

User's Guide



<http://www.omega.com>
[e-mail: info@omega.com](mailto:info@omega.com)

OMB-TEMPBOOK/66
Thermocouple & Voltage
Data Acquisition System



OMEGAnetSM On-Line Service
<http://www.omega.com>

Internet e-mail
info@omega.com

Servicing North America:

USA: One Omega Drive, Box 4047
ISO 9001 Certified 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-OMEGASM
Customer Service: 1-800-622-2378 / 1-800-622-BESTSM
Engineering Service: 1-800-872-9436 / 1-800-USA-WHENSM
TELEX: 996404 EASYLINK: 62968934 CABLE: OMEGA

Mexico and Latin America: Tel: (95) 800-TC-OMEGASM 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: 25 Swannington Road, P.O. Box 7, Omega Drive,
ISO 9002 Certified Broughton Astley, Leicestershire, Irlam, Manchester,
LE9 6TU, England M44 5EX, England
Tel: 44 (1455) 285520 Tel: 44 (161) 777-6611
FAX: 44 (1455) 283912 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 Manual

This manual explains the setup and operation of the TempBook data acquisition system. This manual is divided into a table of contents, 11 chapters, and 1 appendix as follows:

Chapter 1 - **Introduction and Quick Start** begins with an overview description and a listing of system specifications. If you are generally familiar with this type of equipment, the Quick Start section shows how to hook up a simple system; however, most users will prefer the more detailed startup procedures in chapter 2.

Chapter 2 - **Installation, Configuration, and Calibration** describes panel switches, indicators, connectors, hardware hookups, software installation, configuration, and calibration.

Chapter 3 - **Using TempView (16-bit)** explains the use and features of TempView (including PostView). Screen prints show you the pull-down menus, toolbars, charts, and parameter fields discussed in the text.

Chapter 4 - **Using DaqView (32-bit)** explains the use and features of DaqView (including PostView and DaqViewXL). Screen prints show you the pull-down menus, toolbars, charts, and parameter fields discussed in the text. A guided tour gives you hands-on experience with simple examples.

Chapter 5 - **Programmer's Guide** explains how to custom-program for your application. Various concerns are discussed; e.g., a comparison of standard and enhanced APIs and language support.

Chapter 6 - **Standard API Programming of the TempBook With C** describes several example programs using the standard API with the C language.

Chapter 7 - **Software Calibration and Zero Compensation** describes the commands and parameters related to calibration and zero compensation. This chapter organizes and supplements related sections of the *tbkCommand Reference* chapter.

Chapter 8 - **Thermocouple Measurement** describes the commands and parameters related to thermocouple measurement. This chapter organizes and supplements related sections of the *tbkCommand Reference* chapter.

Chapter 9 - **tbkCommand Reference (Standard API)** describes the commands and parameters of the “standard” API including useful reference tables.

Chapter 10 - **Programming Models for Enhanced API** describes the fundamental building blocks for TempBook 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.

Chapter 11 - **daqCommand Reference (Enhanced API)** describes the commands and parameters of the “enhanced” API including useful reference tables.

Appendix A- Differential Measurement Configurations describes setups for “floating” and “referenced” differential signal input connections.

CAUTION



Using this equipment in ways other than described in this manual can cause personal injury or equipment damage. Before setting up and using your equipment, you should read *all* documentation that covers your system. Pay special attention to cautions and warnings formatted like this one.



Software Reference Note:

16-bit PC users can use TempView out-of-the-box (see chapter 3) or program their own application. Programmers, see chapters 5 and 6-9; chapters 6 to 8 show examples and explain how to perform common tasks with the standard API detailed in chapter 9. **32-bit PC users** can use DaqView out-of-the-box (see chapter 4) or program their own application. Programmers, see chapters 5 and 10-11; chapter 10 explains programming models for the enhanced API detailed in chapter 11.

Table of Contents

1 Introduction and Quick Start

Overview	1-1
Description	1-1
Available Accessories	1-2
Specifications	1-2
Quick Start	1-3
Signal Connection	1-3
PC Connection	1-3
Power Connection and Switch	1-4
Software Installation	1-4
Using DaqView	1-4

2 Installation, Configuration, and Calibration

Inspection	2-1
Panel Connectors and Indicators	2-1
Termination Card and I/O Connectors	2-2
Internal Configuration	2-3
Watchdog Timer Enable/Disable (JP8)	2-3
Time Base Selection (JP9)	2-3
Hardware Installation	2-3
Rechargeable Battery Module (DBK30A)	2-4
Charging the Battery Modules	2-4
Battery Module Connection	2-5
Parallel Port Capabilities	2-5
TempBook Software Installation	2-6
Making a Backup Copy	2-6
DOS Installation	2-6
Installation Under Windows 3.1	2-6
Running TempTest	2-7
Installation Under Windows 95 and Windows NT	2-7
TempBook Configuration Under Windows 95/NT	2-8
Connection Troubleshooting (Windows 95/NT)	2-10
Calibration of TempBook	2-11
Calibration Constants File Installation	2-11
Hardware Calibration	2-11

3 Using TempView (16-bit)

Application Startup	3-1
TempView Components	3-2
Analog Input Spreadsheet	3-2
Acquisition Configuration	3-3
Counter/Timer Window	3-4
Digital I/O Window	3-4
Charts and the Spreadsheet's "Reading" column	3-4
TempView Menu Items	3-5
File	3-5
Edit	3-5
Select Device	3-6
Window	3-6
Acquisition	3-6
Charts	3-6
Using PostView	3-6
The PostView Timebase	3-8
PostView Menu Items	3-8

4 Using DaqView (32-bit)

Overview	4-1
DaqView's Main Window	4-1
Starting DaqView	4-2
Beginner's Tour of DaqView	4-2
Analog Input Spreadsheet	4-3
Acquisition Configuration	4-4
Pull-Down Menu Items	4-4
File	4-4
Edit	4-5
Acquire	4-5
Window	4-5

Device	4-6
The Configure System Hardware Window	4-6
Help	4-7
Toolbar Items	4-7
Make All Channels Active (or Inactive)	4-7
Charts	4-7
Bar Graph Meters	4-8
Analog Meters	4-8
Digital Meters	4-8
Properties of Meter Windows	4-9
Start (or Stop) All Indicators	4-9
Analog Output Window	4-9
Digital I/O Window	4-10
Counter/Timer Window	4-10
Arbitrary Waveform Window	4-11
Enable Input Reading Column	4-12
Go	4-12
PostView 1.5	4-13
PostView Menu Items	4-14
PostView Time Base	4-15
Data File Accessibility	4-15
DaqViewXL	4-16
Program Requirements	4-16
Installation of DaqViewXL Software	4-16
Running DaqViewXL for the First Time	4-16
Removing DaqViewXL from Excel	4-18
Basic Function of DaqViewXL	4-18
Configuring an Acquisition	4-19
Real-Time Charting	4-19
Data Header	4-20
Analog Output	4-20
Digital I/O and Counter-Timers	4-21
Hardware Support	4-21
Hints and Tips for DaqViewXL	4-21

5 Programmer's Guide

A Programmer's View of TempBook Operations	5-1
Driver Options	5-3
Standard API (tbk...)	5-3
Enhanced API (daq...)	5-3
Language Support	5-3

6 Standard API Programming of the TempBook With C

Accessing TempBook from a Windows Program	6-1
Accessing TempBook from a C for Windows Program	6-1
High-Level Analog Input	6-1
Low-Level Analog Input	6-2
Analog Input in the Background	6-2
General Purpose Digital I/O Functions	6-4
High-Speed Digital Input	6-4
Counter/Timer Functions	6-5
High-Level Thermocouple Data Acquisition	6-6
Thermocouple Linearization	6-7
Sample Programs	6-8
High-Level Analog Input	6-8
Low-Level Analog Input	6-9
Analog Input in the Background	6-10
General Purpose Digital I/O	6-11
High-Speed Digital Input	6-12
Counter/Timer Functions	6-13
High-Level Thermocouple Measurement	6-14
Low-Level Thermocouple Linearization	6-15
Command Summary, C Language (Windows)	6-17

7 Software Calibration and Zero Compensation

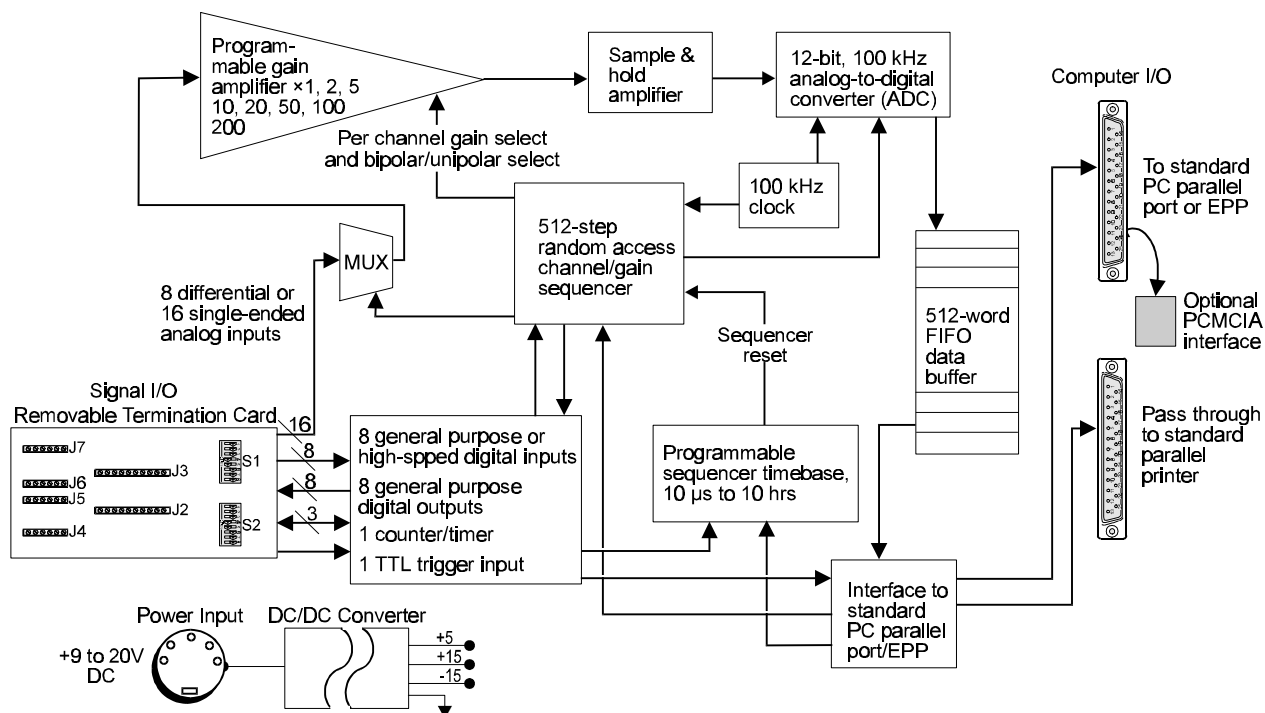
Software Calibration	7-1
Initializing the Calibration Constants	7-1
Calibration Setup and Conversion	7-2

Calibration Example-----	7-2
Zero Compensation-----	7-3
Zero Compensation Example-----	7-4
Automatic Zero Compensation-----	7-5
8 Thermocouple Measurement	
Low-Level Thermocouple Data Conversion Functions-----	8-1
High-Level Thermocouple Measurement Functions-----	8-3
Single-Channel Measurement (tbkRdTemp)-----	8-3
Multiple Measurements from a Single Channel (tbkRdTempN)-----	8-4
Multiple Channel Measurement (tbkRdTempScan)-----	8-4
Multiple Measurements from Multiple Channels (tbkRdTempScanN)-----	8-4
9 tbkCommand Reference (Standard API)	
Overview-----	9-1
Commands in Alphabetical Order-----	9-2
API Reference Tables-----	9-32
A/D Channel Descriptions-----	9-32
A/D Gain Definitions-----	9-32
A/D Trigger Source Definitions-----	9-32
Pretrigger Functions Trigger Source Definitions-----	9-33
Thermocouple Types-----	9-33
API Error Codes - C Languages-----	9-33
API Error Codes - QuickBASIC-----	9-34
API Error Codes - Turbo Pascal-----	9-35
API Error Codes - Visual Basic-----	9-36
10 Enhanced API Programming Models (TempBook)	
Overview-----	10-1
Data Acquisition Environment-----	10-1
Application Programming Interface (API)-----	10-1
Enhanced vs Standard API-----	10-1
Hardware Capabilities and Constraints-----	10-1
Signal Environment-----	10-2
Basic Models-----	10-2
Initialization and Error Handling-----	10-3
Foreground Acquisition with One-Step Commands-----	10-5
Temperature Acquisition Using One-Step Commands-----	10-7
Counted Acquisition Using Linear Buffers-----	10-9
Indefinite Acquisition, Direct-To-Disk Using Circular Buffers-----	10-11
Multiple Hardware Scans, Software Triggering-----	10-14
Background Acquisition-----	10-16
Temperature Acquisition Using TC Conversion Functions-----	10-18
Double Buffering-----	10-21
Direct-To-Disk Transfers-----	10-23
Transfers With Driver-Allocated Buffers-----	10-26
Summary Guide of Selected Enhanced API Functions-----	10-28
11 daqCommand Reference (Enhanced API)	
Overview-----	11-1
Commands in Alphabetical Order-----	11-2
API Reference Tables-----	11-35
Daq Device Property Definitions-----	11-36
Event-Handling Definitions-----	11-36
Hardware Information Definitions-----	11-36
ADC Trigger Source Definitions-----	11-37
ADC Miscellaneous Definitions-----	11-37
TempBook Definitions-----	11-38
General I/O Definitions-----	11-38
DaqTest Command Definitions-----	11-38
Calibration Input Signal Sources-----	11-38
API Error Codes-----	11-39
Appendix - Differential Measurement Configurations	
Floating Differential-----	A-1
Referenced Differential-----	A-1

Overview

This chapter describes the TempBook in general terms including specifications. Also, a Quick Start can help you get a simple system up and running (chapter 2 has more detailed installation instructions).

Description



TempBook/66 Block Diagram

The TempBook/66 adds voltage and thermocouple measurement capability to notebook PCs for portable test applications. The TempBook also provides an effective alternative to plug-in boards for desktop PCs. The TempBook provides 12-bit, 100 kHz data acquisition and can support up to 800 Kbyte/s data transfer to a PC via an enhanced parallel port interface or PCMCIA link. The unit can also connect to a standard parallel port and transfer readings directly to disk at up to 100K readings/s. As an external module with the same footprint as a typical notebook PC, the TempBook can be attached directly under a notebook PC for portability.

The TempBook has a built-in analog capability that permits it to measure 8 channels in a differential mode or 16 channels in single-ended mode. Its on-board programmable-gain instrumentation amplifier can be set to gains of $\times 1, 2, 5, 10, 20, 50, 100$ or 200 on a per channel basis. Its A/D converter scans selected channels at a constant $10 \mu\text{s}/\text{channel}$ rate, minimizing the time skew between consecutive channels. The time between the start of each scan sequence can be programmed to start immediately or at intervals of up to 10 hours. The TempBook features a hardware-based digital/TTL trigger that minimizes trigger latency to less than $10 \mu\text{s}$. The TempBook can also be triggered from a command from the PC.

The TempBook/66 can be powered by an included AC adapter, a standard 12V car battery, any +9 to 20 VDC source, or an optional rechargeable nickel-cadmium battery module (DBK30A). This makes it ideal for field and remote data acquisition applications.

Software included with the TempBook includes: TempView (16-bit) and DaqView (32-bit) Windows-based programs that allow you to set up your acquisition and save acquired data directly to disk. The package also includes thermocouple linearization for direct readout of temperatures. PostView is a Windows-based post application waveform display application that permits you to display previously acquired data.

In addition, the TempBook/66 includes DOS drivers for Quick Basic, C, and Pascal; Windows drivers for Visual Basic and C for Windows. Several graphically analysis and control software packages also support the TempBook/66. These include DASyLab, Snap-Master, LABTECH NOTEBOOK and LabVIEW.

Available Accessories

Additional accessories that can be ordered for the TempBook/66 include:

- DBK35 PCMCIA interface card and cable
- DBK30A Rechargeable battery module

Specifications

General

Connector: Connects to a PC via an included parallel port cable; user signals are connected via screw terminals on a removable screw-terminal card.

Thermocouple Types: J, K, S, T, E, B, R & N

Input Ranges:

Unipolar: .05, 0.1, 0.2, 0.5, 1, 2, 5 V

Bipolar: ± 0.25 , .05, 0.1, 0.25, 0.5, 1, 2.5, 5 V

Analog Inputs: 8 differential or 16 single-ended voltage or 8 differential thermocouple inputs

Analog-to-Digital Converter: 12-bit with no missing codes, unsigned binary output format.

Cold Junction Sensor Output: 100 mV/°C

Input Impedance: 100K/100M Ohm, Switch-selectable on a per-channel basis in parallel with 100 pF

Input RC Filter -3dB Frequency: 15.9 kHz switch-selectable on a per channel basis

Gain Accuracy: 0.1%

Maximum Input Voltage: ± 15 V

CMRR (Input Stage): 90 dB typ, DC to 60 Hz

Offset: Software-compensated

Offset Drift: Software-compensated

Channel Sequencer

Depth: 512 locations

Speed: 10 μ s per channel, fixed

Interval Between Scans: 10 μ s to 10 hours, software-programmable

Gains: Sequencer programmable on a per-channel basis.

Unipolar/Bipolar: Sequencer programmable on a per channel basis.

Single-Ended/Differential: Software programmable for all channels

Temperature

Thermocouple Accuracy (TempBook/66 @ 0 to 50°C)

Type	Min (°C)	Max (°C)	Accuracy (<°C)	(>°C)
J	-200	760	0.9	0.9
K	-200	1260	2.4	1.5
T	-200	400	2.1	1.2
E	-270	1000	2.1	1.3
N28	-270	400	1.2	1.2
N14	-	1300	-	1.5
S	-	1780	-	2.4
R	-	1780	-	2.4
B	-	1820	-	2.7

Digital I/O

Inputs

Number: 8 fixed as inputs

Usage: General-purpose register addressable or high-speed scanned via channel sequencer

Type: TTL-compatible

High Voltage: 2.0 V min

Low Voltage: 0.8 V max

Outputs

Number: 8 fixed as outputs

Usage: General-purpose register addressable

Type: TTL-compatible

High Voltage: 3.0 V @ 2.5 mA source

Low Voltage: 0.4V @ 2.5 mA sink

Trigger Input

Type: TTL-compatible

High Voltage: 2.0 V min

Low Voltage: 0.8 V max

Counter/Timer Port

Device: 8254 (P0 only)

Number: 1 gate input, 1 clock input & 1 output

Clock: Internal 100 kHz or external up to 10 MHz

Usage: Register addressable

Type: TTL compatible

Input High Voltage: 2.0 V min

Input Low Voltage: 0.8 V max

Output High Voltage: 3.0 V @ 2.5 mA source

Output Low Voltage: 0.4 V @ 2.5 mA sink

Thermocouple Resolution (°C)

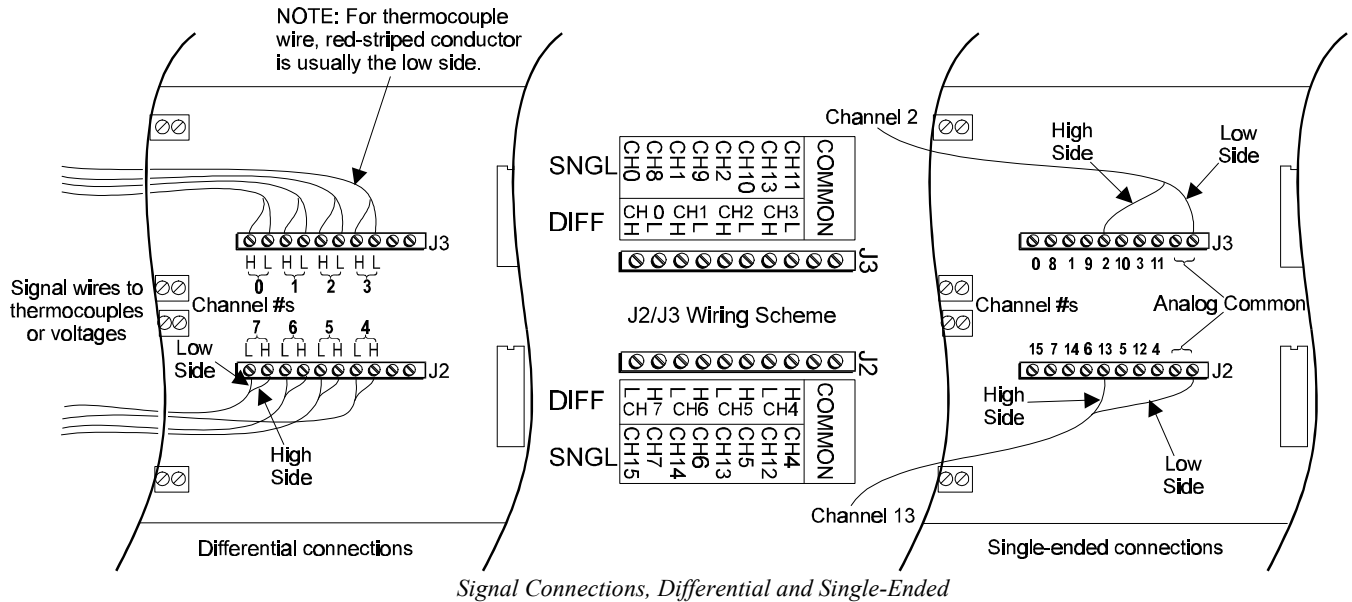
Type	<0°C	>0°C
J	1.2	0.5
K	1.1	0.8
T	0.8	0.3
E	1.9	0.9
N28	0.9	0.9
N14	-	5.0
S	-	1.6
R	-	1.5
B	-	1.8

Quick Start

For those users who wish to get their TempBook up and running as quickly as possible, this section provides a brief explanation of the steps required. **Note:** unless already familiar with this type of system, most users will need to read chapter 2, *Installation and Configuration*.

