CntRepeat     =   0      'False
        C4CntSource     =   0     ' 0 - TC toggled output of last ctr
        C4CntType       =   0     'Binary Count
        C4Enable        =   0     'False
        C4GateCtrl      =   0     'Gating Disabled
        C4OutputCtrl    =   0     'Inactive -Always low
        C4Reload        =   0     'Reload from Load
        C4SpecialGate   =   0     'False
        C5CntDir        =   0     'Count Down
        C5CntEdge       =   0     'Negative Count Edge
        C5CntRepeat     =   0      'False
        C5CntSource     =   0     ' 0 - TC toggled output of last ctr
        C5CntType       =   0     'Binary Count
        C5Enable        =   0     'False
        C5GateCtrl      =   0     'Gating Disabled
        C5OutputCtrl    =   0     'Inactive -Always low
        C5Reload        =   0     'Reload from Load
        C5SpecialGate   =   0     'False
        Comp1Enable     =   0      'False
        Comp2Enable     =   0      'False
        FoutDivider     =   0     ' Divide by 16
        FoutSource      =   0     ' 0 - Fout Disabled
        FreqCntSource   =   1     'Counter 1  Input
        FreqInterval    =   1
        Left            =   600
        NumScans        =   1
        TimeOfDay       =   0     'Disabled
        Top             =   1260
End
Begin DBK Dbk1
        IntLevel        =   7
        Left            =   180
        LptPort         =   0     'LPT1
        Protocol        =   1     '4 Bit I/O
        Top             =   1260
End

Sub Divider_Click ()
        ctr1.FoutDivider = Divider.ListIndex
        ctr1.SetMasterMode = True
End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.open = True
                ' initialize the selection lists
        Dim i As Integer
        For i = 1 To 16
                Divider.AddItem CStr(i)
        Next i
        Source.AddItem "FOUT Disabled"
        Source.AddItem "Counter 1 Input"
        Source.AddItem "Counter 2 Input"
        Source.AddItem "Counter 3 Input"
        Source.AddItem "Counter 4 Input"
        Source.AddItem "Counter 5 Input"
        Source.AddItem "Counter 1 Gate"
        Source.AddItem "Counter 2 Gate"
```

```
        Source.AddItem "Counter 3 Gate"
        Source.AddItem "Counter 4 Gate"
        Source.AddItem "Counter 5 Gate"
        Source.AddItem "1MHz"
        Source.AddItem "100kHz"
        Source.AddItem "10kHz"
        Source.AddItem "1kHz"
        Source.AddItem "100Hz"
                ' select initial settings
        Divider.ListIndex = 0
        Source.ListIndex = 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.open = False
End Sub

Sub Source_Click ()
        ctr1.FoutSource = Source.ListIndex
        ctr1.SetMasterMode = True
End Sub
```

CTR5



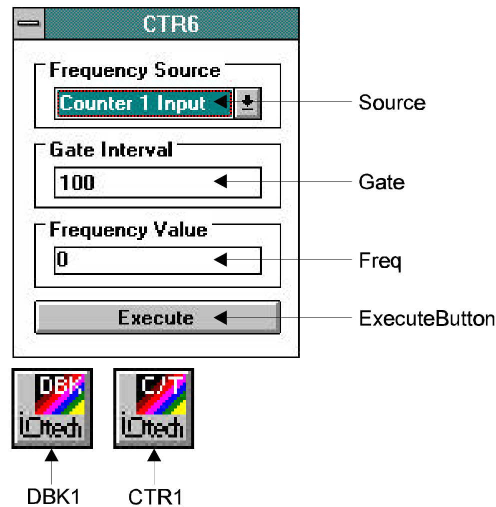
CTR5 Form

```
Begin CTR Ctr1
        BufferLength     =     1
        C1CntDir         =     0     'Count Down
        C1CntEdge        =     0     'Negative Count Edge
        C1CntRepeat      =     0      'False
        C1CntSource      =     0     ' 0 - TC toggled output of last ctr
        C1CntType        =     0     'Binary Count
        C1Enable         =     0     'False
        C1GateCtrl       =     0     'Gating Disabled
        C1OutputCtrl     =     0     'Inactive -Always low
        C1Reload         =     0     'Reload from Load
        C1SpecialGate    =     0     'False
        C2CntDir         =     0     'Count Down
        C2CntEdge        =     0     'Negative Count Edge
        C2CntRepeat      =     0      'False
        C2CntSource      =     0     ' 0 - TC toggled output of last ctr
        C2CntType        =     0     'Binary Count
        C2Enable         =     0     'False
        C2GateCtrl       =     0     'Gating Disabled
        C2OutputCtrl     =     0     'Inactive -Always low
        C2Reload         =     0     'Reload from Load
        C2SpecialGate    =     0     'False
        C3CntDir         =     0     'Count Down
        C3CntEdge        =     0     'Negative Count Edge
        C3CntRepeat      =     0      'False
        C3CntSource      =     0     ' 0 - TC toggled output of last ctr
```

```
        C3CntType       =       0       'Binary Count
        C3Enable        =       0       'False
        C3GateCtrl      =       0       'Gating Disabled
        C3OutputCtrl    =       0       'Inactive  -Always low
        C3Reload        =       0       'Reload from Load
        C3SpecialGate   =       0       'False
        C4CntDir        =       0       'Count Down
        C4CntEdge       =       0       'Negative Count Edge
        C4CntRepeat     =       0        'False
        C4CntSource     =       0       ' 0 - TC toggled output of last ctr
        C4CntType       =       0       'Binary Count
        C4Enable        =       0       'False
        C4GateCtrl      =       0       'Gating Disabled
        C4OutputCtrl    =       0       'Inactive -Always low
        C4Reload        =       0       'Reload from Load
        C4SpecialGate   =       0       'False
        C5CntDir        =       0       'Count Down
        C5CntEdge       =       0       'Negative Count Edge
        C5CntRepeat     =       0        'False
        C5CntSource     =       0       ' 0 - TC toggled output of last ctr
        C5CntType       =       0       'Binary Count
        C5Enable        =       0       'False
        C5GateCtrl      =       0       'Gating Disabled
        C5OutputCtrl    =       0       'Inactive -Always low
        C5Reload        =       0       'Reload from Load
        C5SpecialGate   =       0       'False
        Comp1Enable     =       0        'False
        Comp2Enable     =       0        'False
        FoutDivider     =       0       ' Divide by 16
        FoutSource      =       0       ' 0 - Fout Disabled
        FreqCntSource   =       1       'Counter 1 Input
        FreqInterval    =       1
        Left            =       600
        NumScans        =       1
        TimeOfDay       =       0       'Disabled
        Top             =       1320
End
Begin DBK Dbk1
        IntLevel        =       7
        Left            =       180
        LptPort         =       0       'LPT1
        Protocol        =       1       '4 Bit I/O

Sub ExecuteButton_Click ()
                ' Transfer count value of counters 1 and 2 to the hold register
        ctr1.Save = True
                ' Display the elapsed time
        ElapsedTime.Caption = Format$(Hex$(ctr1.C2Hold), "00:00:") +
Format$(Hex$(ctr1.C1Hold), "00\.00")
End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.Open = True
        'enable counters 1 and 2 to execute the disarm/arm and load commands
        ctr1.C1Enable = True
        ctr1.C2Enable = True
        ctr1.C3Enable = False
        ctr1.C4Enable = False
        ctr1.C5Enable = False
                ' halt counters 1 and 2
        ctr1.Disarm = True
                ' Initialize time of day operation
                ' Use 100Hz time of day setting and set the input
                ' of counter 1 (below) to the internal 100Hz clock
        ctr1.TimeOfDay = 3
        ctr1.SetMasterMode = True
                ' Set non-changing properties of each counter
                ' These properties are set here and never change
                ' Alternatively, they could be set in the properties window
        ctr1.C1CntDir = 1     ' count up
        ctr1.C1CntEdge = 1 ' count on positive edge
        ctr1.C1CntRepeat = True        ' repeat enabled
        ctr1.C1CntType = 1             ' BCD counting
```

```
        ctr1.C1GateCtrl = 0         ' no gating
        ctr1.C1OutputCtrl = 3       ' output disabled, high impedance
        ctr1.C1Reload = 0           ' reload from load
        ctr1.C1SpecialGate = 0      ' special gate disabled
        ctr1.C1CntSource = 15       ' use 100Hz as source
        ctr1.C2CntDir = 1           ' count up
        ctr1.C2CntEdge = 1          ' count on positive edge
        ctr1.C2CntRepeat = True     ' repeat enabled
        ctr1.C2CntType = 1          ' BCD counting
        ctr1.C2GateCtrl = 0         ' no gating
        ctr1.C2OutputCtrl = 3       ' output disabled, high impedance
        ctr1.C2Reload = 0           ' reload from load
        ctr1.C2SpecialGate = 0      ' special gate disabled
        ctr1.C2CntSource = 0        ' use the TC output of the
                                    ' previous counter (counter 1) as source
                ' Program counters 1 and 2
        ctr1.SetCounterMode = 1
        ctr1.SetCounterMode = 2
                ' Initialize counters 1 and 2 to 0
                ' This will set counters 1 and 2 to read the elapsed time
                ' from the start of this program
        ctr1.C1Load = 0
        ctr1.C2Load = 0
        ctr1.LoadArm = True
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub
```

CTR6



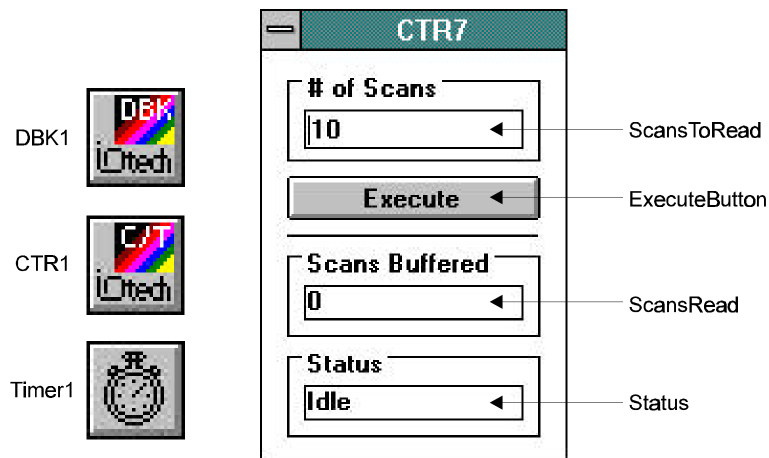
CTR6 Form

```
Begin CTR Ctr1
        BufferLength        =       1
        C1CntDir            =       0       'Count Down
        C1CntEdge           =       0       'Negative Count Edge
        C1CntRepeat         =       0        'False
        C1CntSource         =       0       ' 0 - TC toggled output of last ctr
        C1CntType           =       0       'Binary Count
        C1Enable            =       0       'False
        C1GateCtrl          =       0       'Gating Disabled
        C1OutputCtrl        =       0       'Inactive -Always low
        C1Reload            =       0       'Reload from Load
        C1SpecialGate       =       0       'False
        C2CntDir            =       0       'Count Down
        C2CntEdge           =       0       'Negative Count Edge
        C2CntRepeat         =       0        'False
```