Signal Connection

The thermocouple and voltage input signals are fed to the TempBook through a removable termination card. To remove this card, rotate the ejector handle with your thumb or finger. Once removed, the thermocouple or voltage connections should be made as shown in the figure.



Note: When connecting thermocouple or other low-level signals in addition to high-level signals, connect the low-level signals to the lowest numbered channels with connections in ascending order of signal magnitude.

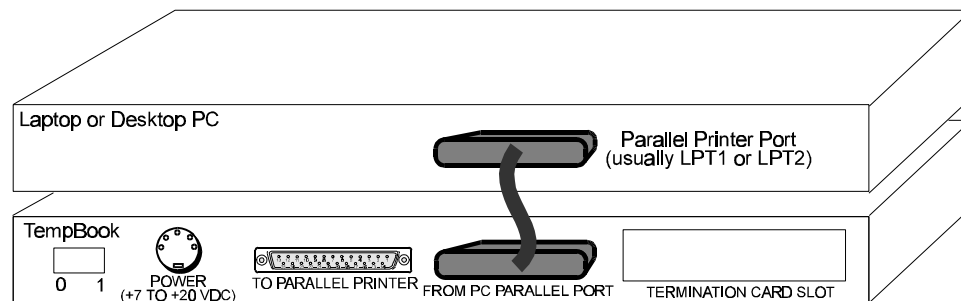
The DIP switches located on the termination card connect optional biasing resistors as well as input filters.

- For thermocouple or other differential inputs, these switches should be closed to provide the required bias current path.
- For single-ended inputs these switches can be optionally opened or closed.

For further details, refer to section *Termination Card and I/O Connectors* in chapter 2.

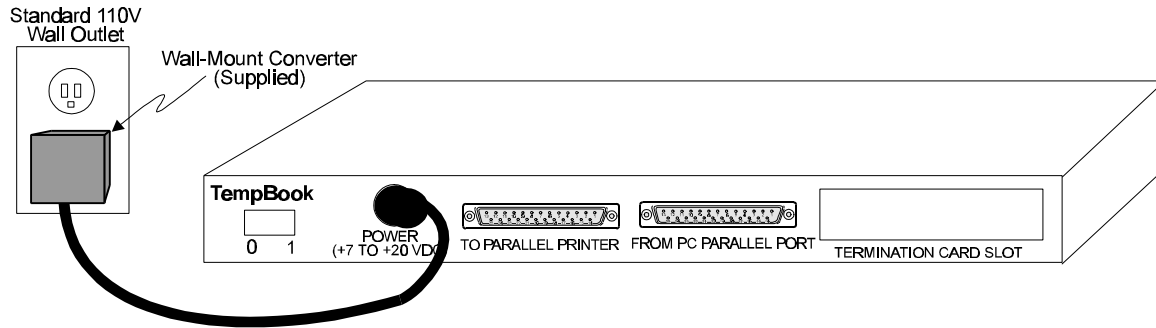
PC Connection

The TempBook communicates with a laptop or desktop computer through the parallel printer port. Connect the supplied cable to the computer's parallel port and the TempBook's parallel port.



Power Connection

The TempBook is typically powered from the supplied wall-mount converter or from an optional DBK30A battery module. The wall-mount converter plugs into any standard 110 VAC wall outlet and its other end plugs into the circular DIN5 receptacle on the TempBook's rear panel. If using the battery module, please refer to the section *Rechargeable Battery Module* in chapter 2.



TempBook Power Connection

Software Installation

The TempBook/66 comes with a Microsoft Windows application (16-bit TempView or 32-bit DaqView) that provides the easiest means to collect data from the TempBook. To install the software (after backing up the supplied disks), insert disk 1 for either 16-bit or 32-bit PCs into your floppy disk drive, usually A or B. Choose RUN from the Windows Program Manager FILE menu (or Start/Run in Win95/NT) and type in **A(or B) : \SETUP.EXE**. Follow the on-screen prompts. If you do not plan on writing programs to control the TempBook, you should only select the options for Temp/DaqView, DaqTest, and TempCal and not the options for the language-support drivers. For more details, see *Software Installation* in chapter 2.

Using TempView or DaqView

Note: TempView and DaqView perform the same basic functions, but DaqView has more power and convenience features.

Once the setup program has installed the software, a TempBook program group will appear within Windows. To start Temp/DaqView, click on its program icon. The main window will soon display giving you a spreadsheet of channels in rows and channel parameters in columns.

To begin acquiring data, you should first select the signal reference for all channels by clicking the appropriate selection within the Signal Reference box. If you have connected thermocouples or differential voltage signals, then select differential, otherwise select single-ended.

Turn on only those channels which have signals connected by selecting YES or NO within the left-most column of the Analog Input spreadsheet. Then select the appropriate thermocouple type or voltage gain setting for each channel within the Type column. The spreadsheet entries for all channels can be changed by clicking once on the column label at the top of the spreadsheet to highlight the column and then making the entry within the analog input box. Next choose the desired polarity, unipolar or bipolar, for each channel in the Polarity column. Finally, choose the desired units for each channel within the Units column.

Now you are ready to read data from the TempBook. To immediately see the signal values present select Enable Input Reading Column from within the Acquisition menu. If thermocouple or low frequency voltage signals are being read, then more accurate and stable readings can be achieved by selecting Averaging from the Signal Conditioning box. When selected, the values displayed will be the average of 100 readings.

The channel readings can also be viewed in a moving graph called Charts or for post-acquisition in PostView. For more details on these topics and use of the digital I/O and counter/timer, see the appropriate chapter in this manual, chapter 3 *Using TempView (16-bit)* or chapter 4 *Using DaqView (32-bit)*.

Inspection

The TempBook components were carefully inspected prior to shipment. When you receive your data acquisition system, carefully unpack all items from the shipping carton and check for any obvious signs of physical damage which may have occurred during shipment. Immediately report any damage to the shipping agent. Retain all shipping materials in case you must return the unit to the factory.

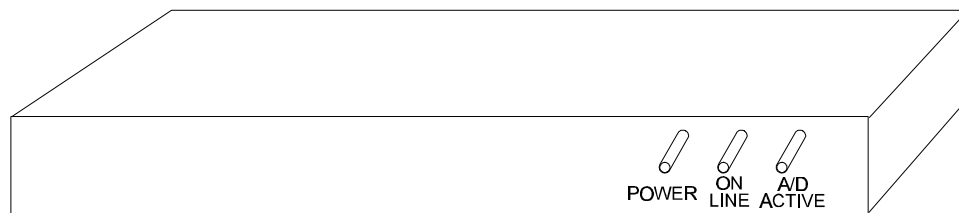
Every TempBook is shipped with the following items:

- TempBook Data Acquisition System
- User's Manual
- Software Disk (2) in IBM Format including DaqView, DOS Drivers and Windows Drivers
- Accessory Kit, including: CA-35-2 (2 ft parallel port cable) and an AC Adapter

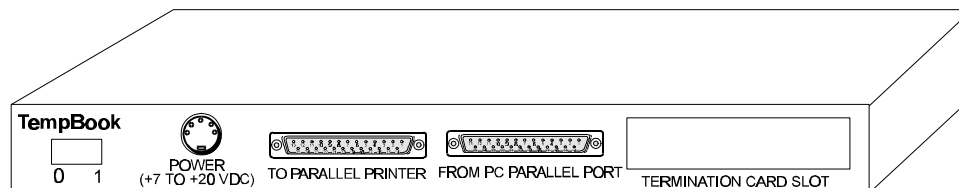
Panel Connectors and Indicators

The TempBook front panel consists of 3 status-indicator LEDs. The rear panel consists of the power switch, power input, two DB25s for parallel port connection and pass-thru, and a slot to accept the input termination card. The function of each of these components is described below.

TempBook Panel Connectors and Indicators	
POWER SWITCH	This rocker arm switch turns on the DC power to the TempBook when the "1" side of the switch is depressed.
POWER INPUT	This input connector accepts +7 VDC to +20 VDC.
TO PARALLEL PRINTER	This parallel port can connect to any standard parallel printer. This allows the user to attach both the TempBook and a parallel printer to the system simultaneously.
FROM PC PARALLEL PORT	This parallel port connects directly to the PC's parallel printer port. This allows the host system to communicate with the TempBook.
POWER	This LED is ON when power is applied to the TempBook (and the power switch is in the ON position). OFF, if power is not present.
ON-LINE	This LED is ON when the TempBook is in an Active state. OFF, when the TempBook is not enabled or in the printer pass through mode.
A/D ACTIVE	ON during an A/D scan sequence. If the sequence has a small number of steps and occurs infrequently, this indicator will only flash briefly.



Front Panel of TempBook/66

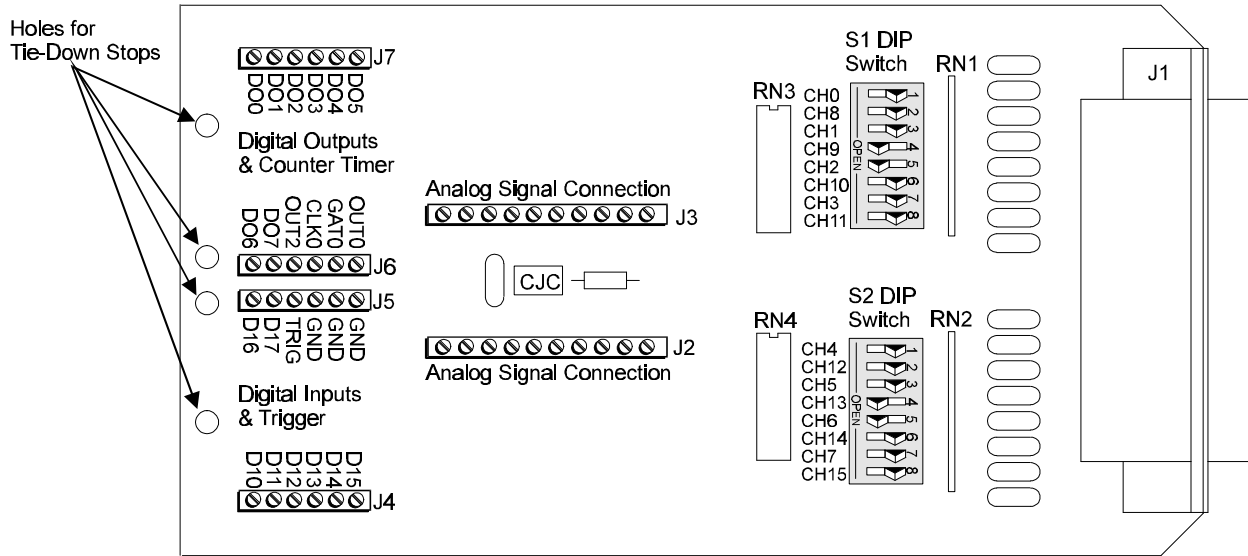


Rear Panel of TempBook/66

TempBook Front and Rear Panels

Termination Card and I/O Connectors

The TempBook/66 accepts all analog and digital I/O signals via a removable termination card (see figure). This termination card has six screw terminal strips, a CJC temperature sensor, and input R/C filters which also provide a bias current return path for the TempBook's instrumentation amplifier. The filters/bias resistors are switch-selectable on a per channel basis.

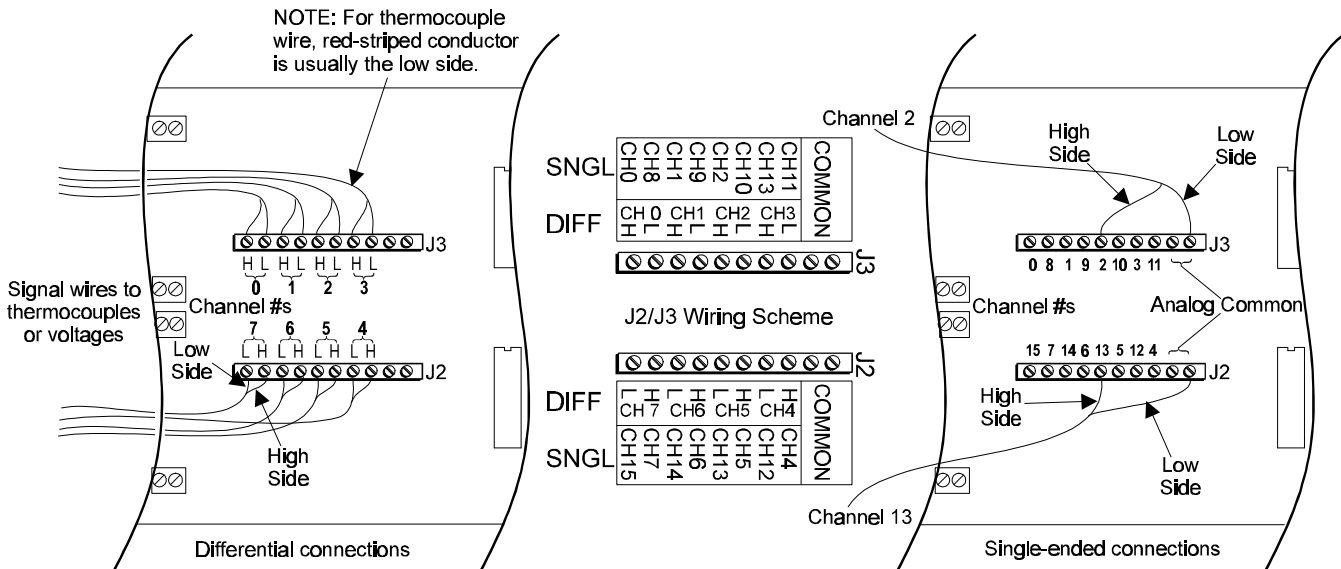


TempBook Termination Card

Terminal strips J2 and J3 provide analog input signal connection. Note that the terminals are arranged in differential pairs for easy thermocouple connection. These terminal strips also provide analog signal common connections for convenient referencing of single-ended input signals.

Terminal strips J4 and J5 provide connections for the eight digital outputs and the TTL trigger input. These terminal strips also provide ground connections for referencing digital signals. Terminal strips J6 and J7 provide connections for the eight digital inputs, the pacer clock output (OUT2), and the three counter/timer (8254 P0) I/O signals.

The thermocouple or voltage connections should be made as shown in the following figure.



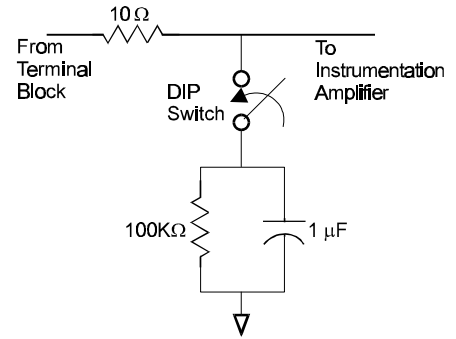
Analog Signal Connection

Note: When connecting thermocouple and other low-level signals in addition to high-level signals, the low-level signals should be connected to the lowest numbered channels with connections following in ascending order of signal magnitude.

Each of the 16 analog input channels is configured as shown in the figure (also, refer to the appendix for more information on wiring differential inputs).

The series resistance and shunt capacitance form a single-pole low-pass filter with a corner frequency of 15.9 kHz. The shunt resistance provides the bias current path for the instrumentation amplifier.

When reading thermocouples, these filters should be switched in. If the filters are not used with thermocouples or any other differential input, then the user must provide a bias current return path to signal common. **Note that for each differential channel, two DIP switches need to be set.**



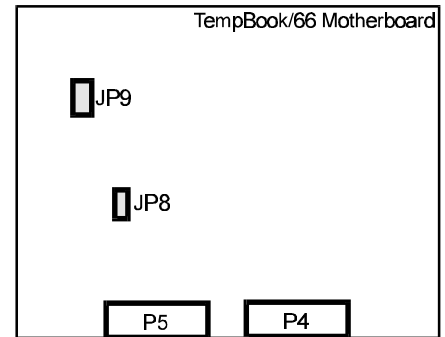
Internal Configuration

To open the unit, place the TempBook on a flat surface. Remove the screw on the top rear of the case, and slide out the top cover. Reverse this procedure to assemble the unit.

The internal configuration of a TempBook/66 consists of setting the following jumpers to reflect the desired mode of operation:

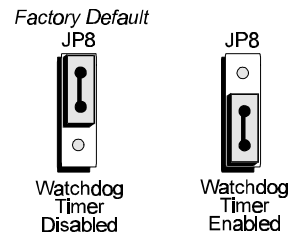
- Time Base (JP9)
- Watchdog Timer Enable (JP8)

The location of each jumper is shown in the figure.



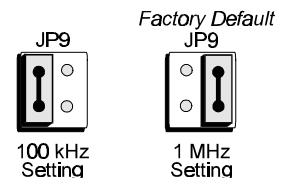
Watchdog Timer Enable/Disable (JP8)

This 3-pin header allows the elective use of the TempBook watchdog timer function. If using a printer with the TempBook, the watchdog timer should be enabled to allow the TempBook to be most reliably reset by the host computer. Note that enabling the watchdog timer might impede background measurements. If the user is not going to attach a printer, the timer is optional. The default setting is Watchdog Timer Disabled. To enable, place the shunt jumper in the enabled position as shown in the figure. To disable, place the jumper in the disabled position, as shown.



Time Base Selection (JP9)

This 2x2 header allows the user to select one of two oscillator derived frequencies to be applied to the pacer clock (8254 P1 & P2). The pacer clock sets the interval between scans in continuous trigger mode. The two frequencies are 1 MHz and 100 kHz. The most useful range of clock output frequencies for the average user would be provided by the 1 MHz setting (the default setting).



Hardware Installation

Connect the TempBook to any PC parallel printer port (female DB25) by unplugging the printer cable and plugging the male end of the supplied cable (CA-35) into the computer and the female end into the mating connector on the TempBook. Any printer port (LPT1, LPT2, or LPT3) may be used but should be noted for use in software installation.

TempBook allows for LPT pass-through for simultaneous data acquisition and printer operation. When using a printer in the system, attach the original printer cable male DB25 into the mating connector on the TempBook.

The TempBook may be powered by the supplied AC adapter that plugs into any standard wall outlet or from an isolated 7-20 VDC source of 1-2 A.

If using the power adapter, plug it into a 120 VAC outlet, and attach the low voltage end to the jack on the TempBook. Turn ON the power switch, and the POWER LED should be on.

At power-on, the printer should behave normally and can be checked by issuing a **PRINT SCREEN** command (or any other convenient method of checking the printer). (Installation of the software will be necessary before the TempBook can perform any functions.)

Rechargeable Battery Module (DBK30A)

For portable applications where external AC or DC power is not available, the DBK30A rechargeable nickel cadmium battery module can be used with the TempBook/66. This module is housed in a rugged metal package that is the same size as the TempBook. It also comes with high-strength Velcro tabs that allow convenient mounting underneath the TempBook/66.

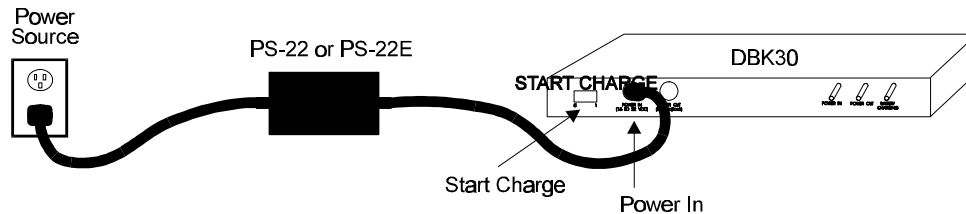
The DBK30A is a revised combination of the DBK30 and DBK31 battery modules. There is an internal slide switch which determines whether a DBK30A will act as a DBK30 or DBK31.