```
                    C2CntSource     =       0       ' 0 - TC toggled output of last ctr
                    C2CntType       =       0       'Binary Count
                    C2Enable        =       0       'False
                    C2GateCtrl      =       0       'Gating Disabled
                    C2OutputCtrl    =       0       'Inactive -Always low
                    C2Reload        =       0       'Reload from Load
                    C2SpecialGate   =       0       'False
                    C3CntDir        =       0       'Count Down
                    C3CntEdge       =       0       'Negative Count Edge
                    C3CntRepeat     =       0        'False
                    C3CntSource     =       0       ' 0 - TC toggled output of last ctr
                    C3CntType       =       0       'Binary Count
                    C3Enable        =       0       'False
                    C3GateCtrl      =       0       'Gating Disabled
                    C3OutputCtrl    =       0       'Inactive -Always low
                    C3Reload        =       0       'Reload from Load
                    C3SpecialGate   =       0       'False
                    C4CntDir        =       0       'Count Down
                    C4CntEdge       =       0       'Negative Count Edge
                    C4CntRepeat     =       0        'False
                    C4CntSource     =       0       ' 0 - TC toggled output of last ctr
                    C4CntType       =       0       'Binary Count
                    C4Enable        =       0       'False
                    C4GateCtrl      =       0       'Gating  Disabled
                    C4OutputCtrl    =       0       'Inactive -Always low
                    C4Reload        =       0       'Reload from Load
                    C4SpecialGate   =       0       'False
                    C5CntDir        =       0       'Count Down
                    C5CntEdge       =       0       'Negative Count Edge
                    C5CntRepeat     =       0        'False
                    C5CntSource     =       0       ' 0 - TC toggled output of last ctr
                    C5CntType       =       0       'Binary Count
                    C5Enable        =       0       'False
                    C5GateCtrl      =       0       'Gating Disabled
                    C5OutputCtrl    =       0       'Inactive -Always low
                    C5Reload        =       0       'Reload from Load
                    C5SpecialGate   =       0       'False
                    Comp1Enable     =       0        'False
                    Comp2Enable     =       0        'False
                    FoutDivider     =       0       ' Divide by 16
                    FoutSource      =       0       ' 0 - Fout Disabled
                    FreqCntSource   =       1       'Counter 1 Input
                    FreqInterval    =       1
                    Left            =       600
                    NumScans        =       1
                    TimeOfDay       =       0       'Disabled
                    Top             =       2700
            End
            Begin DBK Dbk1
                    IntLevel        =       7
                    Left            =       180
                    LptPort         =       0       'LPT1
                    Protocol        =       1       '4 Bit I/O
                    Top             =       2700
            End

Function CvtMinMax (ByVal CvtStr As String, ByVal min As Long, ByVal Max As Long) As Long
                            ' converts a string to a long integer number
                            ' and makes sure that the number is within bounds
                    If CvtStr = “” Then
                            CvtMinMax = min
                    Else
                            Dim CvtVal As Long
                            CvtVal = CLng(CvtStr)
                            If CvtVal  Max Then
                                    CvtVal = Max
                            ElseIf CvtVal  min Then
                                    CvtVal = min
                            End If
                            CvtMinMax = CvtVal
                    End If
            End Function

            Sub ExecuteButton_Click ()
```

```
        Dim i As Integer
                ' Read the user inputs
        ctr1.FreqCntSource = Source.ListIndex + 1
        i = CInt(CvtMinMax(Gate.Text, 1, 32767))
        ctr1.FreqInterval = i
                ' Display the frequency
        MousePointer = 11
        Freq.Caption = CStr(ctr1.FreqCnt * 1000 / i)
        MousePointer = 0
End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.Open = True
                ' Initialize source selection box
        Source.AddItem "Counter 1 Input"
        Source.AddItem "Counter 2 Input"
        Source.AddItem "Counter 3 Input"
        Source.AddItem "Counter 4 Input"
        Source.AddItem "Counter 5 Input"
        Source.AddItem "Counter 1 Gate"
        Source.AddItem "Counter 2 Gate"
        Source.AddItem "Counter 3 Gate"
        Source.AddItem "Counter 4 Gate"
                ' select initial settings
        Source.ListIndex = 0
End Sub

Sub Form_Unload (Cancel As Integer)
        dbk1.Open = False
End Sub

Sub Gate_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39           'All  numbers
                Case &H8                            'Back space
                Case Else
                        keyascii = 0        'Reject all other characters
        End Select
End Sub
```

CTR7




```
VERSION 2.00
Begin CTR Ctr1
        BufferLength    =       1
        C1CntDir        =       0       'Count Down
        C1CntEdge       =       0       'Negative Count Edge
        C1CntRepeat     =       0        'False
```

```
        C1CntSource      =    0      ' 0 - TC toggled output of last ctr
        C1CntType        =    0      'Binary Count
        C1Enable         =    0      'False
        C1GateCtrl       =    0      'Gating Disabled
        C1OutputCtrl     =    0      'Inactive -Always low
        C1Reload         =    0      'Reload from Load
        C1SpecialGate    =    0      'False
        C2CntDir         =    0      'Count Down
        C2CntEdge        =    0      'Negative Count Edge
        C2CntRepeat      =    0       'False
        C2CntSource      =    0      ' 0 - TC toggled output of last ctr
        C2CntType        =    0      'Binary Count
        C2Enable         =    0      'False
        C2GateCtrl       =    0      'Gating Disabled
        C2OutputCtrl     =    0      'Inactive -Always low
        C2Reload         =    0      'Reload from Load
        C2SpecialGate    =    0      'False
        C3CntDir         =    0      'Count Down
        C3CntEdge        =    0      'Negative Count Edge
        C3CntRepeat      =    0       'False
        C3CntSource      =    0      ' 0 - TC toggled output of last ctr
        C3CntType        =    0      'Binary Count
        C3Enable         =    0      'False
        C3GateCtrl       =    0      'Gating Disabled
        C3OutputCtrl     =    0      'Inactive -Always low
        C3Reload         =    0      'Reload from Load
        C3SpecialGate    =    0      'False
        C4CntDir         =    0      'Count Down
        C4CntEdge        =    0      'Negative Count Edge
        C4CntRepeat      =    0       'False
        C4CntSource      =    0      ' 0 - TC toggled output of last ctr
        C4CntType        =    0      'Binary Count
        C4Enable         =    0      'False
        C4GateCtrl       =    0      'Gating Disabled
        C4OutputCtrl     =    0      'Inactive -Always low
        C4Reload         =    0      'Reload from Load
        C4SpecialGate    =    0      'False
        C5CntDir         =    0      'Count Down
        C5CntEdge        =    0      'Negative Count Edge
        C5CntRepeat      =    0       'False
        C5CntSource      =    0      ' 0 - TC toggled output of last ctr
        C5CntType        =    0      'Binary Count
        C5Enable         =    0      'False
        C5GateCtrl       =    0      'Gating Disabled
        C5OutputCtrl     =    0      'Inactive -Always low
        C5Reload         =    0      'Reload from Load
        C5SpecialGate    =    0      'False
        Comp1Enable      =    0       'False
        Comp2Enable      =    0       'False
        FoutDivider      =    0      ' Divide by 16
        FoutSource       =    0      ' 0 - Fout Disabled
        FreqCntSource    =    1      'Counter 1 Input
        FreqInterval     =    1
        Left             =    600
        NumScans         =    1
        TimeOfDay        =    0      'Disabled
        Top              =    2820
End
Begin DBK Dbk1
        IntLevel         =    7
        Left             =    180
        LptPort          =    0      'LPT1
        Protocol         =    1      '4 Bit I/O
        Top              =    2820
End

Const MAXBUF = 160

Dim dataBuffer1(MAXBUF) As Integer    ' The data buffer for counter 1
Dim dataBuffer2(MAXBUF) As Integer    ' The data buffer for counter 1
Dim dataBuffer3(MAXBUF) As Integer    ' The data buffer for counter 1
Dim dataBuffer4(MAXBUF) As Integer    ' The data buffer for counter 1
Dim dataBuffer5(MAXBUF) As Integer    ' The data buffer for counter 1
Dim scansProcessed As Long' Keeps track of how many scans
```

```
                                        ' read have been sent to disk
Dim totalScansToRead As Long            ' The total number of scans to read
Dim readIndex As Integer        ' The array index from which to read the next value

Sub Ctr1_Triggered ()
                ' Update the status box
        Status.Caption = "Triggered"
                ' Enable the timer that checks for new data
        Timer1.Enabled = True
End Sub

Function CvtMinMax (ByVal CvtStr As String, ByVal min As Long, ByVal Max As Long) As Long
                ' converts a string to a long integer number
                ' and makes sure that the number is within bounds
        If CvtStr = "" Then
                CvtMinMax = min
        Else
                Dim CvtVal As Long
                CvtVal = CLng(CvtStr)
                If CvtVal  Max Then
                        CvtVal = Max
                ElseIf CvtVal  min Then
                        CvtVal = min
                End If
                CvtMinMax = CvtVal
        End If
End Function

Sub disarmAcq ()
        Close #1
                'ctr1.ReadCounters = false
        ctr1.Stop = True
        Status.Caption = "Idle"
        ExecuteButton.Caption = "Execute"
        Timer1.Enabled = False
        ScansToRead.Enabled = True
End Sub

Sub ExecuteButton_Click ()
        ' Disarm the  acquisition if the acquisition is already in process
        If ExecuteButton.Caption = "Abort" Then
                disarmAcq
                Exit Sub
        End If
                ' halt the counters
        ctr1.Disarm = True
                ' Update the user interface
        Status.Caption = "Waiting for trigger"
        ScansToRead.Enabled = False
                ' Initialize the acquisition variables
        ExecuteButton.Caption = "Abort"
        readIndex = 0
        scansProcessed = 0
        totalScansToRead = CvtMinMax(ScansToRead.Text, 1, 1000000)
        If totalScansToRead  MAXBUF Then
                ctr1.NumScans = -1
        Else
                ctr1.NumScans = totalScansToRead
        End If
                ' Open the data file
        Open "ctr7.txt" For Output As #1
                ' start the counters
        ctr1.LoadArm = True
        ' Start the acquisition
        ctr1.ReadCounters = True

End Sub

Sub Form_Load ()
                ' Open the DaqBook driver
        dbk1.Open = True
        ' enable all counters to execute the disarm/arm and load commands
        ctr1.C1Enable = True
        ctr1.C2Enable = True
```