Note: Only the DBK30 mode should be used with the TempBook/66.

The DBK30 operating mode provides 14.4 VDC at 3.4 A-Hr. This setup can power the TempBook for 3 to 6 hours depending on the application. The battery module has built-in automatic battery-charging circuits, which quickly and safely recharge the internal batteries when connected to the supplied AC adapters. The only requirement for trouble-free operation is for the user to fully charge the battery module before attempting to use it.

Charging the Battery Modules

The DBK30A package includes a charger for the intended line voltage (either 120 VAC or 230 VAC). To charge the battery module, plug the output cable from the charger into the POWER IN connector on the DBK30A, and plug the charger into an appropriate source of AC power (see figure). The charge cycle will be initiated automatically, and the batteries will be fully charged after the charging cycle terminates.



Connecting the Charger

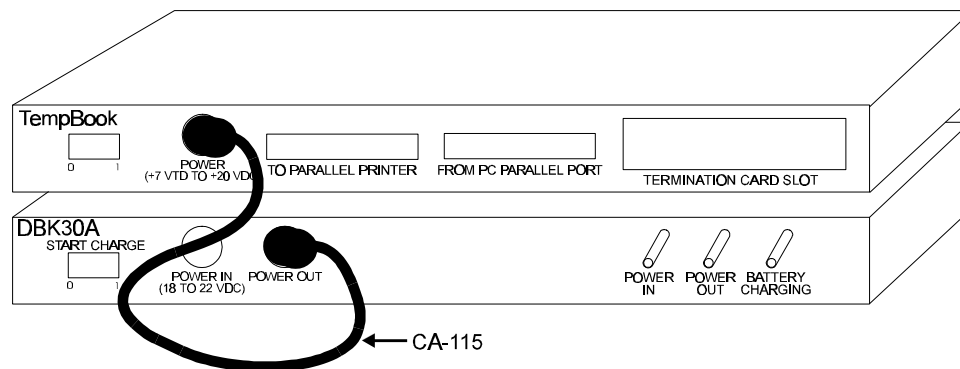
To initiate another charge cycle, depress the START CHARGE momentary rocker-arm switch. Subsequent charge cycles applied to a fully charged DBK30A will have no ill effect because the DBK30A will sense the fully-charged status of the batteries and revert to the trickle-charge state within a few minutes.

There 3 LED status indicators on the DBK30A provide information on the charge process or external load as noted in the table.

Power In	Illuminated when the charger is connected to a source of AC power and to the battery module.
Battery Charging	Illuminated steadily while battery is in the high current (2-amp) charge mode. Flashing briefly, one or two flashes at a time, when the internal batteries are fully charged.
Power Out	Illuminated steadily when an external TempBook product is connected and drawing current from the battery modules.

Battery Module Connection

The DBK30A package includes a short connecting cable (CA-115) to connect to the TempBook. This cable connects the Power Out connector on the DBK30A to the Power In connector on the TempBook (see figure).



Connecting the TempBook to the Battery Module

The run time available from a DBK30A operating a TempBook will depend on what the TempBook is doing. This time can be as little as 3 hours or as much as 6 hours.

CAUTION	
	<p>The DBK30A must periodically be fully discharged or the cells may develop "lazy chemistry" that may limit storage capacity. (This memory is characteristic for nickel-cadmium batteries.) To manually discharge a battery pack, connect a TempBook, and leave it turned on until the indicator lights go dark.</p>

Parallel Port Capabilities

The TempBook includes a test program which verifies your computer's parallel port, testing its standard and enhanced capabilities.

- Standard LPT ports on an IBM compatible computer have two modes to read data from the printer port: 4-bit and 8-bit. The 8-bit mode is somewhat faster than the 4-bit, but not all printer ports support this mode.
- Enhanced parallel ports (EPP) are parallel ports which include additional hardware that allows the TempBook to communicate with the PC at 3 to 10 times the speed possible with standard parallel ports. This greatly improves data acquisition performance.

Enhanced parallel ports require special hardware, and only certain computers are EPP capable. Most laptop computers that use the Intel 386SL or 486SL chip set (which includes the 82360 I/O Controller) are EPP capable. For those computers which have plug-in card slots, EPP adapter cards are available. You may wish to contact your computer's manufacturer for details about your machine and possible adapter cards.

The TempBook test program, TEMPTTEST.EXE (described in a later section) should be run to determine the PC's printer port capabilities.

Note: To take advantage of the improved performance of EPP when using a programming language, you must add the software command `tbkSetProtocol` (standard API) to your program. For details, refer to the command description in the Command Reference section of this manual.

TempBook Software Installation

The TempBook/66 software installation consists of two steps: making a backup copy and installing the user-selected files. The software can be installed under DOS, Windows 3.1, or Windows 95/NT. A description of each method follows. **Note:** any previous installation of the TempBook drivers should be removed before installing a new version.

Making a Backup Copy

Before attempting to load the software, make a backup copy of the release disks according to directions for your operating system. Store the original disks in a safe place, and use the copies for software installation.

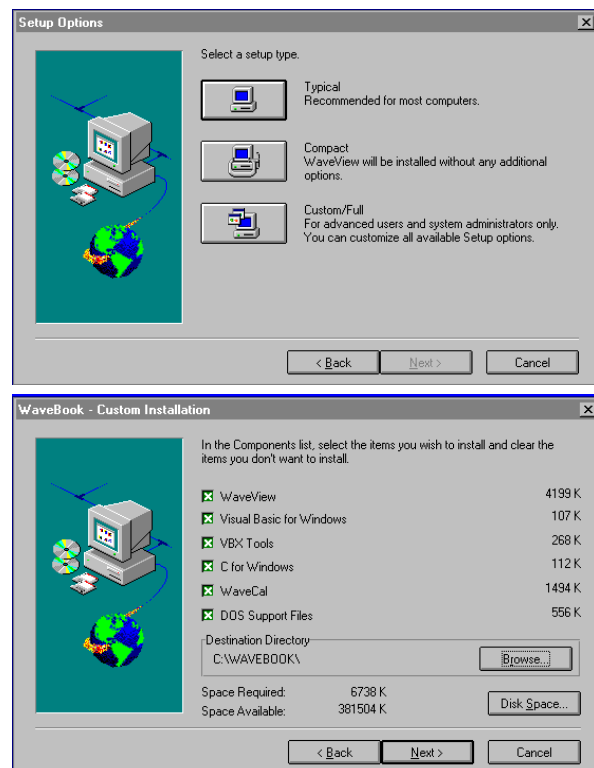
DOS Installation

1. Move to the A drive by typing A: at the DOS prompt.
2. Type: INSTALL to run the install script which describes how to install the software onto a hard disk. The script also allows the user to select which DOS support files are to be installed. The destination disk and path must be specified. For example, to install the software into the \TempBook directory of the C drive, type the command **INSTALL C:\TEMPBOOK** at the DOS prompt. If the installation file is run with no parameters, it will print a help message on screen.

Installation Under Windows 3.1

The TempBook Windows Install program copies the user-selected files and creates a Windows program group and icons.

1. After exiting other programs, begin installation by putting disk #1 of the Windows 3.1 set into your floppy disk drive. From the program manager choose File/Run to run the **SETUP.EXE** file.
2. The Windows Setup dialog box asks you for a destination directory.
3. From the next dialog box, select the Typical, Compact, or Custom/Full installation (see figure). After the options have been selected, the bottom of the screen will display the amount of hard disk space required for the installation and the amount of disk space remaining after installation. Make sure there is room available for installation before continuing.
4. After installing the files from disk #1, the program will prompt you to place the rest of the installation disk set into the floppy drive. As an option, DOS drivers may also be installed. A final message will display when installation is complete.



Note: To verify proper performance, TempTest (described in the next section) should be run after installation under Windows 3.1 (installation under Windows 95/NT has its own hardware test).

Running TempTest

Once the software is installed, the TempBook test program (TEMPTEST.EXE) should be run. TempTest verifies the PC's printer port capabilities (standard and enhanced) and estimates the port's maximum performance. To run the TempBook test program, type **TEMPTEST** at the DOS prompt.

TEMPBook Test Program			
Do you wish to test high-speed (enhanced parallel port) operation (Y/N)? N			
TEMPBook Test Summary			
	LPT1	LPT2	LPT3
I/O Address	0378	0278	03bc
EPP Type	---	---	---
TempBook Detected	Yes	No	No
Reliable Operation	8-bit	---	---
Output Speed	67 kB/sec	---	---
EPP Input Speed	---	---	---
8-bit Input Speed	131 kB/sec	---	---
4-bit Input Speed	86 kB/sec	---	---

For proper Enhanced Parallel Port (EPP) operation, both TEMPTEST and the TempBook driver must be informed of the specific EPP implementation available on your computer. If you select Y to test for EPP operation, TEMPTEST provides a selection of known EPP computers and interfaces and allows you to choose which one to test.

When TempTest is complete, a test summary will display. It verifies the following:

IOAddress	The I/O base address of the LPT port. All available ports are automatically detected by TempTest.
EPP Type	If "Yes" was selected to test EPP, this field will display the type of EPP implementation used.
TempBook Detected	"Yes" if a TempBook was detected on the port under test. If "No," the following tests are not run.
Reliable Operation	"Yes" if all communication tests with the TempBook pass.
Output Speed	An estimated rate at which data can be sent out to the TempBook.
EPP Input Speed	An estimated rate at which data can be read from the TempBook in EPP mode. Note that this will vary depending on the manufacturer of the printer port.
8-bit Input Speed	An estimated rate at which data can be read from the TempBook in 8-bit mode.
4-bit Input Speed	An estimated rate at which data can be read from the TempBook in 4-bit mode.

The driver software is then configured for EPP with the `tbkSetProtocol` software command which is described in the Command Reference Chapter. If TEMPTEST is unable to identify a reliable method of communication, contact the factory.

Installation Under Windows 95 and Windows NT

This section describes the installation of TempBook software under the Windows 95 and Windows NT operating systems. Installation for both Windows 95 and Windows NT are operationally equivalent. In fact, they use the same installation disk set. The installation program automatically detects which operating system is running and then proceeds accordingly.

Two preliminary steps are to:

- Locate the disk set labeled *TempBook Software for Windows95/NT* and make a backup copy of the disk set before proceeding.
- (If the TempBook is to be used immediately) Attach the TempBook to the desired LPT port and power-on the system before starting the software installation.

Now insert the diskette labeled *Disk 1* of the Windows 95/NT set into the floppy drive. Locate the drive and double-click on the file `Setup.exe`. The installation process begins and displays a Welcome screen with copyright and other information regarding the product. Click Next> to continue with the installation. Click Cancel to exit without installing the software. **Note:** If other Windows programs are running, select Cancel, exit all other programs, and return to TempBook installation when ready.

The next screens allows you to specify the program folder in which to install the software and then various options of program components. Use your mouse to click your selections, and follow on-screen instructions.

The next screen shows the progress of the installation of the software components. As the installation progresses, you may be asked to periodically insert the next disk into the floppy drive. When prompted, replace the current disk with the next disk. When the software component installation is complete, you are given 3 options:

- Exit and perform device configuration (If the TempBook is to be used immediately, you should select this configuration option and proceed to the next section).
- Exit and review the readme file for up-to-date information on the current release version.
- Exit completely and return to your operating system.

TempBook Configuration Under Windows 95/NT

This section describes the configuration of TempBook devices under the Windows 95 and Windows NT operating systems. A configuration utility is supplied via a control panel applet. The **Daq Configuration** applet allows you to add a device, remove a device, or change existing configuration settings. Daq Configuration also has a built-in test utility to test the device. The test utility provides feedback on the validity of the current configuration settings as well as providing relevant performance summaries.

Daq Configuration can be found in the Windows 95/NT control panel and can be executed any time that it is desirable to add, remove or change device configuration settings. Daq Configuration may also be entered during driver installation. The following description applies to either method.

The Daq Configuration/Device Inventory screen at right will display all currently configured devices. Displayed devices are indicated by their name and an identifying icon which indicates the device type. If no devices are currently configured, no devices will appear in this field.

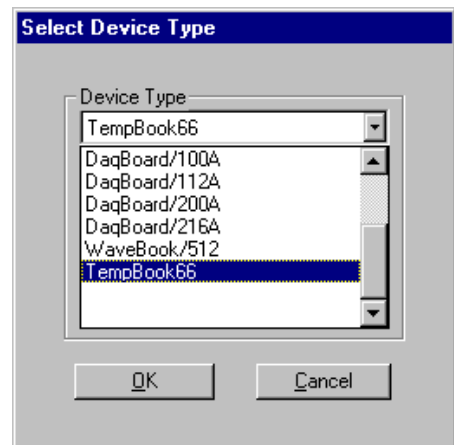
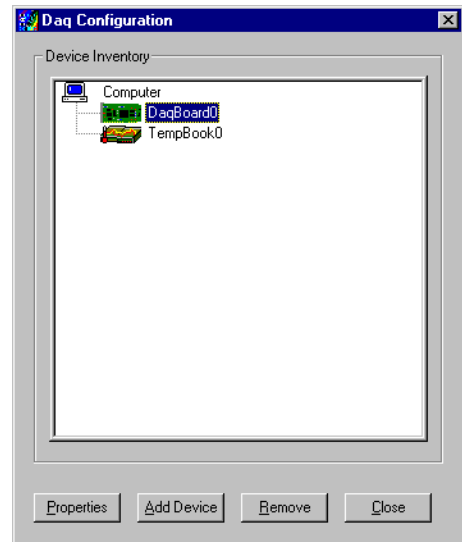
The 4 buttons across the bottom of the Daq Configuration screen are used as follows:

- **Properties.** Current configuration settings for a device can be changed by bringing up the corresponding properties screen. To do so, double-click the device icon or single-click the device and then double-click the **Properties** button.
- **Remove.** The **Remove** button is used to remove a device from the configuration. A device may be removed if it is no longer installed or if the device's configuration no longer applies. **Note:** if a device is removed, applications may no longer access the device. However, the device can be re-configured at any time using the **Add Device** function described below.
- **Close.** The **Close** button may be used at any time to exit the Daq Configuration applet.
- **Add.** The **Add Device** button is used to add a device configuration whenever a new device is added to the system. Failure to perform this step will prevent applications from properly accessing the device. Double-clicking the **Add Device** button will display a window as shown in the figure.

Use the scroll bar to find the TempBook device type to be configured. Once found, click the device type (the type should then appear in the main edit box). Now double-click the OK button.

The next screen displays the properties for the TempBook device with the default configuration settings. Fields include:

- The **Device Name** field is displayed with the default device name. However, this field can be changed to any descriptive name as desired. This device name is the name to be used with the `daqOpen` function (see enhanced API chapter) to open the device.
- The **Device Type** field should indicate the device type which was initially selected. However, it can be changed here if necessary.



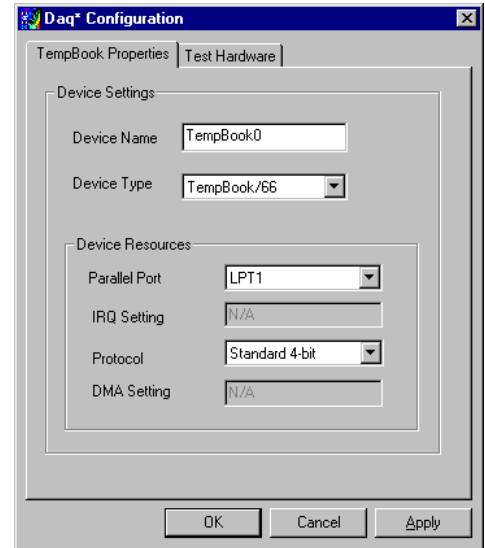
- The **Protocol** field is used to set the parallel port protocol for communicating with the TempBook. Depending on your system, not all protocols may be available.

Note: **IRQ Setting** and **DMA Setting** for the TempBook are currently not configurable. These fields are reserved for future use.

When all fields have been changed to the desired settings, you can click:

- the **Apply** button to store the configuration.
- the **OK** button to store the configuration and exit the current property screen.
- the **Cancel** button to exit the current device configuration property screen without storing any changes.
- the **Test Hardware** tab to test the current stored configuration for the device. This selection will bring up the Test property screen. **Note:** the next figure displays results from a previously run test. Initially, the screen will show no test results.

Before testing the TempBook, make sure the device has been properly installed and powered-on. Make sure the parallel port cable is firmly in place on both the TempBook and the proper LPT port in the computer. **Note:** Testing the TempBook device may, in some cases, cause the system to hang. If test results are not displayed in 30 seconds or the system does not seem to be responding, reboot the system. Upon power-up, re-enter the Daq Configuration and change the configuration settings to those that work properly. To test the current stored configuration for the device, click the **Test** button. Test results should be displayed within a few seconds. The test results have 2 components: Resource Tests and Performance Tests.



Resource Tests

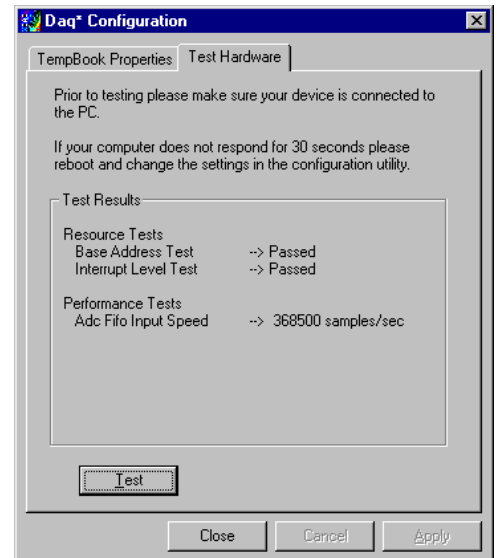
The resource tests are intended to test system capability for the current device configuration. These tests are pass/fail. Resource test failure may indicate a lack of availability of the resource or a possible resource conflict.

- **Base Address Test** - This test will test the base address for the selected parallel port. Failure of this test may indicate that the parallel port is not properly configured within the system. See relevant operating system and computer manufacturer's documentation to correct the problem.

Performance Tests

The performance tests are intended to test various TempBook functions with the current device configuration. These tests give quantitative results for each supported functional group. The results represent maximum rates at which the various operations can be performed. These rates depend on the selected parallel port protocol and will vary according to port hardware capabilities.

- **ADC FIFO Input Speed** - This test will test the maximum rate at which data can be transferred from the TempBook's internal ADC FIFO to computer memory through the parallel port. Results are given in samples/second (sample is 2 bytes in length representing a single A/D count).



Connection Troubleshooting (Windows 95/NT)

If communications cannot be established with the TempBook or if trying to connect causes the system to hang or crash, then you should:

- Check that TempBook's power LED is ON. If not ON, verify power connection between the TempBook and the power source.
- Make sure the LPT cable is firmly attached to the computer's proper LPT port and to the TempBook port labeled "TO COMPUTER".
- Check that the desired LPT port has the proper resource configurations. The base address and IRQ level must be properly configured and recognized by the operating system. The parallel port must be capable of generating interrupts for proper operation. (This information may be obtained in the Device Manager in the Control Panel of the operating system). More information on this subject can be found in the `readmew.txt` file in the current software release.
- Check the BIOS settings for the LPT port. Make sure the BIOS LPT protocol settings are compatible with the settings selected for the LPT port with the Control Panel applet.
- Make sure the Daq Configuration Applet has been run and the proper LPT port and protocol have been selected for the device. The Daq Configuration applet can be found in the Control Panel of the operating system. The Test Hardware function in the control panel applet can be used to confirm proper communication with the device.
- WINDOWS NT: Make sure the driver has been loaded. The installation will configure the operating system to automatically load the driver at bootup. However, if there is a problem communicating with the device, the driver can be loaded manually by using the following start sequence from a DOS shell:
`NET START TEMPBK.` To unload the driver manually, use the following sequence: `NET STOP TEMPBK.`

Calibration of TempBook

The TempBook/66 is factory calibrated for gain and offset. After calibration, the unit is characterized for gain and offset errors, and software correction constants are calculated. For users who wish to use these constants within their own programs, see the Software Calibration and Zero Compensation chapter. For users who wish to use these constants within DaqView, follow the next section.

Calibration Constants File Installation

Each TempBook is shipped with a disk containing a calibration constants text file. The file is named serial_no.cal where serial_no is the serial number of the TempBook for which the constants were generated.

The default calibration constants filename assumed by TempView is tempbook.cal. If you want to use DaqView with software calibration, then the file serial_no.cal must be renamed tempbook.cal and placed in the DaqView working directory, usually C:\TEMPBOOK\WIN\UTILS.