```
ctr1.C3Enable = True
ctr1.C4Enable = True
ctr1.C5Enable = True
        ' Halt all counters
ctr1.Disarm = True
        ' Set non-changing properties of each counter
        ' These properties are set here and never change
        ' Alternatively, they could be set in the properties window
ctr1.BufferLength = MAXBUF
ctr1.C1CntDir = 1               ' count up
ctr1.C1CntEdge = 1              ' count on positive edge
ctr1.C1CntRepeat = True         ' repeat enabled
ctr1.C1CntType = 0              ' binary counting
ctr1.C1GateCtrl = 0             ' no gating
ctr1.C1OutputCtrl = 3           ' output disabled, high impedance
ctr1.C1Reload = 0               ' reload from load
ctr1.C1SpecialGate = 0          ' special gate disabled
ctr1.C1CntSource = 1            ' use counter 1 input as source
ctr1.C1Load = 0                 ' initial load register value is 0
ctr1.C1Buffer = addressOf(dataBuffer1(0))
ctr1.C2CntDir = 1               ' count up
ctr1.C2CntEdge = 1              'count on positive edge
ctr1.C2CntRepeat = True         ' repeat enabled
ctr1.C2CntType = 0              ' binary counting
ctr1.C2GateCtrl = 0             ' no gating
ctr1.C2OutputCtrl = 3           ' output disabled, high impedance
ctr1.C2Reload = 0               ' reload from load
ctr1.C2SpecialGate = 0          ' special gate disabled
ctr1.C2CntSource = 2            ' use counter 2 input as source
ctr1.C2Load = 0                 ' initial load register value is 0
ctr1.C2Buffer = addressOf(dataBuffer2(0))
ctr1.C3CntDir = 1               ' count up
ctr1.C3CntEdge = 1              ' count on positive edge
ctr1.C3CntRepeat = True         ' repeat enabled
ctr1.C3CntType = 0              ' binary counting
ctr1.C3GateCtrl = 0             ' no gating
ctr1.C3OutputCtrl = 3           ' output disabled, high impedance
ctr1.C3Reload = 0               ' reload from load
ctr1.C3SpecialGate = 0          ' special gate disabled
ctr1.C3CntSource = 3            ' use counter 3 input as source
ctr1.C3Load = 0                 ' initial load register value is 0
ctr1.C3Buffer = addressOf(dataBuffer3(0))
ctr1.C4CntDir = 1               ' count up
ctr1.C4CntEdge = 1              ' count on positive edge
ctr1.C4CntRepeat = True         ' repeat enabled
ctr1.C4CntType = 0              ' binary counting
ctr1.C4GateCtrl = 0             ' no gating
ctr1.C4OutputCtrl = 3           ' output disabled, high impedance
ctr1.C4Reload = 0               ' reload from load
ctr1.C4SpecialGate = 0          ' special gate disabled
ctr1.C4CntSource = 4            ' use counter 4 input as source
ctr1.C4Load = 0                 ' initial load register value is 0
ctr1.C4Buffer = addressOf(dataBuffer4(0))
ctr1.C5CntDir = 1               ' count up
ctr1.C5CntEdge = 1              ' count on positive edge
ctr1.C5CntRepeat = True         ' repeat enabled
ctr1.C5CntType = 0              ' binary counting
ctr1.C5GateCtrl = 0             ' no gating
ctr1.C5OutputCtrl = 3           ' output disabled, high impedance
ctr1.C5Reload = 0               ' reload from load
ctr1.C5SpecialGate = 0          ' special gate disabled
ctr1.C5CntSource = 5            ' use counter 5 input as source
ctr1.C5Load = 0                 ' initial load register value is 0
ctr1.C5Buffer = addressOf(dataBuffer5(0))
        ' program the counters with the previous set parameters
ctr1.SetCounterMode = 1
ctr1.SetCounterMode = 2
ctr1.SetCounterMode = 3
ctr1.SetCounterMode = 4
ctr1.SetCounterMode = 5

End Sub

Sub Form_Unload (Cancel As Integer)
```

```
        disarmAcq
        dbk1.Open = False
End Sub

Function IntToUInt (ByVal i As Integer) As Long
        Dim l As Long
        l = i
        If l  0 Then l = l + 65536
        IntToUInt = l
End Function

Sub ScansToRead_KeyPress (keyascii As Integer)
        Select Case keyascii
                Case &H30 To &H39          'All numbers
                Case &H8                   'Back space
                Case Else
                        keyascii = 0       'Reject all other characters
        End Select
End Sub

Sub Timer1_Timer ()
        Dim cnt As Long
        Dim ctrStr As String
        Dim unprocessed As Long
        Dim i As Integer
        Dim active As Integer
                ' check if the transfer has stopped
        active = ctr1.Active
                ' get the number of scans collected
        cnt = ctr1.Buffered