The following command can be used to copy and rename the file if executed from the floppy disk drive.

```
COPY SERIAL_NO.CAL C:\TEMPBOOK\WIN\UTILS\TEMPBOOK.CAL
```

Hardware Calibration


Since the TempBook is factory characterized after calibration, user recalibration is not recommended. However, one exception to this recommendation is the calibration of the TempBook's internal 5V reference. This 5V reference is used to level-shift the amplified analog input signal for unipolar measurements. Reference-voltage drift is not compensated for with zero compensation; and therefore, periodic recalibration can improve unipolar measurement accuracy.

The following characteristics can be calibrated through potentiometer adjustments on the TempBook main board:

- 5V Reference Voltage
- Instrumentation Amplifier Offset
- Level Shift Amplifier Offset
- ADC Offset
- ADC Span

The TempBook includes a simple calibration program which can be used to perform these adjustments. If the user is only performing the reference voltage adjustment, then only a 4-1/2 digit DMM is required. If the user is performing the full TempBook calibration, then an adjustable voltage calibrator is also required.

A Microsoft Windows program, TEMPCAL.EXE, is provided to perform calibration of the TempBook. To use this program, install the TempBook Windows support and launch the TEMPCAL program; follow the on-screen instructions.

 *Notes*

TempView is a graphical Microsoft Windows application for operating the TempBook hardware with a 16-bit PC. No programming knowledge is required to operate this application. TempView allows you to set up an application to acquire data and save it directly to disk with the ability to transmit the data to other Windows applications, such as spreadsheets and databases. TempView provides the following capabilities:

- Set up all of the analog input parameters, then acquire and save the data to a disk file.
- Exercise TempBook's digital I/O section.
- Exercise the counter/timer.
- Launch PostView, an independent application to graphically view waveforms previously recorded by TempView.

Application Startup

If you have not already installed TempView, refer to chapter 2 of this manual. To launch the application, double-click the TempView icon. TempView holds many user-configured parameters which can be saved to disk. The default configuration file name is "TEMPVIEW.CFG". TempView will then proceed to search the working directory for the TEMPVIEW.CFG configuration file. The following conditions may apply:

- If the default configuration file is found, all the required setup information will be extracted from it, and the application's main window will open.
- If the default configuration file is not found, TempView will try to connect with the TempBook hardware with the following default parameters: Printer Port LPT1, Interrupt Level 7, and 4-bit Standard Protocol.
- If connection with the TempBook is established, the applications main window will open with default setup.
- If all of the above fails to establish communications with the TempBook, then a dialog box will appear asking whether or not you want to open a different setup file. Answering "Yes" will open another dialog box where you can select your file. If this is the first time you have used this application, no saved setups will be available; select "No".
- If a configuration file is selected, no further dialog boxes will appear, all the required setup information will be extracted from it, and the applications main window will open.
- If no user configuration file is selected or found, or communications is not established, the next dialog prompts the user to choose a real instrument or a simulated instrument. If the hardware is not available and you just want to try out the software, select Simulated Instrument. The Simulated Instrument mode allows you to exercise all the software functions without any hardware installed. If the TempBook hardware is connected and switched on, select Real Instrument.
- If Real Instrument was selected, an additional dialog box will appear, prompting you to select the LPT port that the hardware is connected to, the interrupt level of that port, and the parallel port protocol to use.

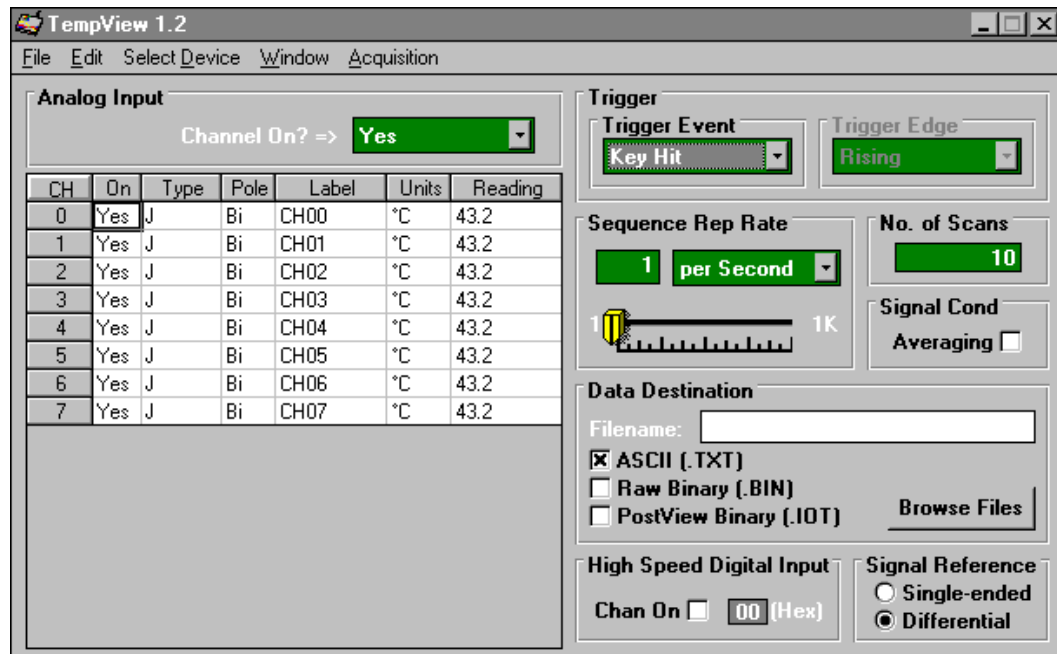
TempView will attempt to find the TempBook on the specified LPT port. If the hardware is found, the application's main window will open. If no hardware is found, a message appears and the application will open with the controls disabled.

To reconfigure the LPT port setting and try again, click Select LPT Port under the Select Port menu. If the TempBook hardware still cannot be identified by the software, exit TempView and try the TEMPTEST utility program.

TempView Components

The figure shows TempView's main window that has areas for the following:

- Analog input spreadsheet to summarize system configuration and show current readings
- Trigger configuration, event type and edge
- Sequence repetition rate, number of scans, and averaging enable
- Data destination (filename and data format)
- High-speed digital input enable
- Signal reference type



TempView's Main Window

Analog Input Spreadsheet

The input spreadsheet allows the analog input channels to be configured and displayed. The spreadsheet consists of rows and columns much like a standard accounting spreadsheet.

- Each **row** configures a single analog input channel. There are 16 rows if single-ended inputs are selected; or 8 rows if differential inputs are selected.
- The **columns** contain the configuration information for each channel. Some columns allow blocks of cells to be altered simultaneously while others allow only one cell to be changed at a time. Clicking a column header will select the entire column if possible.

The 7 columns for each row are used as follows:

CH - The channel-number column labeled "CH" is static and cannot be altered. This column identifies rows by the corresponding analog input channel to configure.

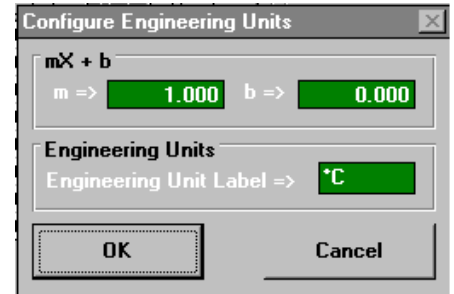
ON - The column labeled "On" allows you to select whether or not data will be collected from the corresponding input during an acquisition. An acquisition consists of reading data to disk, either to charts or the input reading column of this spreadsheet. When a cell or block of cells in this column is selected, a selection box will appear above the spreadsheet that allows a "Yes" or "No" choice to enable or disable the channel. Double-clicking a cell in this column will toggle the channel status. The Make All Channels Active and Make All Channels Inactive menu items under the Edit menu can be used to globally change all channels to a "Yes" or "No".

Type - The "Type" column allows you to set the thermocouple or gain type for the selected channel or block of channels. Double-clicking a cell or block of cells in this column will select the next available gain or type.

Pole - The "Pole" column shows the state of the channel polarity which can be either unipolar or bipolar. The channel polarity can be programmed on a per channel basis. If a cell or block of cells in this column is selected, a selection box will be displayed above the spreadsheet with the selections "Uni" and "Bi" when the cell is selected. Double-clicking in one of these cells will toggle the polarity.

Label - The "Label" column contains a descriptive name for the input channel. By default it contains a label similar to its channel number; however, you can enter a more relevant, descriptive label of 8 characters. This label will be used when selecting a specific channel in the analog trigger and chart selection lists. This column does not have a selection list above the spreadsheet and does not allow selecting blocks of cells.

Units - The "Units" column allows you to change the engineering units of each channel and apply a linear equation to the data read from the TempBook. When a cell or block of cells in this column is selected, a selection box is displayed containing $mX+b$ as well as common engineering units (see figure). Selecting $mX+b$ will pop up a window that allows the m and b of this equation and the engineering units label to be defined. The engineering units will then be displayed in the "Units" column, and the $mX+b$ equation will be applied to the reading from the TempBook before the reading is displayed or written to disk. The X in this equation is the voltage or the degrees in Celsius read back from the TempBook. For example, if a TempBook channel is configured as bipolar and unity ($\times 1$) gain, the default voltage would be ± 5 V. This corresponds to an m of 1, a b of 0 and an engineering unit of V . This could be changed to millivolts by setting m to 1000 and units to mV . This column could also be used to perform a software calibration of the TempBook. This is performed by reading known inputs at two different points of the input voltage range (usually at 0 and full scale) and solving the equation $y = mX+b$. The full-scale voltage, which changes according to the gain of the channel, is $5V/gain$ for bipolar channels and $10V/gain$ for unipolar channels.



Reading - The "Reading" column can display the analog input readings of the TempBook. This column cannot be altered by the user and is enabled by selecting Enable Input Reading Column under the Acquisition menu. This column will update the readings as fast as the computer will allow. If data is being written to disk while this column is enabled, it will be updated whenever possible. The spreadsheet cannot be altered while the input reading column is enabled.

In addition to the analog inputs, there is one 8-bit digital input channel accessible on the termination card. This high-speed digital input, which is read at the same rate as any analog inputs, can be enabled or disabled by clicking the "ChanOn" checkbox in the lower right of the analog input spreadsheet.

Acquisition Configuration

The acquisition configuration section of TempView's main window is just to the right of the analog input spreadsheet. This section has 5 parts: the Trigger, Scan Frequency, Number of Scans, Signal Conditioning, and Data Destination sections. These sections allow you to set up all of the acquisition parameters for the analog inputs and the high-speed digital input channel. The analog acquisition configuration includes parameters for setting up a trigger source, the scanning frequency after the trigger is satisfied, the number of scans to take after the trigger, and the file name for the collected data. These settings will be used when an acquisition-to-disk is started by selecting "Go" under the acquisition menu.

Trigger - The Trigger section selects the source of the trigger. When the trigger is satisfied, the scans are collected at the selected scan frequency and stored to disk. The sources are: Key Hit, which arms the acquisition and waits for the user to hit a key; and External TTL, which waits for a falling or rising edge on the 'trig' input terminal on the termination card.

Sequence Rep Rate - The scan frequency can be set in units of seconds, minutes and hours. Moving the slide switch changes the rate. The cursor can also be placed in the numeric field and a number can be entered directly. The maximum scan frequency is dependent on the number of channels that are enabled and whether or not averaging is enabled. Enabling more channels or enabling averaging will lower the maximum scan frequency.

Signal Cond Averaging - The Averaging checkbox enables or disables averaging of the analog input data. Averaging can be used to increase the effective accuracy of a noisy signal. Averaging will increase the

actual scan frequency and number of scans, but the perceived scan frequency and number of scans (which is set by TempView) does not change.

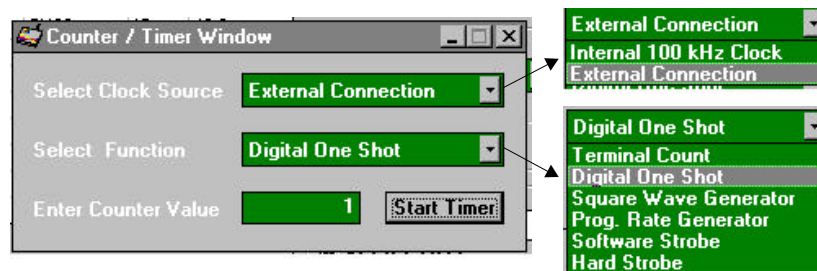
No. of Scans - The number of scans can range from 1 to 100,000. A scan includes all of the channels that are marked as "On" in the analog input configuration spreadsheet.

Data Destination - This section contains the file name and type of file(s) that exist after an acquisition-to-disk is complete. A file name can be typed in directly or the Browse Files button can be pressed to open a file-selection dialog box. The selected file will be placed directly into the filename field. During an acquisition, a raw binary file is created and updated as data is read. After the acquisition is complete, an ASCII text file and PostView binary file can be created if the appropriate checkbox is enabled. Both of these files can be read by PostView (described later in this chapter). If the raw binary checkbox is not enabled, the raw binary file will be deleted after creating the PostView or ASCII file.

After the acquisition is started, these parameters can not be altered.

Counter/Timer Window

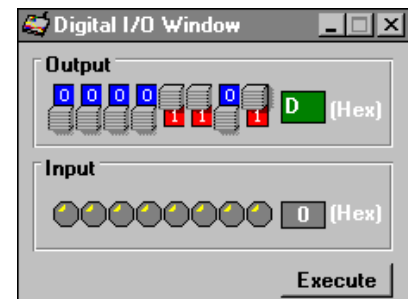
The counter/timer window is displayed when ctr/tmr is selected under the Windows pull-down menu. This window allows configuration of the 8254's counter 0. Counter 0 is configured by selecting its function, clock source and initial count. When the Start Timer button is clicked, the counter configuration will take place.



Counter/Timer Window

Digital I/O Window

The digital I/O window is displayed when digital I/O is selected under the windows menu. This window provides interactive control of the 8 digital input lines and 8 digital output lines on the TempBook termination card. Select output values by clicking on the output witches which toggle between 0 and 1—or you can enter the hex value and <Enter>; the switch positions will update to reflect the new setting. When the Execute button is pressed, all output lines will be updated and all input lines will be read.



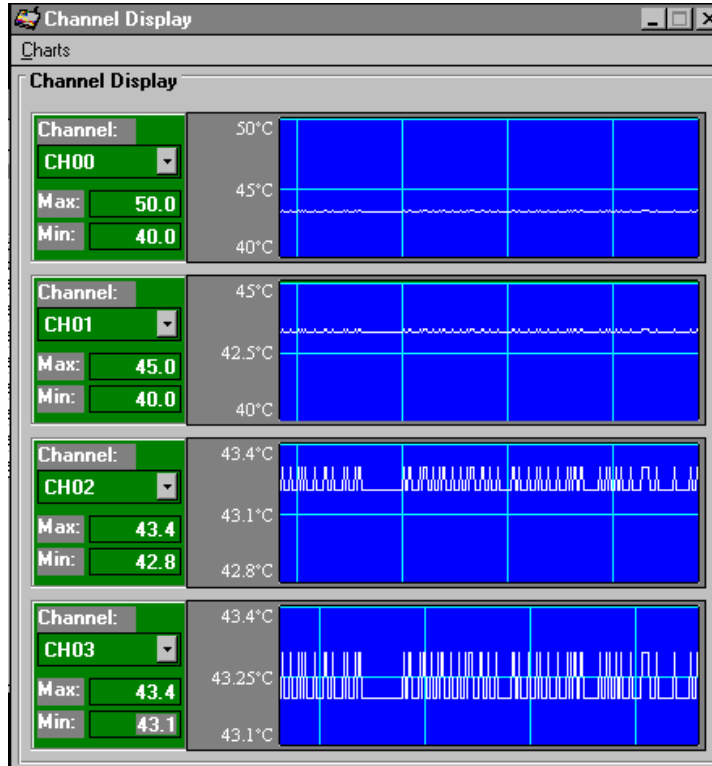
Charts and the Spreadsheet's "Reading" Column

Up to 4 charts can be displayed by selecting charts under the Window menu. Before the charts can be enabled, at least one chart must be assigned a channel through the channel's drop-down list. When a channel is selected, you can change the minimum and maximum values displayed in the chart (see figure). This can be done whether or not the charts are currently enabled.

The charts and the spreadsheet Readings column are independently enabled. The charts can be enabled under the Charts menu of the Charts window. The Readings column can be enabled under the Acquisition menu of the main TempView window. These windows provide instant feedback of the analog input readings.

Data is read and displayed in the charts and Readings column as fast as the computer will allow. When an acquisition to disk has begun using the Go command under the Acquisition menu, the charts and the spreadsheet Reading column take a lower priority, updating only when there is sufficient time in the acquisition-to-disk task. Therefore, the data seen in the charts may not be an accurate reflection of the data

that is being placed on the disk. As the scan rate is increased, the acquisition-to-disk task will take up more processor time and the charts will be unable to keep up.



TempView Display Charts

TempView Menu Items

File

The file menu provides four basic functions:

New	Set all parameters to their startup setting.
Save	Save the existing configuration for later recall.
Load	Load a saved configuration.
Convert Binary to ASCII	Convert a previously acquired binary file to an ASCII file that can be read by spreadsheets or other analysis programs.
Convert Binary to PostView Binary	Convert a previously acquired binary file to a binary file that can be read by the PostView application.
Exit	Leave the TempView program.

Edit

The Edit menu includes the following functions:

Make All Channels Inactive	This command places a "No" in the On field of all of the channels. If your channel scan includes only a few channels, it may be easier to make all of the channels inactive, then turn on only those few channels that you want.
Make All Channels Active	This command places a "Yes" in the On field of all of the channels.
Fill Down	When multiple cells within a column are selected, this command takes the top-most selected cell and copies its contents in the selected cells below.

Select Device

The Select Device menu includes the following functions:

Select LPT Port	Brings up a dialog box prompting the user to select the LPT port on which the TempBook is connected. After an LPT port is selected, TempView opens a new session with the TempBook hardware and attempts to communicate with it. If the hardware is found, the main window is opened. If no hardware is found, the user is alerted and the application is opened with the controls disabled. To reconfigure the LPT port setting and try again, click Select LPT Port under the Select Port menu. If the TempBook hardware still can not be identified by the software, exit TempView and try the TEMPTTEST utility program.
Simulated Device	This command opens a TempView session but does not attempt to communicate with TempBook hardware. Instead, the application simulates the interaction between the software and the hardware. If TempView is presently attached to real TempBook hardware, this command will close that session.

Window

The Window menu includes the following functions:

PostView	This command launches an instance of the PostView application.
Charts	This command displays the charts window.
Analog Output	This command displays the analog output window.
Digital I/O	This command displays the digital I/O window.
Ctr/Tmr	This command displays the counter/timer window.

Acquisition

The Acquisition menu includes the following functions:

Go	This command arms the hardware for an acquisition to disk. When the trigger is satisfied, the acquisition begins. All of the interactive I/O controls are disabled while the system is armed. No acquisition parameters can be altered at this time.
Enable Input Reading Column	This command reads the analog inputs and puts the numeric values in the spreadsheet in the "Reading" column. If it is already enabled, this command disables it.

Charts

The Charts menu of the charts window includes the following function:

Enable Charts	This command starts the scroll chart running. If the chart is already running, this command stops it.
---------------	---

Using PostView

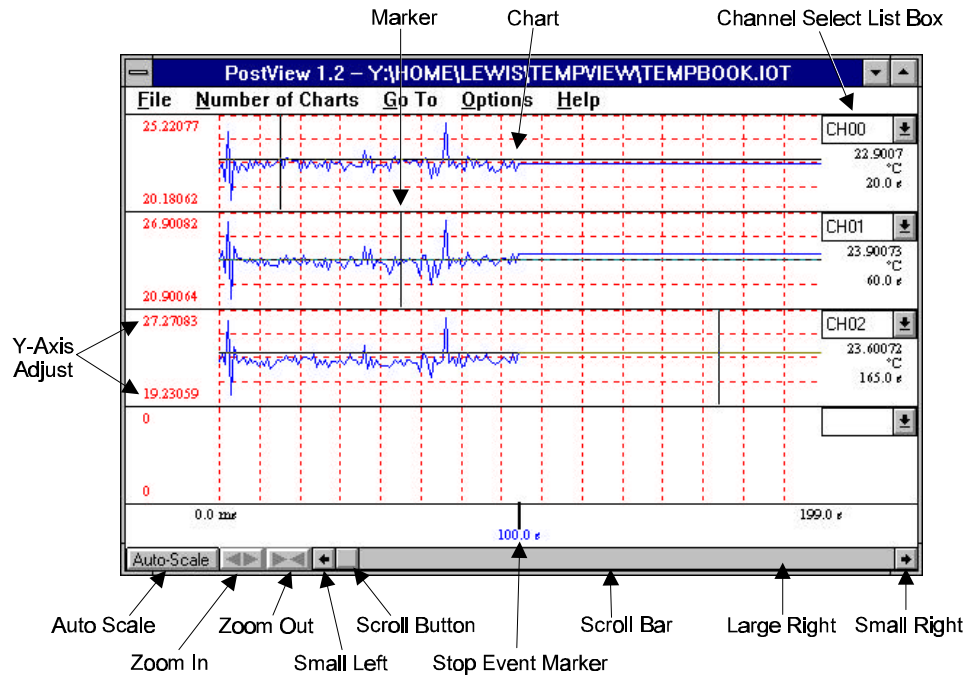
PostView is an independent application that allows the user to graphically view the waveforms recorded by TempView. As the data file is being created by the acquisition application, a descriptor file used by PostView is also created. PostView can be launched independently or launched from TempView's Window menu. Multiple sessions of PostView can be invoked concurrently to view multiple data files.

To view a data file from within PostView, select Open under the File menu. A File Open dialog box provides a means of selecting a TempView data file. To place channel waveforms into the window, select the number of charts from 1 to 16 under the Number of Charts menu item. Selecting N number of charts will automatically place the first N channels in the charts. Use the Channel Select List Box next to the chart to associate a different channel with the chart.

When PostView is launched from TempView, it automatically opens the file selected as the destination file in those applications. To view other files, use Open under the File menu.

Zoom In - The Zoom In button halves the visible timebase, showing less of the waveform. For example, if 10 seconds of information is visible, clicking the Zoom In button will show 5 seconds.

Zoom Out - The Zoom Out button doubles the visible timebase, showing more of the waveform. For example, if 10 seconds of information is visible, clicking the Zoom Out button will show 20 seconds.



PostView Charts and Scroll Bar Functions

Scroll Bar - The Scroll Bar allows the waveforms to be scrolled right or left. The Scroll Bar has 5 active areas for scrolling the waveforms, as shown in the figure below. The Small Left and Small Right scroll the waveforms left and right approximately 20%. The Large Left and Large Right scroll the waveforms left and right approximately 80%. The Scroll button shows the relative location of the visible region of the waveforms and can be dragged along the scroll bar to any location desired.

Y-Axis Adjust - The Y-Axis Adjust fields show the chart's minimum and maximum for every visible chart in the engineering units shown. Clicking the Auto Scale button automatically adjusts the Y-Axis Adjust fields. To adjust any chart's minimum or maximum, place the cursor in the desired Y-Axis Adjust field and type in a new value.

Channel Select List Box - Each chart has a Channel Select List Box to allow the user to assign any of the available channels to that chart. The Channel Select List Boxes contain labels that were assigned to the recorded channels by TempView.

Auto Scale - Clicking the Auto Scale button adjust the Y-Axis labels so that the visible waveform fills 90% of the chart's range.

Markers - Each chart contains a marker that shows the numerical values of time and magnitude at its present location in the waveform. The Markers start out at the far left of every chart, showing the time and magnitude of the first visible point. The left mouse button allows the user to drag the marker of each chart independently. The right mouse button moves the markers from all of the charts synchronously. The Options menu contains a function which allows markers to be turned on and off by the user. When a check appears in front of this item, its associated indicator is on or visible. Selecting the menu item toggles the indicator (and the check mark) on and off.

Stop Event Marker - The Stop Event Marker on the time axis shows the location of the stop point.

The PostView Timebase

PostView automatically detects the timebase of the data file and shows the time in the X-Axis labels in seconds.

Timebase for TempView - All TempView files begin at the trigger point ($t = 0$), and the time between each scan is constant.

PostView Menu Items

File

The file menu provides two basic functions:

Open	Open a data file created by TempView. PostView automatically detects whether the file contains ASCII or binary data.
Print Window	Print the present PostView window.

Number of Charts

The number of charts menu provides one basic function:

1-16	After a data file has been opened, the number of desired charts can be selected. You can also use this menu selection to change the number of charts displayed.
------	---

Go To

The go to menu provides four basic functions:

Trigger point	Automatically scrolls the waveforms so that the trigger point ($t = 0$) is in view.
Scan Number	Automatically scrolls the waveforms so that the desired scan number is in view. This menu selection invokes a dialog box which displays the number of scans in the file.
Time	Automatically scrolls the waveforms so that the desired time is in view.
Percentage	Automatically scrolls to the desired percent of the data file. For example, selecting 50% would display a waveform segment from the middle of the data file.

Options

The options menu provides two basic functions:

Grids (Ctrl-G)	Allows grids to be turned off and on for all visible graphs. When a check appears in front of an item, its associated indicator is on or visible.
Markers (Ctrl-K)	Allows markers to be turned off and on for all visible graphs. When a check appears in front of an item, its associated indicator is on or visible.

Help

The help menu provides three basic functions:

Contents	The initial PostView help screen provides an overview and listing of the help file contents. A single topic can be selected for quick access to help information.
Search	Type a word or select one from the Show Topics list for quick access to help information.
How to Use Help	Provides instructions on how to use a Windows Help system.

Overview

Note: The TempBook does not use all the features documented here; e.g., no DBKs or digital I/O). The full feature-set is described for the benefit of users running a combination of related devices.

DaqView is a Windows-based data acquisition program for operating the TempBook/66, DaqBooks, DaqBoards, or Daq PC-Cards with DBK option cards and modules. DaqView was designed for ease-of-use; programming expertise is not required. DaqView allows you to:

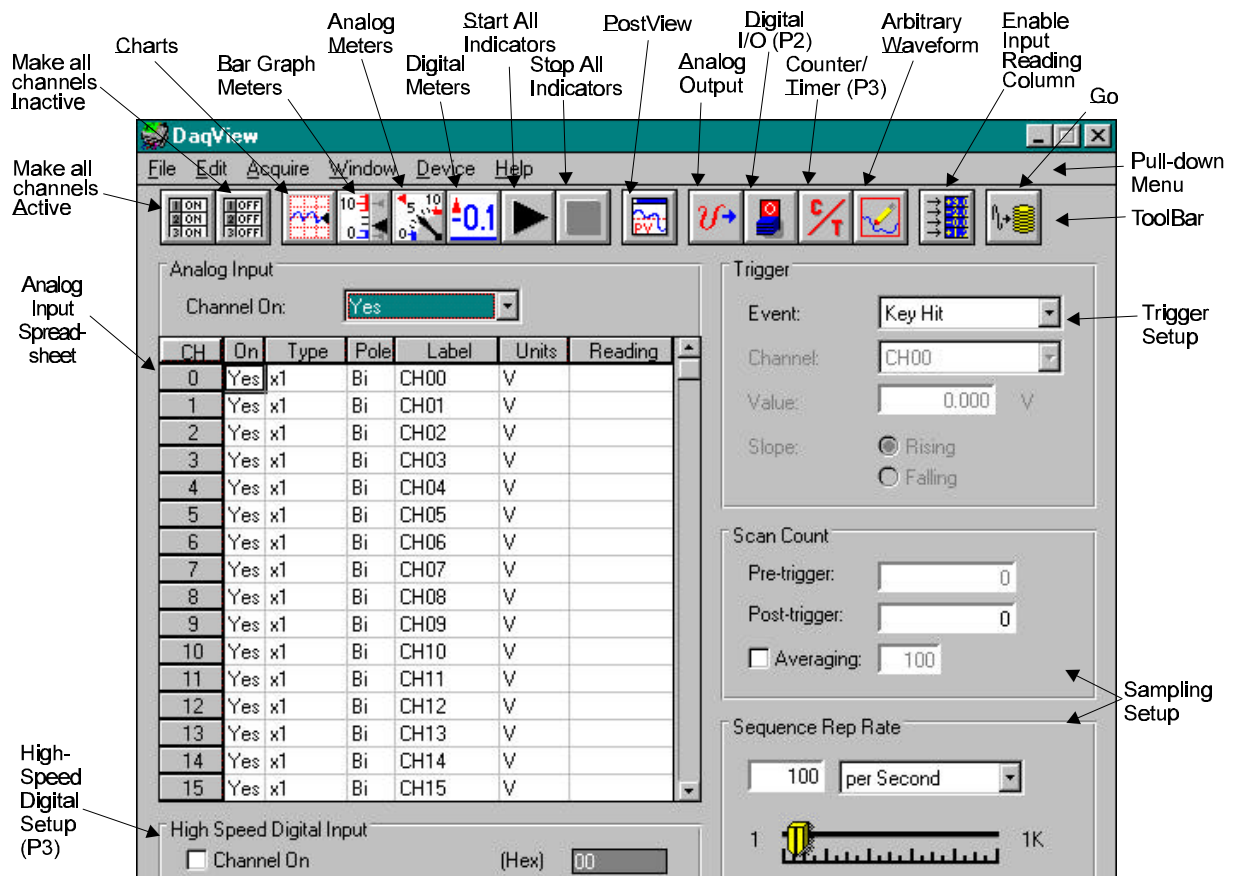
- Set up system parameters (selecting channels, gains, transducer types, etc) to acquire data.
- Save data to disk and to transmit data to spreadsheets and databases.
- Automatically re-arm the trigger function and save data in new files as needed.
- Configure and operate the DBK option modules (does not apply to TempBook).
- Configure the counter/timer into one of three modes for measuring frequency, totaling, or generating pulse trains.
- Use the 2 analog outputs, including waveform generation for DaqBoards.
- Use digital I/O (for Daq*s with digital I/O).

PostView (included) can be launched from DaqView to view waveforms for up to 16 data channels.

DaqViewXL (optional) is an “add-in” for Microsoft Excel and is discussed at the end of this chapter.

DaqView's Main Window

The figure shows DaqView's main window and basic components.



DaqView's Main Window

Starting DaqView

Before running DaqView, the program and related libraries must be properly installed—if necessary, refer to the installation chapter. Computer requirements include:

- 386 PC/AT or higher
- 8 Mbytes of RAM
- Windows 3.x or Windows 95
- IOtech data acquisition hardware
- (for DaqViewXL) Excel 5.0 or higher

To run DaqView, (double) click the DaqView icon. Depending on existing files, DaqView StartUp will begin in 1 of 3 ways:

- **Auto-Device Select with Full Setup.** DaqView looks for a special auto-start file named *daqview.daq* in the startup directory. If found, DaqView loads this setup with the specified device and configuration. This setup file (if it exists) was created during a previous session.
- **Auto-Device Select with Default Setup.** If the auto-start file is not found, DaqView will try to use settings from the program configuration file (*daqview.ini*) in the Windows directory. The default setup (equivalent to the “file, New” menu item) uses DaqBook as the device on parallel port 1 (LPT1).
- **Manual Device Select with Default Setup.** If DaqView cannot connect to a device via a setup file or defaults, it prompts you with a series of dialog boxes:
 - The first dialog box allows you to retry a connection, select a device, or load a setup file.
 - The Select Device button opens a device interface box; you can choose the hardware installed or a simulated instrument. If you select a Daq, another dialog box will ask for device parameters (printer port, interrupt level, and port protocol for the DaqBook; base address and interrupt level for the DaqBoard and Daq PC-Card). **Note:** If the hardware is not available or you just want to explore the software, select *Simulated Instrument*; the main window will open.
 - The Load File button allows you to find and open an existing setup file.

If the hardware is found, the DaqView’s main window will open. If no hardware is found, the system alerts you and gives you another chance to select the working parameters. (For a DaqBook communication problem, exit DaqView and run the utility test program.)

Beginner’s Tour of DaqView

A quick tour of DaqView will show you some of its basic features that are explained later. Proceed as follows:

1. Open DaqView as described above. If your Daq or DBK signal conditioning units are not connected, select *Simulated Instrument* as your device.
2. Select the next to last tool in the toolbar menu, Enable Input Reading Column.
 - If in the Simulated Instrument mode, the Reading column of the Analog Input Spreadsheet will display simulated data. Selecting the Enable Input Reading Column button again will freeze the Reading column’s display.
 - If working hardware is connected, the readings will quantify actual signals.
3. From the toolbar, select Bar Graph Meters, Analog Meters, or Digital meters and then the triangular Start Indicator on the new window’s toolbar. Incoming data will be displayed in your chosen format. The last item in the meter window’s toolbar allows you set the number of channels to be displayed.
4. From the DaqView main window toolbar, select Charts (third item from left), and the DaqView Channel Display window will appear. If inactive, you must first enable the desired channel in the pull-down box to the right of the chart. You can start or stop the data display with the Start or Stop Indicators buttons. After stopping the indicators, you can reset the number of channels to be displayed.
5. As an introduction to system configuration, select Device from the pull-down menu and then Configure Hardware Settings. The window Configure System Hardware will appear. On the left side of the window, select any channel’s pull-down box to reveal the external connection options. Besides the default, direct signal connection, you can choose among the DBK option cards and modules. The right side of the screen sets up digital connections if your Daq* is so equipped.

Analog Input Spreadsheet

The analog input spreadsheet (left half of main window) displays the analog input channels and allows you to configure them. Each row shows a single channel and its configuration. The number of rows may vary, but each row has seven columns. Some columns allow blocks of cells to be altered at the same time (clicking a column header can select the entire column). Other columns allow only a single cell to change. The table summarizes the function of each row.

Analog Input Spreadsheet Parameters on DaqView's Main Window	
Column	Use and Description
CH	The channel number (cannot be changed from this window). This number includes the main channel number and the expansion board number and channel (if used). Expansion channels are configured using the Hardware Configuration window described later in this chapter.
On	This column allows you to select whether data will be collected from that channel. When a cell or block of cells in this column is selected, a selection box will appear that allows "Yes" to enable or "No" to disable the channel. Double-clicking a cell in this column will toggle the channel status. The Edit menu allows you to Make All Channels Active or Make All Channels Inactive.
Type	This column allows you to set the gain or input type for each channel. The gains and types will vary among the option cards. A block of cells in this column can be selected for multiple channels with the same type of option card. Double-clicking a cell will select the next available gain or type.
Pole	This column shows the channel polarity (unipolar or bipolar) for each channel. The polarity can be programmed here on a per channel basis when using a DaqBook/200, /216 and any DaqBoard or a DBK15 Universal Current/Voltage expansion card. When using any other Daq*s, this column is set in the Hardware Setup window. For selected cells that can be changed, a selection box will display "Uni" or "Bi". Double-clicking in a cell will toggle the polarity. If the hardware cannot program the polarity, no selection box will be displayed.
Label	This column contains a descriptive name for the input channel. The default label is the channel number, but it can be changed to any 8 characters. This label is used when selecting a channel in the analog trigger and chart selection lists.
Units	This column allows you to change the engineering units of each channel and apply a linear equation to the Daq data. When a cell or block of cells in this column is selected, the analog input box displays entry options in a pull-down box. Selecting "mx+b" allows you to define m and b and the engineering units label. The engineering units will then be displayed in the "Units" column and the "mx+b" equation will be applied to the reading from the Daq before the reading is displayed or written to disk. The X in this equation is the voltage read back from the Daq (or degrees Celsius for a DBK14). For example, if a Daq channel is configured as bipolar and unity ($\times 1$) gain, the default voltage would be ± 5 V. This corresponds to an m of 1, a b of 0, and a unit of V. V can be changed to millivolts by setting m to 1000 and units to mV . The Units column can also be used to perform a software calibration of the Daq. This is done by reading known inputs at two different points of the input voltage range (usually at 0 and full-scale) and solving the equation $y = mx + b$. The full-scale voltage, which changes according to the gain of the channel, is 5 V/gain for bipolar channels and 10 V/gain for unipolar channels.
Reading	This column displays the Daq*s analog input readings. This column cannot be altered by the user and is enabled by selecting Enable Input Reading Column under the Acquisition menu or by selecting "Start/Stop All Indicators" under the window menu. This column will update the readings as fast as the computer will allow. The spreadsheet cannot be altered while the input reading column is enabled.
<p>Note: In addition to the analog inputs, there is a 16-bit digital input channel accessible on the DaqBook's P3 connector. This high speed digital input, which is read at the same rate as any analog input, can be enabled or disabled by clicking in the checkbox below the analog input spreadsheet.</p>	

Acquisition Configuration

Next to the analog input spreadsheet on the right side of the main window is the acquisition configuration section. The 3 main parts of this section include setup parameters for the trigger, the scan, and sampling. These settings will be used when an acquisition to disk is started by selecting “Go” (last item in the toolbar). When the trigger is satisfied, the scans are collected at the selected scan frequency and stored to disk in the designated file. **Note:** the high-speed digital input channel is on the lower left of the screen and can be used with Daq*s that support it.

Acquisition Configuration Options in DaqView s Main Window	
Parameter	Options, Use, or Description
Trigger Event	Selects the source of the trigger. The 4 possible trigger sources are: <i>Immediately</i> arms and executes the trigger immediately <i>Key Hit</i> arms the acquisition and waits for the user to hit a key <i>External TTL</i> waits for a falling or rising edge on pin 25 of connector P1 <i>Channel Value</i> monitors level on selected channel; executes trigger when parameter is satisfied
Trigger Slope	Specifies whether the trigger responds to a rising or falling slope on the external TTL.
Analog Trigger Level Setup	Sets the trigger based on a specified signal value on a particular channel.
Sequence Rep Rate	The scan frequency can be set in units of seconds, milliseconds, minutes, or hours via a pull-down box. Moving the slide switch or typing directly into the numeric field changes the rate. The maximum scan frequency is dependent on the number of channels that are enabled and whether or not averaging is enabled. Enabling more channels or averaging will lower the maximum scan frequency.
Scan Count	The number of scans can range from 1 to 10,000,000. A scan includes all of the channels that are marked as “On” in the analog input configuration spreadsheet. The Pre-trigger scans is the number of the scans to acquire before the executing the trigger event (available for Channel Value trigger only). Note: If Pre-trigger scans are greater than 0, Pre-trigger and Post-trigger samples cannot exceed 32508. If Pre-trigger scans equals 0, Post-trigger scans cannot exceed 10,000,000.
Averaging	The checkbox allows averaging of the analog input data to be enabled or disabled. Averaging can be used to increase the effective accuracy of a noisy signal. Averaging will increase the actual scan frequency and number of scans, but the perceived scan frequency and number of scans (which is set by DaqView) does not change.
Note: Once the acquisition starts, these parameters cannot be altered.	

Pull-Down Menu Items

Some (but not all) items in the pull-down menu can also be enabled from the toolbar. Their description in the toolbar section is more detailed than presented in this section.

File

File Menu Items and Descriptions	
New	Set all parameters to their startup, default setting.
Open	Set all parameters as directed by a specified setup file.
Save	Save the existing configuration for later recall (overwrites the existing version).
Save As	Saves the existing configuration for later recall; asks whether to overwrite the original version or save under a new filename.
Convert Binary to ASCII	Convert a previously acquired binary file to an ASCII file that can be read by spreadsheets or other analysis programs.
Convert Binary to PostView Binary	Convert a previously acquired binary file to a binary file that can be read by PostView.
Exit	Leave the DaqView program.

Edit

Edit Menu Items and Descriptions	
Make All Channels Inactive	This command places a “No” in the On field of all channels. To scan only a few channels, it may be easier to make all channels inactive, then turn on the desired channels.
Make All Channels Active	This command places a “Yes” in the On field of all of the channels.
Go To Channel F7	This command pops up a dialog box which requires a channel number. For hardware configurations with hundreds of channels, this is a quick way to get around.
Fill Down F8	For multiple cells selected within a column, this command copies the value in the top cell to all the cells below.

Acquire

Acquire Menu Items and Descriptions	
Go	This command arms the hardware for an acquisition to disk. When the trigger is satisfied, the acquisition begins. All of the interactive I/O controls are disabled while the system is armed. No acquisition parameters can be altered at this time.
Enable Input Reading Column	This command reads the analog inputs and puts the numeric values in the spreadsheet in the “Reading” column. If the Reading column is already enabled, this command disables it.
Data File Setup	This section determines the filename and type of file(s) that exist after an acquisition. A filename can be typed in directly, or the <i>Browse Files</i> button can be pressed to open a file selection dialog box (the selected file will be placed directly into the filename field). During an acquisition, a raw binary file is created and updated as data is read. After acquisition, an ASCII text file and PostView binary file can be created if their checkboxes are enabled. Both of these files can be read by the PostView. If the raw binary checkbox is not enabled, the raw binary file will be deleted after creating the PostView or ASCII file. The Multiple Destination (Auto Re-Arm) box directs DaqView to create successive files for long data runs as needed. The “Validate File Overwrite” checkbox (if checked) will require confirmation for overwriting existing binary, ASCII, or IO files.