                ' limit cnt to the number of requested scans
        If cnt  totalScansToRead Then
                cnt = totalScansToRead
        End If
                ' process any new scans
        unprocessed = cnt - scansProcessed
        If unprocessed  0 Then
                ' Check to see if the buffers are overflowed
                If unprocessed  MAXBUF Then
                        disarmAcq
                        MsgBox "Internal buffer overrun"
                        Exit Sub
                End If
                For i = 0 To unprocessed - 1
                        ctrStr = CStr(IntToUInt(dataBuffer1(readIndex))) + Chr$(9)
                        ctrStr = ctrStr + CStr(IntToUInt(dataBuffer2(readIndex))) + Chr$(9)
                        ctrStr = ctrStr + CStr(IntToUInt(dataBuffer3(readIndex))) + Chr$(9)
                        ctrStr = ctrStr + CStr(IntToUInt(dataBuffer4(readIndex))) + Chr$(9)
                        ctrStr = ctrStr + CStr(IntToUInt(dataBuffer5(readIndex)))
                        Print #1, ctrStr
                        readIndex = readIndex + 1
                        If readIndex = MAXBUF Then
                                readIndex = 0
                        End If
                Next i
        End If
                ' update the number of scans processed
        scansProcessed = cnt
        ScansRead.Caption = CStr(cnt)
                ' stop the acquisition if necessary
        If (scansProcessed = totalScansToRead) Or Not active Then
                disarmAcq
        End If
End Sub
```

Notes

Porting Applications A

Overview

This appendix outlines methods for porting applications written to the 16-bit Standard API from the original Daq* Windows 3.1 driver to either a 32-bit version of the Standard API or to the new 32-bit Enhanced API. The Daq* Windows 95/NT driver provides 3 modes of operation:

16-bit standard API (16STD) (Windows 95 Only)	Identical to the Windows 3.1 driver API and documented in chapters 4 and 5. Users that have written programs using the Windows 3.1 driver that want to port their application to Windows 95 in 16-bit mode should use this API. This standard API can also be used by new developers for 16-bit applications. Can process up to 32,767 samples at a time.
32-bit standard API (32STD)	Identical to the Windows 3.1 driver API and documented in chapters 4 and 5. Users that have written programs using the Windows 3.1 driver that want to port their application to Windows 95/NT 32-bit mode should use this API. This API can also be used by new developers for 32-bit applications. Can process up to 2 billion samples at a time.
32-bit enhanced API (32ENH)	Provides enhanced features for applications running under Windows 95 and Windows NT. **This enhanced API (in chapters 2 and 3) is not code-compatible with the standard API (in chapters 4 and 5)**. Legacy applications require modifications to use this API. 32-bit operation only. Can process up to 2 billion samples at a time.

Note: Daq* systems ordered for Windows 95/NT include a Win32 driver capable of native 32-bit mode operation for both Windows NT and Windows 95 systems. Additionally, 16-bit operation (through a thunking layer) is supported under Windows 95.

The enhanced API is device-handle based and allows applications to run in a multi-device/multi-tasked environment. To successfully port existing Windows 3.1 API applications to the new Windows 95/NT enhanced API requires coding changes. In addition to the required device handles, other coding changes may be necessary (refer to the appendix for more information on porting applications). Applications written for this API must use these new API header files. Support files (under the ***Langs\C\32ENH*** and ***Langs\VB\32ENH*** sub-directories of the installation directory) include, respectively:

- For C, the ***DaqX.h*** header file should be used with the ***DaqX.lib*** import library.
- For Visual Basic, the ***DaqX.bas*** file should be used.

Porting Daq Applications Written for Windows 3.1*

The following sections provide information needed to support applications written for the 16-bit standard API under the Windows 3.1 Daq* drivers. Windows 3.1 applications of the driver *may* be binary compatible with the 16-bit Windows 95 version of the driver. *Binary compatible* means that applications already written and compiled for the Windows 3.1 version of the driver may not require re-compilation at all. If it is desirable or necessary to change the application in any way, Visual Basic and C language support has been provided so that an application may be built in either 16-bit or 32-bit modes. Compatibility and porting issues for Visual Basic and C are described in the following sections. (Since previous driver versions did not support Delphi, there is no need to review Delphi porting issues here.)

Windows 3.1 Binary Compatibility (16-bit)

This section refers to Windows 95 Only.

In some cases, it may be possible to run Windows applications written for the Windows 3.1 version of the driver without re-compiling the application. It does not matter which language the application was written in, as long as the application is a Windows application written to use ***DaqBook.dll***. To do this, the 16-bit Windows 3.1 version of the driver (***DaqBook.dll***) needs to be replaced by the 16-bit Windows 95 binary-compatible version of the driver. To attempt this, perform the following steps:

1. Go to the ***Windows\System*** directory.
2. Copy the ***DaqBook.dll*** to a backup file outside of ***Windows\System***.
3. Delete ***DaqBook.dll*** and make sure that there are no others in your path.
4. Copy **<installation path>*Utils\DaqBookX.dll*** to ***Windows\System\DaqBook.dll***.
5. Run the application.

The application should run as before. If there are problems, it may be necessary to recompile the application as described below.

To switch back to the Windows 3.1 version of the driver:

1. Go to the ***Windows\System*** directory.
2. Delete ***DaqBook.dll*** from the ***Windows\System*** directory.
3. Copy the original ***DaqBook.dll*** to the ***Windows\System*** directory. Until this is done, the Windows 95 version of the driver will be used by the application.