Window

Window Menu Items and Descriptions	
Charts	Displays the charts window.
Bar Graph Meters	Displays incoming data on a bar graph.
Analog Meters	Displays incoming data on an analog dial.
Digital Meters	Displays incoming data numerically.
Start All Indicators	Starts displaying data in the Reading column and any open Chart or meters window.
Stop All Indicators	Stops displaying data in the Reading column and any open Chart or meters window.
PostView	Launches an instance of the PostView application.
Analog Output	Displays the analog output window.
Digital I/O	Displays the digital I/O window.
Counter/Timer	Displays the counter/timer window.
Arbitrary Waveform	Displays the arbitrary waveform window.

Device

Device Menu Items and Descriptions	
Select TempBook DaqBook	Configures DaqView for operating the TempBook/66. Brings up a dialog box to enter the printer port (<u>L</u> PT1 to LPT4), the interrupt level (3 to <u>Z</u>), and the port protocol (<u>8-bit standard</u> , 4-bit standard and several EPP options). Note: for Fast EPP and SMC666 port protocols, interrupt levels include 3-7, 10, 11, 12, 14, 15.
DaqBoard	Brings up a dialog box to enter a base address (0330 default) and interrupt level (<u>10</u> , 11, 12, 14, 15).
Daq PC-Card	Brings up a dialog box to enter a base address (0330 default) and interrupt level (5, 6, 7, <u>10</u> , 11, 12, 14, 15). Note: DaqView attempts to open a new session and communicate with the selected device. If the hardware is found, the main window is opened. If no hardware is found, the user is alerted to reconfigure and try again. If the hardware still is not found, exit DaqView and try the DaqTest program.
Simulated Device	Opens a DaqView session but does not attempt to communicate with hardware. Instead, the application simulates the interaction between software and hardware. If DaqView is attached to real hardware, this command will close that session.
Configure Hardware Settings	Opens the hardware configuration window which allows the user to tell the software how the hardware is set up. Setup sections include: Analog Input Option Cards (DBKs), Digital Option Cards, Input Polarity, A/D Signal Reference, and D/A External Reference.

The Configure System Hardware Window

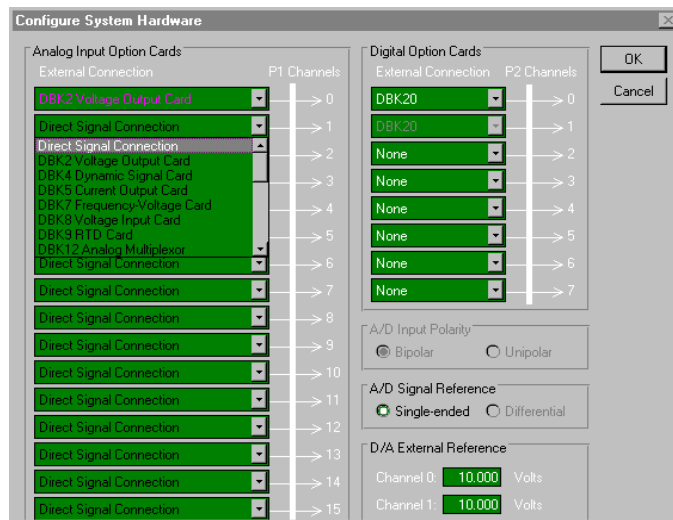
Select *Device* in the pull-down menu and then *Configure System Hardware* to set up the software for your hardware. DaqView needs this information to interpret the data and provide controls to access the hardware.

The Analog Input Option Cards section allows the user to assign DBK expansion cards to the 16 analog input channels. If no expansion cards are added, leave the default value (Direct Signal Connection) on all 16 channels. As expansion cards are added, this window must be updated.

Selecting a DBK expansion board will add channels in the main window's analog input spreadsheet. Channels associated with DBK expansion boards will also have more gains or modes to choose from in the analog input spreadsheet. If needed, a dialog box will appear and allow you to set sub-channel addresses or other parameters.

If using the DaqBook/100, DaqBook/112 or DaqBook/120, set the radio buttons for *Single-Ended* or *Differential* and *Bipolar* or *Unipolar* to match the settings of the corresponding hardware jumpers (the default hardware settings are *Single-Ended* and *Bipolar*). When *Differential* is selected, only 8 analog inputs are available. When analog expansion boards are connected, set the Daq for *Single-Ended*.

If using the DaqBook/100, DaqBook/200, DaqBoard/100A or DaqBoard/200A, the Digital Option card section allows you to configure the digital I/O hardware. If no DBK20 digital expansion boards are connected, three 8-bit I/O ports are accessible from the main window. Six additional 8-bit ports are made available in the main window for each DBK20 added. Make sure this window is kept updated for the actual hardware used.



Configure System Hardware Window

Help

The Help menu provides access to the on-line Help file. *About DaqView* gives you the current software version number.

Toolbar Items

Each item in the toolbar has its own icon and is also accessible from the pull-down menu. Placing the cursor on the icon and clicking the mouse button enables the tool or opens the corresponding window.

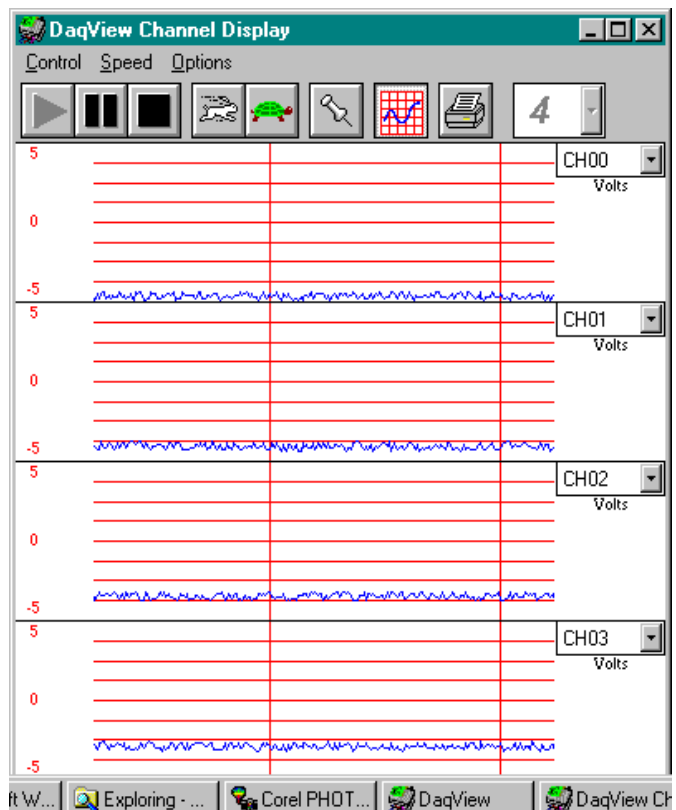
Make All Channels Active (or Inactive)

The first 2 toolbar buttons can turn all the channels ON or OFF at a single stroke. This feature is convenient during setup and troubleshooting or if only 1 or 2 channels must be set differently from the rest. Both these commands are also available from the Edit pull-down menu.

Charts

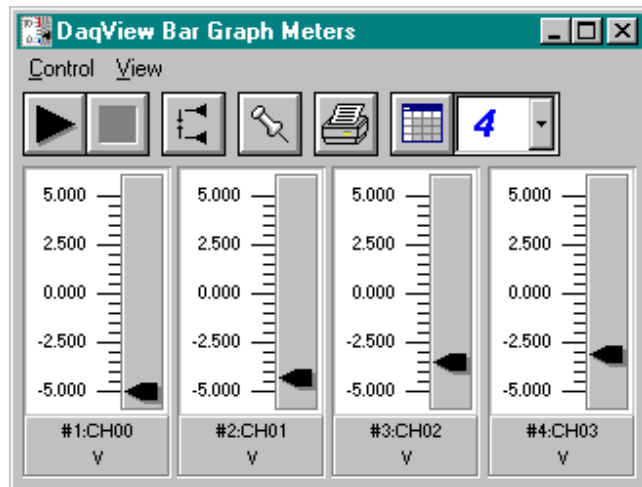
The third toolbar item displays real-time data in a strip-chart format for several channels on the DaqView Channel Display window. Charts are enabled by selecting the triangular “Play” button on the top left (see figure). Before “playing”, at least one chart must be assigned to an active channel through the drop-down list on the right side of the chart. For the selected channel, you can change the minimum and maximum values as needed. This command can also be enabled from the Window pull-down menu.

Data is read and displayed in the charts, meters, and Readings column as fast as the computer will allow. When an acquisition to disk has begun using the *Go* command, the charts, meters, and the Reading column take a lower priority, updating only when there is time in the acquisition-to-disk task. Thus, the data seen in the charts may not be the same as on the disk. As the scan rate is increased, the acquisition-to-disk task will take up more processor time and the charts will be unable to keep up.



Bar Graph Meters

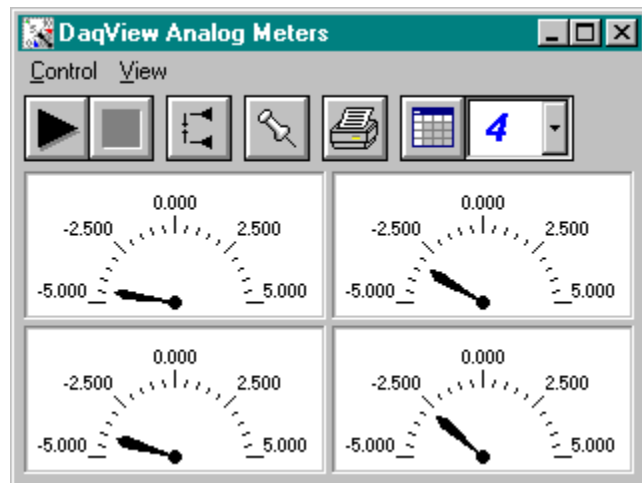
Selecting the Bar Graph Meter icon brings up the Bar Graph window to display several channels in bar graph format. To activate the display, select the Start button on the left side of the toolbar (or Start All Indicators in the pull-down menu or in the toolbar). You can vary the number of channels displayed by selecting the input box at the end of the toolbar. The Grid tool (next to last item on toolbar) is used to arrange the display for convenient reading. The pushpin icon in the center of the toolbar locks this window on top of other windows until you unlock it by selecting the pushpin again.



Bar Graph Window

Analog Meters

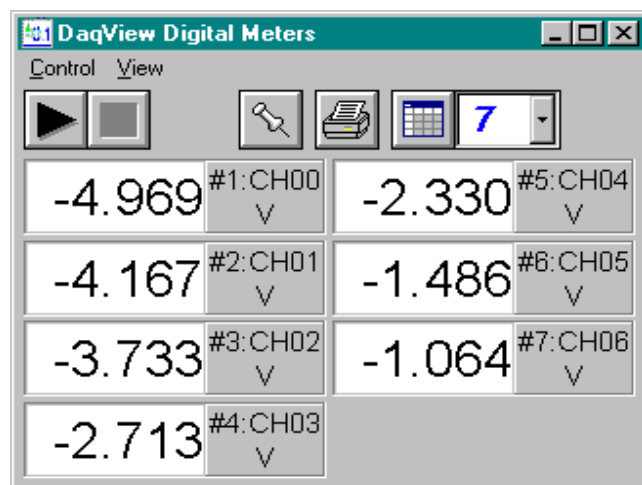
Selecting the Analog Meter icon brings up the Analog Meters window to display several channels in a dial/gage format. To activate the display, select the Start button on the left side of the toolbar (or Start All Indicators in the pull-down menu or in the toolbar). You can vary the number of channels displayed by selecting the input box at the end of the toolbar. The Grid tool (next to last item on toolbar) is used to arrange the display for convenient reading. The pushpin icon in the center of the toolbar locks this window on top of other windows until you unlock it by selecting the pushpin again.



Analog Meters Window

Digital Meters

Selecting the Digital Meters icon brings up the Digital Meters window to display several channels in numeric format. To activate the display, select the Start button on the left side of the toolbar (or Start All Indicators in the pull-down menu or in the toolbar). You can vary the number of channels displayed by selecting the input box at the end of the toolbar. The Grid tool (next to last item on toolbar) is used to arrange the display for convenient reading. The pushpin icon in the center of the toolbar locks this window on top of other windows until you unlock it by selecting the pushpin again.

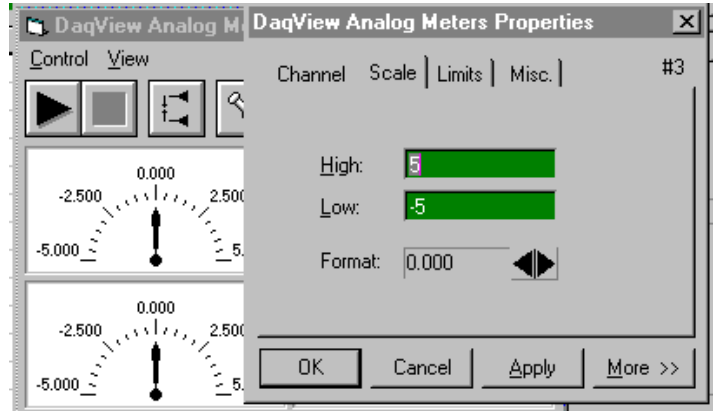


Digital Meters Window

Properties of Meter Windows

The meter windows simulate the look of 3 popular meter types: the bar graph, the analog dial, and the digital readout. Within each meter type, you can adjust their display properties to fit your needs. While in the meter window, **place the cursor in the display area and click the right mouse button**; then select Properties. A Properties window will appear and allow you to:

- Channels - select a new channel for display
- Scale - set the high and low points and the format (number of decimal places)
- Limits - set the high and low points and whether to display these limits
- Misc - set option to show the Trend Indicator and/or the Peak Hold Indicators.



Start (or Stop) All Indicators

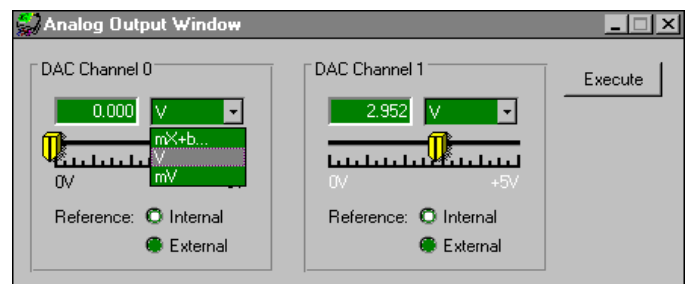
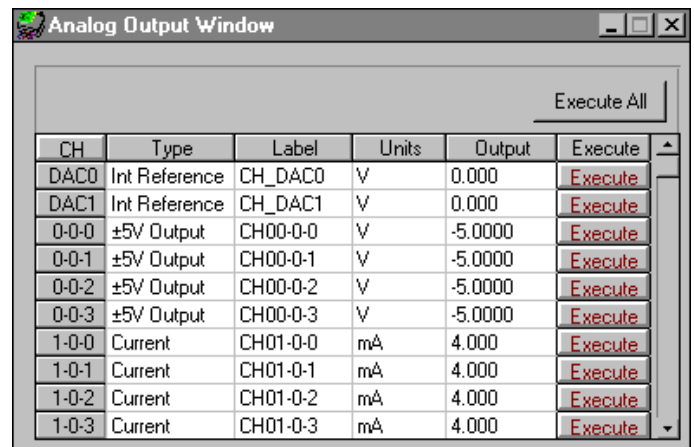
These buttons use the same icons as many CD players (right-pointing triangle for start; square for stop). To change some parameters, you may have to stop the data display. The Start and Stop All Indicator buttons affect several windows if open, including: Reading column, Charts, Bar Graph Meters, Analog Meters, and Digital Meters. **Note:** You can start or stop any of these active windows separately using their own Start, Stop (or Pause) buttons.

Analog Output Window

The analog output window provides interactive access to the two analog outputs on the DaqBook/DaqBoard. When set to the default internal reference, these outputs can be set from 0 to +5 VDC. An external reference can be connected ranging from -10 to +10 V allowing unipolar output from 0 to the inverse of the applied reference voltage.

The output voltage can be changed by moving the slide control or placing the cursor in the numeric field and entering a valid voltage value. Selecting the *Execute* button sends the voltage values to the outputs.

Two versions of this window exist. If a DBK2 or DBK5 are active in the Analog Input Spreadsheet, the window will accommodate voltage and current output in addition to the DAC reference. **Note: This window is not available when the arbitrary waveform window is visible.**

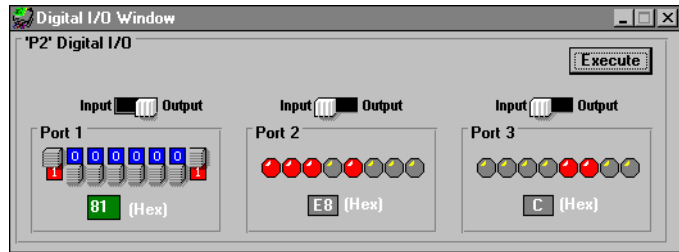


Analog Output Window (with and without DBK2 & DBK5)

Digital I/O Window

For DaqBook/100, DaqBook/200, DaqBoard/100A and DaqBoard/200A Only

The digital I/O window is displayed when digital I/O button on the main window is selected. This window provides interactive control of all configured digital I/O ports (as configured in the Hardware Setup Window). The base unit has three 8-bit ports that can be configured as either inputs or outputs. Up to twenty 8-bit ports can be accessed when expansion boards are added. When the *Execute* button is pressed, all ports configured as outputs will be updated and all input ports will be read.



Digital I/O Window

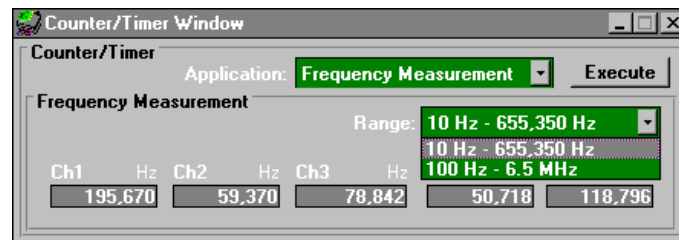
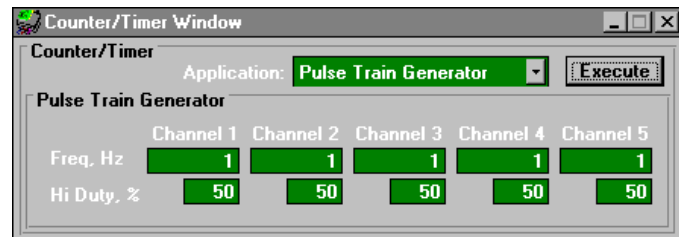
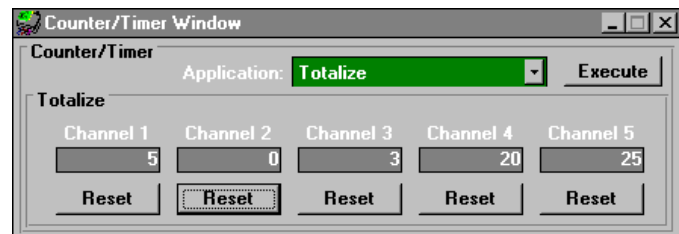
Counter/Timer Window

(For DaqBook/100, DaqBook/200, DaqBoard/100A or DaqBoard/200A Only)

The counter/timer window is displayed when the counter/timer (the red C/T button) is selected from the toolbar. Selecting the *Execute* button begins the selected function. When the mode is selected via the drop-down list, the controls change accordingly (see figure). Three C/T modes are available:

- Totals - counts signals from the related pin on P3. Each channel has its own Reset button.
- Pulse train generation - frequency and the high (+5 VDC) percentage of a cycle can be set for 5 channels.
- Frequency measurement - 2 ranges are available.

As used by DaqView, the 9513 chip for C/T functions has all 5 channels assigned to 1 of 3 modes. **Note:** full function (using separate channels and modes) of the 9513 is possible via API programming (refer to the daqCtr* commands in the *Programmer's Manual*).



3 Modes of the Counter/Timer

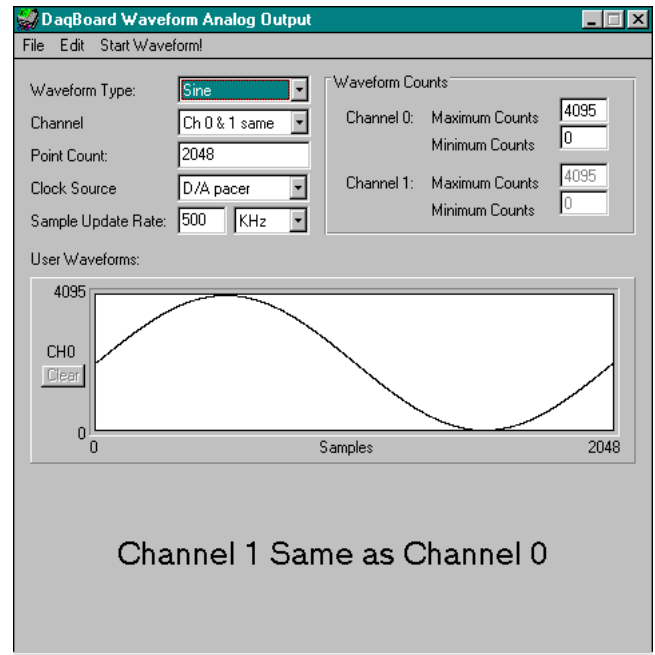
Arbitrary Waveform Window

Unlike the DaqBook, the DaqBoard has a waveform mode for its two analog output channels. DaqView allows either or both DaqBoard DACs to be configured in interactive mode or waveform mode.

With a DaqBoard, both the *Analog Output* and the *Arbitrary Waveform* are enabled. Selecting *Analog Output* will pop-up the standard analog output window for both DaqBoard and DaqBook (see previous section). Selecting *Arbitrary Waveform* will pop-up the arbitrary waveform window for one or both analog outputs.

Note: This window is not available when the analog output window is visible.

In the Waveform box, you can select a standard function generator waveform (sine, square, triangle, sawtooth) or a freehand drawing. In Freehand, move the mouse to the waveform window and draw a waveform using the left mouse button.



Arbitrary Waveform Window

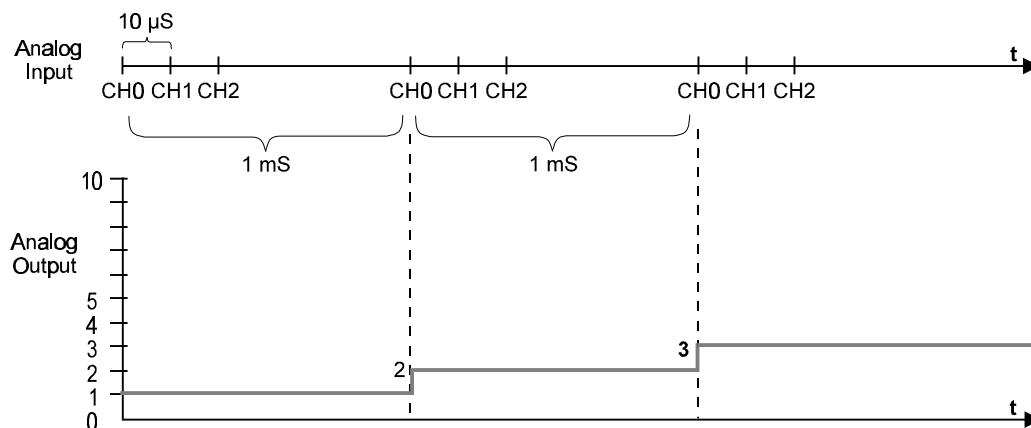
In the Channel Selection box, you can designate the channel for the waveform output. The DaqBoard contains a waveform buffer with 4096 locations. If the two DACs are programmed to generate two different waveforms, the buffer is split into two 2048-location buffers.

The Point Count field holds the number of points desired for the waveform. Using only one DAC (or both DACs outputting the same waveform), a maximum of 4096 points are available. If two DACs are outputting different waveforms, a maximum of 2048 points are available for each.

The Frequency Setup section sets up the waveform's play rate. Two clock sources are available:

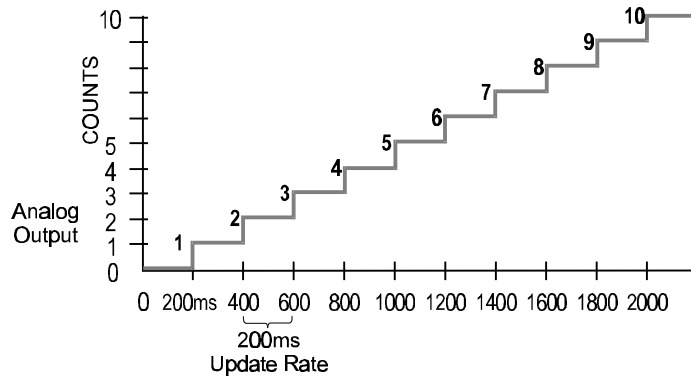
- D/A pacer with a rate set in the Sample Update Rate field. When using the D/A pacer, the Sample Update Rate field controls the speed at which the DAC is updated.
- A/D pacer, the clock used by the analog input section of the DaqBoard. Using the A/D pacer clock synchronizes the update of the DAC output with the analog input data collection.

The following example uses the A/D pacer clock as the update source. The DAC waveform is linearly ascending voltage starting at count 1 and increasing by 1 count on every update. The analog input frequency is set to 1 kHz.



Synchronization of D/A and A/D

The following example shows a DAC waveform with the clock source set to D/A pacer and the Sample Update Rate field set to 5 kHz.



DAC Waveform

The Waveform Attributes section contains fields for setting the max and min for the Y-axis. The DaqBoard's DACs are 12-bit devices, providing 4096 discrete voltage outputs. If the default, internal voltage reference is applied, the count of 4096 represents 5 V and 0 represents 0 V. The following equation shows the relationship between counts and volts:

$$V_{out} = \text{Count} * (V_{ref} / 4096)$$

Under the File menu item, Load and Save Waveform commands allow you to save the displayed waveform(s) to disk in an ASCII format. The files created are compatible with spreadsheets and word processors, allowing you to numerically inspect and/or alter the saved waveforms. The Load Waveform command reads any ASCII file of numbers up to the number of points specified in the Point Count field. The table shows the file format.

Sample Waveform File Formats	
1-channel:	2-channel:
2043<CR>	2432<TAB>293<CR>
1019<CR>	394<TAB>345<CR>
300<CR>	2934<TAB>3456<CR>
923<CR>	743<TAB>875 <CR>
Note: In the two-channel example, the first column is channel 1, and the second column (after the TAB) is channel 2.	

Under the Edit menu, Copy and Paste use the system clipboard to post and receive waveform data. The data formats are identical to the Save and Load operations. Copy and Paste are recommended for use with spreadsheets to numerically inspect and/or modify waveforms. To start the waveforms playing on the DACs, click the *Start Waveform!* item in the menu bar. **Note:** When the analog input section is set for analog input triggering, DAC1 is used internally to supply the desired threshold voltage. In this case, DAC1 is not available for waveform output.

Enable Input Reading Column

The next to last toolbar item enables/disables the Reading column of the Analog Input Spreadsheet to provide a numeric view of incoming data. This function toggles on and off when the button is repeatedly selected. Some windows require the Reading column to be disabled while changing channels or other parameters. This command is also available from the Acquire pull-down menu.

Go

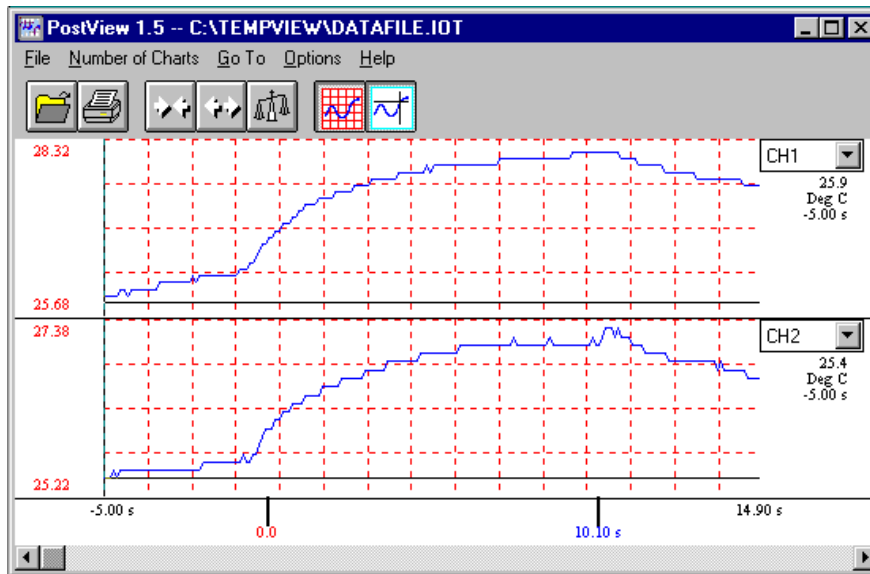
The Go selection, the last icon on the toolbar, activates an acquisition of data to file. The *Event* option under the *Trigger Setup* portion of the Main window determines when the Go process is initiated. The *Event* selections are detailed in the *Acquisition Configuration* section of this chapter. After Go is selected, the process is automatic beginning with the *DaqView Armed* screen. This screen posts the *Trigger Armed* time as well as the *Trigger Event* information. The Go process makes use of the *Data File Setup* parameters to format data collection.

Once the *Trigger Event* occurs, the *DaqView Triggered* screen appears. This screen allows you to witness the data sampling parameters you set in the Main window prior to initiating the Go process as well as the progress of the data acquisition. If file conversion is selected, the Go process is concluded by converting the generated .BIN file from binary to ASCII data. The resulting file is saved under a user-specified directory and file name or as a default under the DaqView subdirectory as DAQV.TXT. This .TXT file is available for data processing or analysis by various software packages such as DaqViewXL.

PostView 1.5

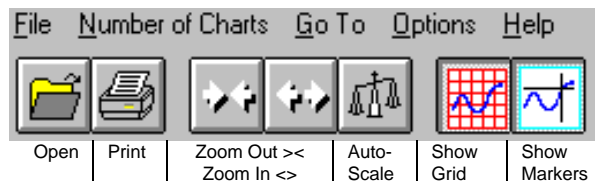


PostView is an independent program that allows you to view waveforms recorded by DaqView, TempView, and WaveView. As the data file is being created, a descriptor file used by PostView is also created.



The program can be started from a toolbar PostView icon, or from a pull-down menu. PostView can also be started independent of DaqView, TempView, and WaveView. Multiple sessions of PostView can be invoked concurrently to view several data files. To view a data file from within PostView, select *Open* under the File menu. When PostView is started from DaqView, TempView, or WaveView, it automatically opens the selected destination file. To view other files, use *Open* under the File menu for the applicable program's data files. To place channel waveforms into the window, select the number of charts from 1 to 16 under the *Number of Charts* menu item. Selecting *N* number of charts will automatically place the first *N* channels in the charts. Use the *Channel Select List Box* (Upper right corner of each chart) to view the desired channel. The Channel Select List Boxes contain labels that were assigned to the recorded channels by DaqView, TempView, or WaveView.

The next table explains PostView toolbar items and is followed by a description of PostView Menu items. Menu and toolbar items are identified by the following figure.



PostView Toolbar Items	
Open	Accesses the Open Data File window.
Print	Sends the PostView chart(s) to an assigned printer.
Zoom Out (><)	The Zoom Out button doubles the visible timebase, showing more of the waveform. For example, if 10 seconds of information is visible, clicking the Zoom Out button will show 20 seconds.
Zoom In (<>)	The Zoom In button halves the visible timebase, showing less of the waveform. For example, if 10 seconds of information is visible, clicking the Zoom In button will show 5 seconds.
Autoscale	Clicking the Auto Scale button adjusts the Y-axis labels so that the visible waveform fills 90% of the chart's range.
Y-axis Adjust	The Y-axis Adjust fields show the chart's minimum and maximum for visible charts in the engineering units shown. Clicking the Auto Scale button automatically adjusts the Y-axis Adjust fields. To adjust any chart's minimum or maximum, place the cursor in the desired Y-axis Adjust field, and type in a new value.
Show Grid	Places a grid on the chart (s), or removes the grid if already present.
Show Markers	Each chart contains a cross-hair marker that shows the numerical values of time and magnitude at its present location in the waveform. The Markers start out at the far left of every chart, showing the time and magnitude of the first visible point. <ul style="list-style-type: none"> • <u>Left mouse button</u> allows the user to drag the marker of each chart <i>independently</i>. • <u>Right mouse button</u> moves the markers from <u>all</u> the charts <i>in unison</i>. The Options menu contains a function which allows you to turn markers on and off. When a check appears in front of this item, its associated indicator is on or visible. Selecting the menu item toggles the indicator (and the check mark) on and off.
Trigger Event Marker	The Trigger Event Marker on the time axis shows the location of the trigger point.
Stop Event Marker	The Stop Event Marker on the time axis shows the location of the stop point.
The Scroll Bar at the bottom of the PostView window (see figure on previous page) allows the waveforms to be scrolled right or left in two ways: <ol style="list-style-type: none"> 1. When clicked on, the small left and right arrow boxes scroll the waveforms approximately 20%. 2. The plain scroll button shows the relative location of the visible region of the waveforms and can be dragged along the scroll bar to any location desired. 	

PostView Menu Items

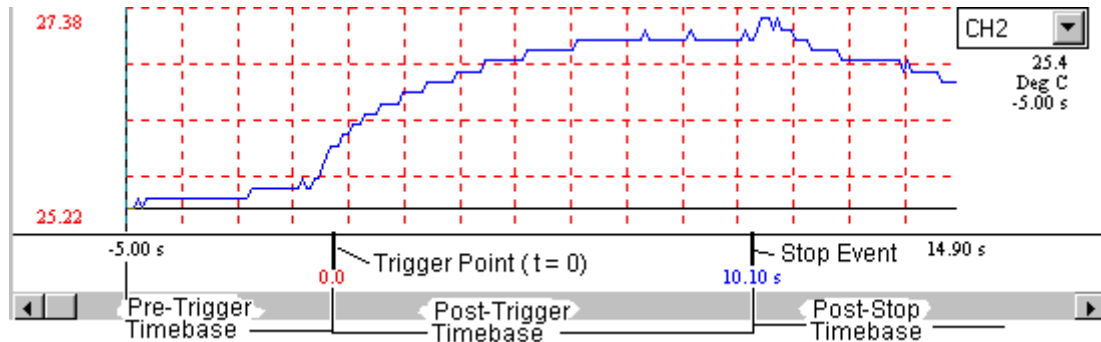
The following tables pertain to functions allowed by the pull-down menus.

<u>F</u>ile	
<u>O</u> pen	Opens a data file created by DaqView, TempView, or WaveView. PostView automatically detects whether the file contains ASCII or binary data.
<u>P</u> rint Window	Prints the present PostView window.
<u>E</u> xit	Exits the File menu.
<u>N</u>umber of Charts	
1-16	After a data file has been opened, the number of desired charts can be selected. You can also use this menu selection to change the number of charts displayed.
<u>G</u>o To	
<u>P</u> ercentage	Automatically scrolls to the desired percent of the data file. For example, selecting 50% would display a waveform segment from the middle of the data file.
<u>S</u> can Number	Automatically scrolls the waveforms so the desired scan number is in view. This menu selection invokes a dialog box which displays the number of scans in the file.
<u>T</u> ime	Automatically scrolls the waveforms so the desired time is in view.
<u>T</u> rigger Point	Automatically scrolls the waveforms so the trigger point (t = 0) is in view.
<u>O</u>ptions	
<u>Z</u> oom <u>O</u> ut	Allows more of a chart(s) to be seen by decreasing the dimensions.
<u>Z</u> oom <u>I</u> n	Zooms in on a chart(s), providing more detail to a smaller area of the chart(s).
<u>A</u> utoscale	Automatically generates a scale, in contrast to manually assigning the scale.
Show Grid (Ctrl+G)	Allows grids to be turned off and on for all visible graphs. When a check appears in front of an item, its indicator is on or visible.
Show Markers (Ctrl+K)	Allows markers to be turned off and on for all visible graphs. When a check appears in front of an item, its indicator is on or visible.

Help	
Contents	The initial PostView help screen provides an overview and listing of the help file contents. A single topic can be selected for quick access to help information.
Search	Type a word or select one from the Show Topics list for quick access to help information.
Help on Help	Provides instructions on how to use a Windows Help system.
About	Provides the PostView version number, for example: Version 1.5.

The PostView Timebase

PostView automatically detects the timebase of the data file and shows the time in seconds in the X-axis labels.



Note: Post -Stop Timebase does not apply to DaqView or WaveView.

PostView can create files containing multiple timebases, pre-trigger and post-trigger data for DaqView, TempView and WaveView programs. In addition, with TempView you can create files which include post-stop data.

The following comments apply to DaqView, TempView and WaveView, except when noted otherwise.

- Note that the trigger point ($t = 0$) is not necessarily the first point in the data file.
- If pre-trigger scans are available, they will be shown prior to the trigger point and labeled with negative time numbers.
- If a dual timebase was used when collecting the data, PostView will show a discontinuity in the time axis when the timebase changes. When a dual timebase is used, the post-trigger scans are collected at one frequency while the pre-trigger and post-stop scans are collected at another. **Note: DaqView does not support dual timebase.**
- For **TempView**, if post-stop data is available (scans collected after the Stop Event), they are displayed after the Stop Event marker.
Note: This feature does not apply to DaqView and WaveView.

Data File Accessibility

Depending on the source program, file accessibility is as follows:

- **DaqView** and **WaveView** - PostView can access these program files only after the entire acquisition is complete.
- **TempView** - PostView can access the program files at any time, even during the acquisition. If PostView reaches the end of a file while TempView is still collecting data, PostView will automatically display the new data as it becomes available.

Note: For the fastest display of waveforms, select binary data storage. This applies to DaqView, TempView and WaveView.

DaqViewXL

DaqViewXL is a software component add-in that installs into Microsoft Excel and provides complete setup and data acquisition within Excel under Windows 3.x or Windows 95. Acquired data is immediately placed in an active spreadsheet, updating cells and graphics. Data is then analyzed and graphically displayed.

DaqViewXL:

- Augments Microsoft's Excel spreadsheet software with data acquisition capability
- Provides strip-chart graphics for real-time data display
- Automatically converts data to engineering units

Once DaqViewXL is installed within Excel, the spreadsheet provides a floating toolbar that contains all the data acquisition controls. This DaqViewXL tool bar (also accessible from Excel's menu) can enable all configuration and data acquisition tasks. The features of Excel and DaqView are seamlessly combined to form a powerful data acquisition solution.

DaqViewXL performs like DaqView despite a slightly different user-interface. To set up your data acquisition system, you just click on the Configure button in the Excel tool bar. DaqViewXL's main window appears with all of the controls required to configure data acquisition including setting up input and output channels on an easy-to-use grid. Each column of the channel-configuration grid corresponds to specific parameters in the data acquisition system and allows you to set these parameters on a per-channel basis.

Excel provides a wealth of functions such as entering and editing data, building formulas to calculate values, and creating graphics. Excel offers high-powered analysis functions including frequency domain functions such as fast Fourier transforms (FFTs). The newest version, 5.0, includes additional features for retrieving and analyzing data. Such features make it easy to set up a spreadsheet for processing raw data, generating graphs, and printing reports. Excel is a perfect tool for repetitive tests where consistency is important.

You can obtain automated reports by taking advantage of the Software Components concept. For example, you can embed an Excel spreadsheet object into a Word document. When the spreadsheet object is activated, it gives you access to DaqViewXL. In this case, the Word document holds all text, as well as an embedded Excel spreadsheet object that holds raw and calculated data and graphics. Double-clicking on the Excel object in Word can access DaqViewXL to efficiently prepare a statistical report. This compound document lets you collect, configure, analyze, graph, and annotate data with a few clicks of a mouse button.

Program Requirements

DaqViewXL requires the following software:

- IOtech DaqBook/DaqBoard Series driver version 4.2 or later. (Installed components must include the DaqView Windows utility program.)
- Microsoft Excel, version 5 or 7
- Microsoft Windows Operating System (Windows 3.1x, Windows for Workgroups 3.1x, Windows 95)

Installation of DaqViewXL Software

While running Windows (3.X, NT, or 95) insert DaqViewXL disk1 into the disk drive. From the *Run* dialog box initiate the software installation by running the setup.exe file. Follow the on screen directions (dialog boxes) to finish the software installation.

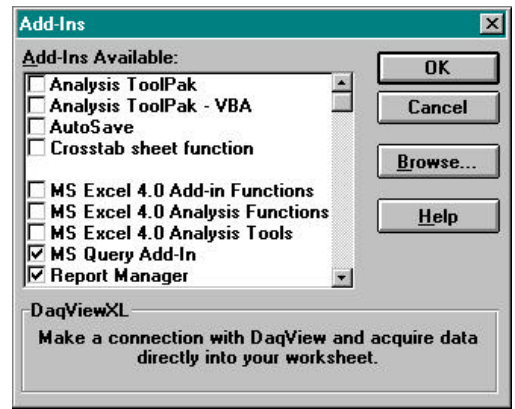
Note: If you are installing a new version of DaqViewXL over a previous version, you should perform the steps in the section (Removing DaqViewXL from Excel) before installing the new version.