Note: Problems are likely to occur if both *DaqBookX.dll* and *DaqBook.dll* are in the *Windows\System* directory at the same time.

Unsupported Windows 3.1 API Functions

The following functions have become obsolete for the 16-bit and 32-bit standard API release. The functions are present; however, they perform no action.

daqAdcRdFore	Use the **daqAdcRd** API function to acquire one sample from a selected channel at a selected gain. This function always returns "**DerrNotCapable**".
daqCtrRdNFore	
daqCtrRdNBack	
daqCtrGetBackStat	
daqSetProtocol	The protocol must be set through the Daq* Configuration utility. This function always returns "**DerrNoError**".

Porting Visual Basic Programs

16-bit Mode

This section refers to Windows 95 Only.

To convert existing Visual Basic applications requires very little effort. Perform the following steps, and then run or re-compile your application. The new Daq Windows 95 Visual Basic header file, ***DaqX16.bas***, is compatible with Visual Basic versions 2.0 through 4.0 (16-bit).

1. Remove the ***DaqBook.bas*** file from the project, and add the ***DaqX16.bas*** file (that resides in the **<installation path>*Langs\vb\16std*** directory).
2. Remove or replace obsolete function calls (see *Unsupported Windows 3.1 API Functions*).

32-bit Mode

To convert existing Visual Basic applications requires more work. The majority of changes involve converting integer sample or scan counts to long. Perform the following steps, and then run or re-compile your application using the 32-bit version of Visual Basic 4.0.

1. Remove the ***DaqBook.bas*** file from the project, and add the ***DaqComp.bas*** file (that resides in the **<installation path>*Langs\vb\32std*** directory).
2. Remove or replace all obsolete function calls (see *Unsupported Windows 3.1 API Functions*).
3. Change function parameters as specified in the following table:

Function Prototype	Previous Parameter Definition	Change Parameter Definition To ...
VBdaq200SetScan	count%	count&
VBdaqAdcRdN	count%	count&
VBdaqAdcRdScanN	count%	count&
VBdaqAdcRdNFore	count%	count&
VBdaqAdcRdNForePreT	count% retCount%	count& retCount&
VBdaqAdcRdNForePreTWait	count% retCount%	count& retCount&
VBdaqAdcRdNBack	count%	count&
VBdaqAdcRdNBackPreT	count%	count&
VBdaqAdcConvertTagged	count%	count&
VBdaqAdcSetScan	count%	count&
VBdaqAdcSetTrigPreT	preCount% postCount%	preCount& postCount&
VBdaqBrdDacPredefWave	samples%	samples&
VBdaqBrdDacUserWave	samples%	samples&
VBdaqBrdDacWriteFIFO	samples%	samples&
VBdaqCalConvert	scans%	scans&
VBdaqCalSetupConvert	scans%	scans&
VBdaqCtrRdNFore	count%	count&
VBdaqCtrRdNBack	count%	count&
VBdaqLinearConvert	scans% nValues%	scans& nValues&
VBdaqLinearSetupConvert	scans% nValues%	scans& nValues&
VBdaqRtdConvert	scans% ntemp%	scans& ntemp&
VBdaqRtdSetupConvert	scans% ntemp%	scans& ntemp&
VBdaqTCConvert	scans% ntemp%	scans& ntemp&
VBdaqTCSetupConvert	scans% ntemp%	scans& ntemp&
VBdaqZeroConvert	scans%	scans&
VBdaqZeroSetupConvert	scans%	scans&

Porting C Programs

16-bit Mode

This section refers to Windows 95 Only.

To convert existing C applications requires very little effort. Perform the following steps and then re-compile your application with a 16-bit C compiler. The ***DaqX16.h*** and ***DaqX16.lib*** files reside in the **<installation path>*Langs**C**16std*** directory.

1. Replace all `#include "DaqBook.h"` lines with `#include "DaqX16.h"`.
2. Replace `DaqBook.lib` in your project file or makefile with `DaqX16.lib`.
3. Remove or replace obsolete function calls (see *Unsupported Windows 3.1 API Functions*).

32-bit Mode

To convert existing C applications requires more work. The majority of changes involve converting integer data buffers to short data buffers. Integers are 16 bits in 16-bit C compilers, but are 32 bits in 32-bit C compilers. Short integers are 16 bits for both. Perform the following steps and then re-compile your application with a 32-bit C compiler. The ***DaqComp.h*** and ***Daqcomp.lib*** files reside in the **<installation path>*Langs**C**32std*** directory.

1. Replace all `#include "DaqBook.h"` lines with `#include "DaqComp.h".`
2. Replace `DaqBook.lib` in your project file or makefile with `DaqComp.lib`.
3. Remove or replace obsolete function calls (see *Unsupported Windows 3.1 API Functions*).
4. Change function parameters as specified in the following table:

Function Prototype	Previous Parameter Definition	Change Parameter Definition To ...
daqAdcConvertTagged	unsigned int *taggedData unsigned int *buf	unsigned short *taggedData unsigned short *buf
daqAdcRd	unsigned int *sample	unsigned short *sample
daqAdcRdFore	unsigned int *sample	unsigned short *sample
daqAdcRdN	unsigned int *buf	unsigned short *buf
daqAdcRdNBack	unsigned int *buf	unsigned short *buf
daqAdcRdNBackPreT	unsigned int *buf	unsigned short *buf
daqAdcRdNFore	unsigned int *buf	unsigned short *buf
daqAdcRdNForePreT	unsigned int *buf	unsigned short *buf
daqAdcRdNForePreTWait	unsigned int *buf	unsigned short *buf
daqAdcRdScan	unsigned int *buf	unsigned short *buf
daqAdcRdScanN	unsigned int *buf	unsigned short *buf
daqAdcStopBack_LV	unsigned int *bufP	unsigned short *bufP
daqBrdDacUserWave	unsigned int *buf	unsigned short *buf
daqBrdDacWriteFIFO	unsigned int *storage	unsigned short *storage
daqCalConvert	unsigned int *counts	unsigned short *counts
daqCalSetupConvert	unsigned int *counts	unsigned short *counts
daqCtrGetHold	unsigned int *ctrVal	unsigned short *ctrVal
daqCtrRdFreq	unsigned int *count	unsigned short *count
daqCtrRdNBack	unsigned int *ctr1Buf unsigned int *ctr2Buf unsigned int *ctr3Buf unsigned int *ctr4Buf unsigned int *ctr5Buf	unsigned short *ctr1Buf unsigned short *ctr2Buf unsigned short *ctr3Buf unsigned short *ctr4Buf unsigned short *ctr5Buf
daqCtrRdNFore	unsigned int *ctr1Buf unsigned int *ctr2Buf unsigned int *ctr3Buf unsigned int *ctr4Buf unsigned int *ctr5Buf	unsigned short *ctr1Buf unsigned short *ctr2Buf unsigned short *ctr3Buf unsigned short *ctr4Buf unsigned short *ctr5Buf
daqDacWtMany	unsigned int *dataVals	unsigned short *dataVals
daqDbkSetChanOption	double optionValue	float optionValue
daqLinearConvert	unsigned *counts	unsigned short *counts
daqLinearSetupConvert	unsigned *counts	unsigned short *counts
daqRtdConvert	unsigned *counts int *temp	unsigned short *counts short *temp
daqRtdSetupConvert	unsigned *counts int *temp	unsigned short *counts short *temp
daqTCConvert	unsigned *counts int *temp	unsigned short *counts short *temp
daqTCSetupConvert	unsigned *counts int *temp	unsigned short *counts short *temp
daqZeroConvert	unsigned int *counts	unsigned short *counts
daqZeroSetupConvert	unsigned int *counts	unsigned short *counts

WARRANTY/DISCLAIMER

OMEGA ENGINEERING, INC. warrants this unit to be free of defects in materials and workmanship for a period of **13 months** from date of purchase. OMEGA Warranty adds an additional one (1) month grace period to the normal **one (1) year product warranty** to cover handling and shipping time. This ensures that OMEGA's customers receive maximum coverage on each product.

If the unit should malfunction, it must be returned to the factory for evaluation. OMEGA's Customer Service Department will issue an Authorized Return (AR) number immediately upon phone or written request. Upon examination by OMEGA, if the unit is found to be defective it will be repaired or replaced at no charge. OMEGA's WARRANTY does not apply to defects resulting from any action of the purchaser, including but not limited to mishandling, improper interfacing, operation outside of design limits, improper repair, or unauthorized modification. This WARRANTY is VOID if the unit shows evidence of having been tampered with or shows evidence of being damaged as a result of excessive corrosion; or current, heat, moisture or vibration; improper specification; misapplication; misuse or other operating conditions outside of OMEGA's control. Components which wear are not warranted, including but not limited to contact points, fuses, and triacs.

OMEGA is pleased to offer suggestions on the use of its various products. However, OMEGA neither assumes responsibility for any omissions or errors nor assumes liability for any damages that result from the use of its products in accordance with information provided by OMEGA, either verbal or written. OMEGA warrants only that the parts manufactured by it will be as specified and free of defects. OMEGA MAKES NO OTHER WARRANTIES OR REPRESENTATIONS OF ANY KIND WHATSOEVER, EXPRESSED OR IMPLIED, EXCEPT THAT OF TITLE, AND ALL IMPLIED WARRANTIES INCLUDING ANY WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE HEREBY DISCLAIMED. LIMITATION OF LIABILITY: The remedies of purchaser set forth herein are exclusive and the total liability of OMEGA with respect to this order, whether based on contract, warranty, negligence, indemnification, strict liability or otherwise, shall not exceed the purchase price of the component upon which liability is based. In no event shall OMEGA be liable for consequential, incidental or special damages.

CONDITIONS: Equipment sold by OMEGA is not intended to be used, nor shall it be used: (1) as a "Basic Component" under 10 CFR 21 (NRC), used in or with any nuclear installation or activity; or (2) in medical applications or used on humans. Should any Product(s) be used in or with any nuclear installation or activity, medical application, used on humans, or misused in any way, OMEGA assumes no responsibility as set forth in our basic WARRANTY/DISCLAIMER language, and additionally, purchaser will indemnify OMEGA and hold OMEGA harmless from any liability or damage whatsoever arising out of the use of the Product(s) in such a manner.

RETURN REQUESTS/INQUIRIES

Direct all warranty and repair requests/inquiries to the OMEGA Customer Service Department. BEFORE RETURNING ANY PRODUCT(S) TO OMEGA, PURCHASER MUST OBTAIN AN AUTHORIZED RETURN (AR) NUMBER FROM OMEGA'S CUSTOMER SERVICE DEPARTMENT (IN ORDER TO AVOID PROCESSING DELAYS). The assigned AR number should then be marked on the outside of the return package and on any correspondence.

The purchaser is responsible for shipping charges, freight, insurance and proper packaging to prevent breakage in transit.

FOR **WARRANTY** RETURNS, please have the following information available BEFORE contacting OMEGA:

1. P.O. number under which the product was PURCHASED,
2. Model and serial number of the product under warranty, and
3. Repair instructions and/or specific problems relative to the product.

FOR **NON-WARRANTY** REPAIRS, consult OMEGA for current repair charges. Have the following information available BEFORE contacting OMEGA:

1. P.O. number to cover the COST of the repair,
2. Model and serial number of the product, and
3. Repair instructions and/or specific problems relative to the product.

OMEGA's policy is to make running changes, not model changes, whenever an improvement is possible. This affords our customers the latest in technology and engineering.

OMEGA is a registered trademark of OMEGA ENGINEERING, INC.

© Copyright 1996 OMEGA ENGINEERING, INC. All rights reserved. This document may not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without prior written consent of OMEGA ENGINEERING, INC.

Where Do I Find Everything I Need for Process Measurement and Control? OMEGA...Of Course!

TEMPERATURE

- ☑ Thermocouple, RTD & Thermistor Probes, Connectors, Panels & Assemblies
- ☑ Wire: Thermocouple, RTD & Thermistor
- ☑ Calibrators & Ice Point References
- ☑ Recorders, Controllers & Process Monitors
- ☑ Infrared Pyrometers

PRESSURE, STRAIN AND FORCE

- ☑ Transducers & Strain Gauges
- ☑ Load Cells & Pressure Gauges
- ☑ Displacement Transducers
- ☑ Instrumentation & Accessories

FLOW/LEVEL

- ☑ Rotameters, Gas Mass Flowmeters & Flow Computers
- ☑ Air Velocity Indicators
- ☑ Turbine/Paddlewheel Systems
- ☑ Totalizers & Batch Controllers

pH/CONDUCTIVITY

- ☑ pH Electrodes, Testers & Accessories
- ☑ Benchtop/Laboratory Meters
- ☑ Controllers, Calibrators, Simulators & Pumps
- ☑ Industrial pH & Conductivity Equipment

DATA ACQUISITION

- ☑ Data Acquisition & Engineering Software
- ☑ Communications-Based Acquisition Systems
- ☑ Plug-in Cards for Apple, IBM & Compatibles
- ☑ Datalogging Systems
- ☑ Recorders, Printers & Plotters

HEATERS

- ☑ Heating Cable
- ☑ Cartridge & Strip Heaters
- ☑ Immersion & Band Heaters
- ☑ Flexible Heaters
- ☑ Laboratory Heaters

ENVIRONMENTAL MONITORING AND CONTROL

- ☑ Metering & Control Instrumentation
- ☑ Refractometers
- ☑ Pumps & Tubing
- ☑ Air, Soil & Water Monitors
- ☑ Industrial Water & Wastewater Treatment
- ☑ pH, Conductivity & Dissolved Oxygen Instruments