Running DaqViewXL for the First Time

DaqViewXL is an Excel *Add-In* that must be loaded into the Excel environment before it can be used. Perform the following steps to install DaqViewXL:

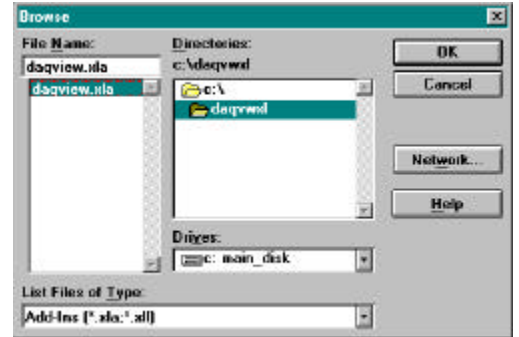
1. Start Excel (refer to Microsoft documentation as needed).

2. Select the "Tools" menu and click on "Add-Ins...". This will open the "Add-Ins" dialog box, as shown to the right.



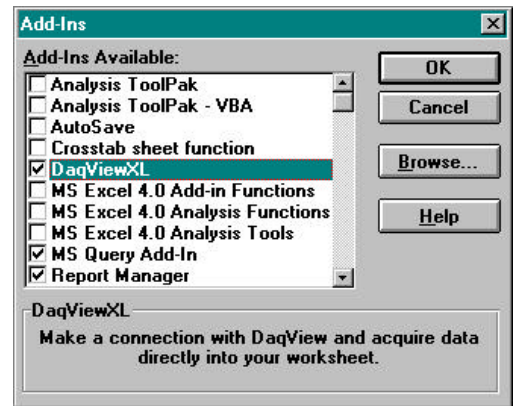
Excel Add-Ins Dialog Box

3. Click the "Browse" button and locate the file DAQVIEW.XLA. It should be in the install directory specified in the setup program. Click OK.



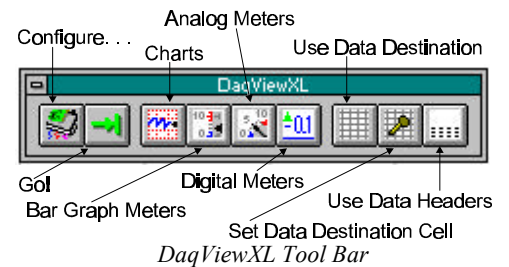
Browse Option Dialog Box

4. DaqViewXL should now be in the "Add-Ins Available" list box. Make sure there is a check next to it and click OK.



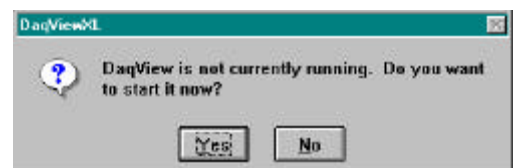
Add-Ins Dialog Box (post addition)

5. Excel loads the add-in which creates a Toolbar and adds a menu item to the "Tools" menu. Click on the "Tools" selection to see the new DaqViewXL menu item.



DaqViewXL Tool Bar

6. After DaqViewXL is loaded in this manner, it will automatically load whenever you run Excel. If DaqView is not running at the time you launch Excel (with DaqViewXL activated), the following dialog box will appear.



DaqView Start Up Confirmation

Removing DaqViewXL from Excel

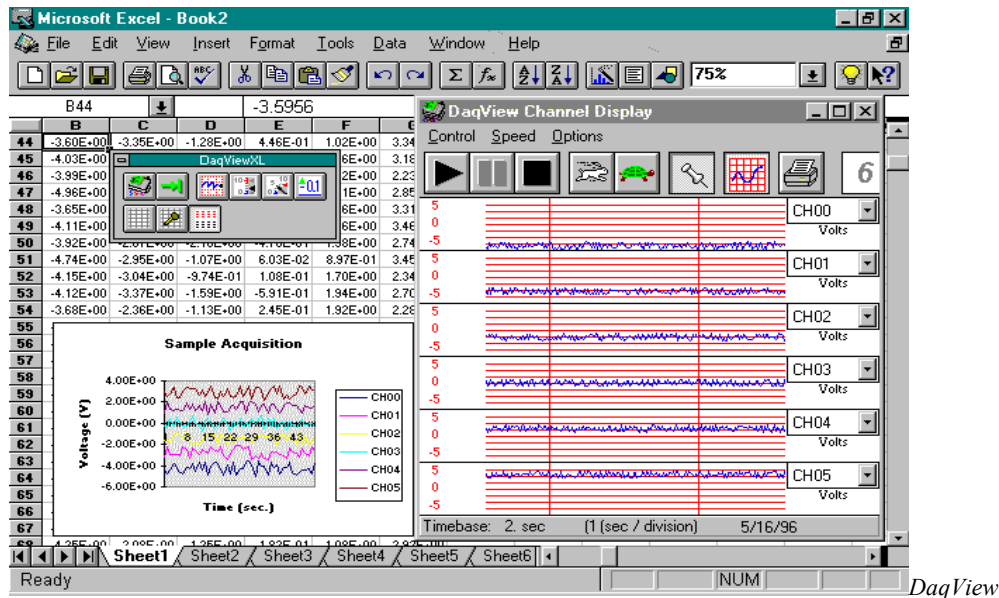
If you do not want DaqViewXL to load each time you run Excel, perform the following steps:

1. Start Microsoft Excel.
2. Select the "Tools" menu and click on "Add-Ins...". This will open the "Add-Ins" dialog box.
3. Find the DaqViewXL line in the "Add-Ins Available" list box and click on it. This should remove the checkmark next to the name. Click OK.

DaqViewXL will no longer auto-load with Excel. The next time you want to use DaqViewXL repeat the above steps, this time checking the DaqViewXL line instead of unchecking it. DaqViewXL will then load and once again become part of the Excel environment.

Basic Function of DaqViewXL

DaqViewXL performs much like the standard DaqView. With Excel's macro capabilities, you can tailor DaqViewXL to meet various requirements. Your knowledge of Excel will help you use all its tools for better data acquisition and processing. The following sections give you the basics of how DaqViewXL can organize and analyze your data. These basic ideas will give you the ability to explore all DaqViewXL has to offer. The discussion mirrors previous sections and highlights the differences between DaqViewXL and DaqView. The *Hints and Tips* section gives you tools and suggestions to improve your productivity with DaqViewXL.

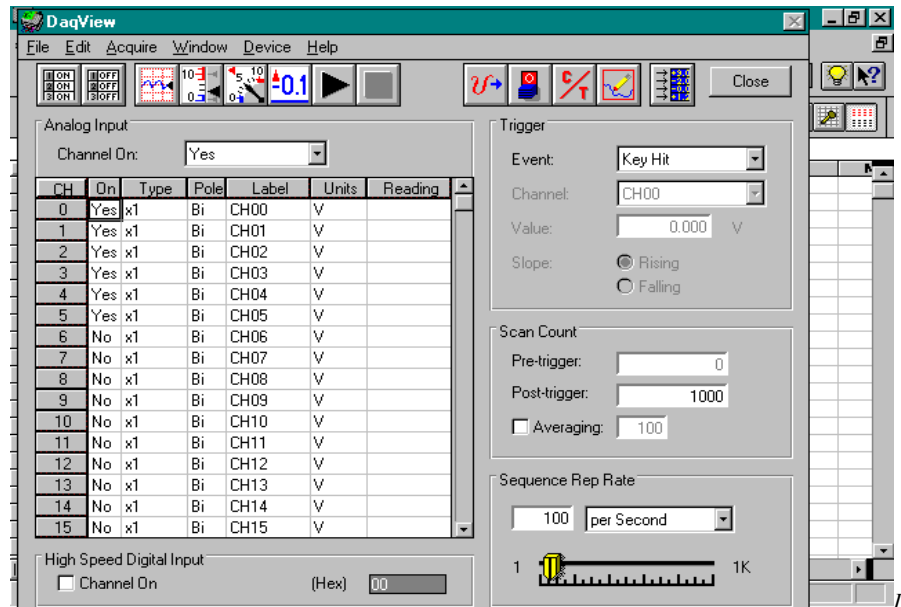


XL Toolbar and Displays

DaqView

Configuring an Acquisition

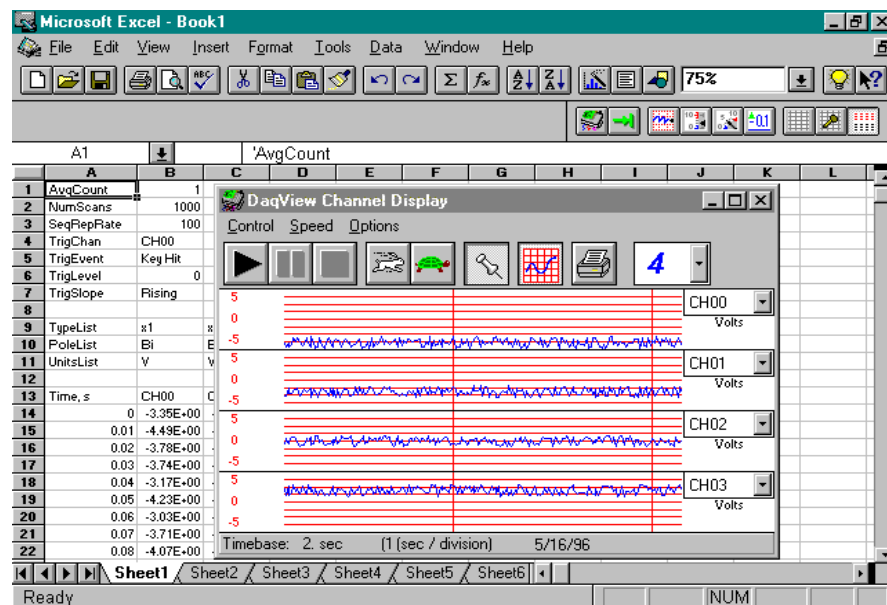
After installing DaqViewXL and entering Excel, click the *Configure...* icon (first in the DaqViewXL toolbar) to set up your system. This launches DaqViewXL's main screen (see following figure). This screen is slightly different from the stand-alone version shown at the beginning of the chapter. (Specifically, DaqViewXL does not have a PostView button—Excel performs those functions better. Also, the Go button is now the 2nd button on the toolbar window rather than the main screen.) The earlier discussion of DaqView configuration still applies.



aqView Screen Within Excel

Real-Time Charting

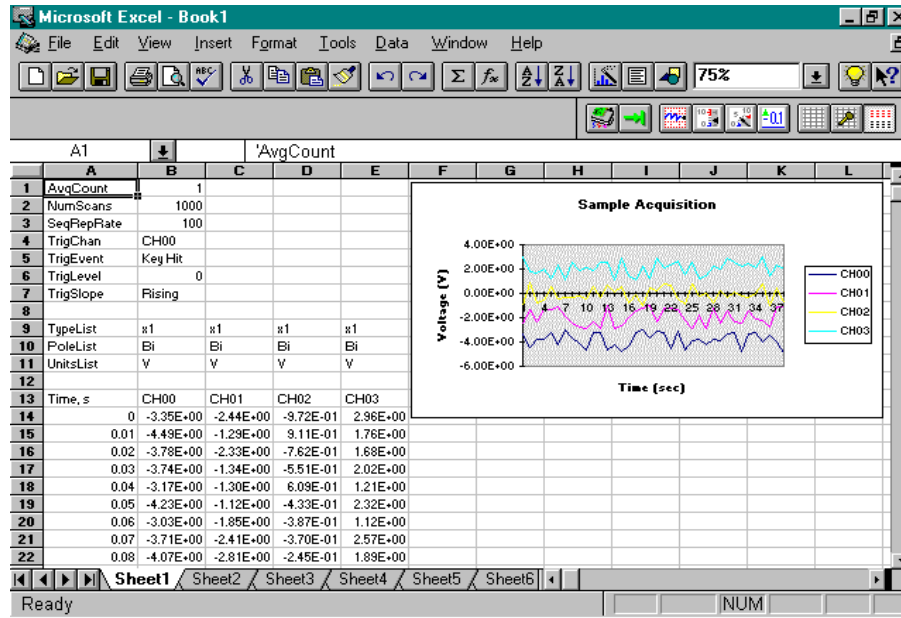
DaqViewXL provides a real-time strip-chart window (same as in DaqView) for graphical presentation of data. You can select up to 16 channels of data assigning a channel to each graph. The strip-chart representation is continuous while data is being collected at up to 100 kHz. All 16 charts are synchronized and have an adjustable scrolling speed that can be set independent of the sampling rate. Selecting the *Charts* icon in the DaqViewXL toolbar will activate the following window. The chart selection must be made prior to initiating an acquisition (selecting the *Go!* icon).



DaqViewXL Real-Time Charting

Data Header

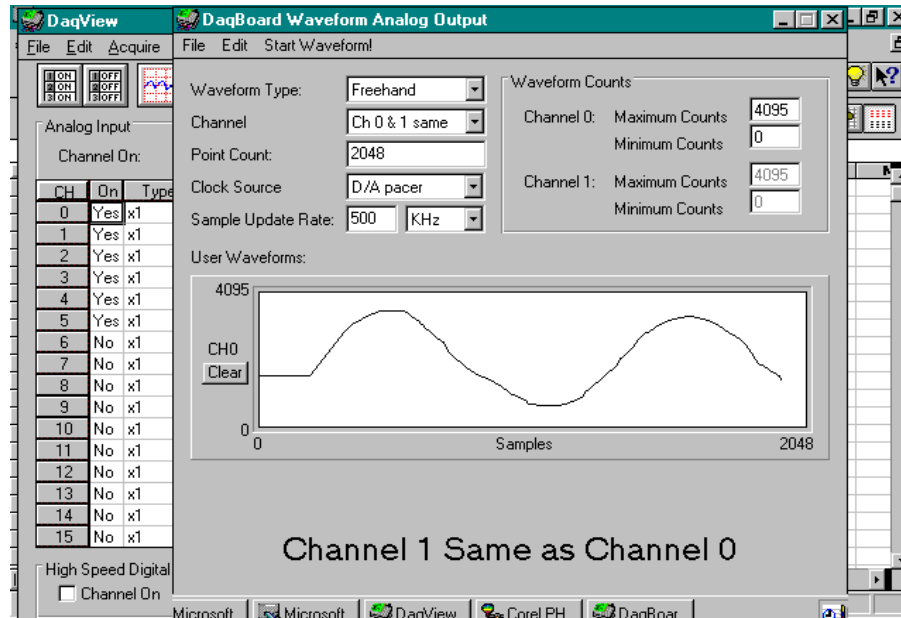
DaqViewXL's optional data header supplies the global acquisition parameters and the configuration of each channel and places this information directly in the spreadsheet along with acquired data. This information includes channel gain or thermocouple type, bipolar/unipolar setting, units, and channel label. To activate Data Headers select the *Use Data Headers* icon in the DaqViewXL toolbar.



DaqViewXL Data Header

Analog Output

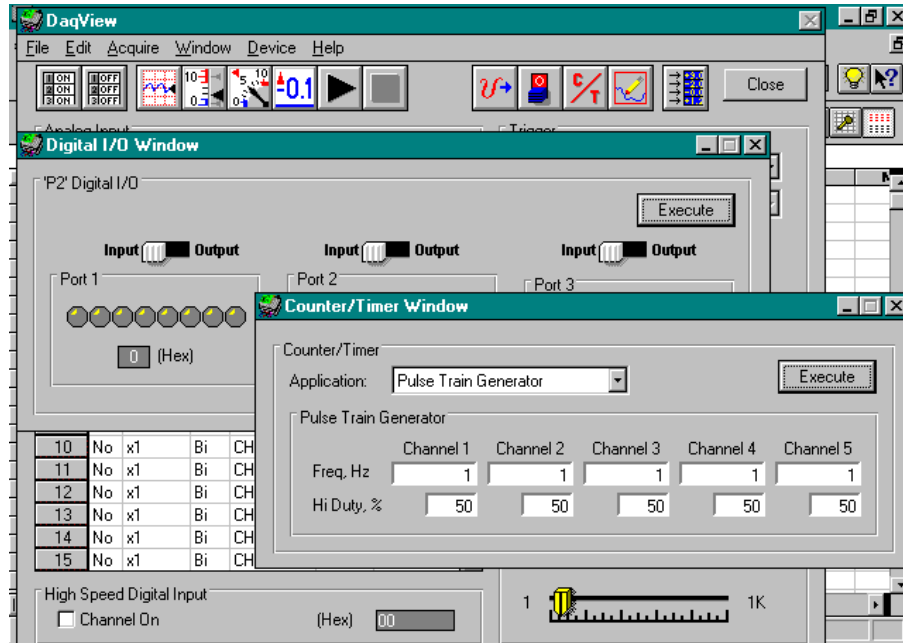
DaqViewXL provides an analog output window to set the output voltage from the D/A converters. The window features a slider and a text-entry field for each D/A converter. With DaqBoards, you can manually draw a waveform using your mouse and output that waveform at up to 500 kHz per channel via the DaqBoard's two 12-bit D/A converters. Once the DaqView screen is open, select the *Arbitrary Waveform* icon from the DaqView toolbar. The DaqView options previously discussed still apply.



DaqViewXL Static Voltages & Arbitrary Waveforms

Digital I/O and Counter-Timers

DaqView's digital I/O and counter-timer windows provide you with full interactive control of the hardware's on-board digital I/O and all digital expansion channels (up to 192). This window allows you to configure each port as either an input or output. The *Counter/Timer* and *Digital I/O* icons are also selectable from the DaqView toolbar. The counter-timers provide a pulse-train generator, which allows you to specify the frequency and duty-cycle for digital output; a totalizer function, which counts pulses; and a frequency measurement function.



DaqViewXL Digital I/O & Counter-Timers

Hardware Support

DaqView XL supports all models of the Daq Family including all DBK expansion modules and signal-conditioning cards. To add DBK options to your system, you must select the desired signal conditioning in the DaqViewXL Hardware Configuration window. Once configured, DaqViewXL automatically provides access to new channels from within Excel. The channel configuration grid provides a means of configuring each channel.

DaqViewXL also provides the engineering unit output required by each DBK option. For example, data from a DBK current card is returned in amps, while data from a thermocouple or RTD card is returned in degrees. When you see the mx+b feature in DaqViewXL's units column, the collected data for the associated channel is automatically converted to custom units and then scaled and offset before appearing in the spreadsheet. The mx+b conversion features is especially handy for transducers such as strain gages because it converts the return voltage value to units such as pounds or units of pressure. Depending on DBK used, DaqViewXL accommodates special gains and other capabilities.

Hints and Tips for DaqViewXL

- Do not run DaqView first and then try to use DaqViewXL. When DaqViewXL runs DaqView it puts it into a special "server" mode which supports transactions with "client" applications such as Excel. Normally DaqView does not run in this mode and cannot support DaqViewXL.
- When you acquire data into a worksheet, some of the data may be displayed as a string of hash marks (e.g. #####). Excel will do this whenever the column is not wide enough to display all the digits of a number. To eliminate the hash marks, just widen the column. An easy way to do this is to use the "Format, Column, Auto Fit Selection" menu command. Immediately after an acquisition, all the rows and columns of data are selected. Click this menu item to automatically fit all the columns to the new data.
- DaqViewXL does not have to be used as an Excel Add-In; it can also be loaded like a workbook. To do so, open DAQVIEW.XLA using the File Open dialog box. The DaqViewXL toolbar and menu items will be added to Excel and behave exactly as they would if DaqViewXL had been loaded as an Add-In. You can

also load DaqViewXL via the Icon created by the setup program. When you do it this way, Excel does not automatically create a blank workbook for you. After Excel loads with DaqViewXL, access the File, New menu item to create one. When you load DaqViewXL like a workbook, it will not automatically load the next time you run Excel. However, it will appear in the recently-used list at the bottom of the File menu, making it easy to load the next time you run Excel.

- You can start an acquisition from an Excel macro as shown in the following VBA macro code example:

```
Sub Macro1()  
Application.Run Macro:="menuGoDoIt"  
End Sub
```

This is equivalent to clicking the Go! button on the DaqViewXL toolbar. It arms the hardware for an acquisition, and if DaqView is set up to trigger on "External TTL" or "Channel Value", the acquisition will take place automatically when the trigger condition is met. For acquisitions configured to trigger on a "Key Hit", the acquisition will not start until the user presses the "Manual Trigger" button on the "DaqView Armed" dialog box.

To automatically press this button after arming DaqView use the following macro:

```
Sub ArmAndKeyHit()  
Application.Run Macro:="menuGoDoIt"      'presses the Go! button  
SendKeys "{ENTER}"                      'presses the space bar  
End Sub
```

This macro will start a "Key Hit" acquisition immediately. (Note: Macro1, shown above, was generated by Excel's macro recorder).

- If you chose to customize DaqViewXL's toolbar, you should avoid removing buttons. If DaqViewXL detects missing buttons it will prompt you to allow it to remove the toolbar and recreate it with all buttons. You can decline to do so, but DaqViewXL will continue to ask you to recreate the toolbar each time it starts-up. You can add buttons, rearrange them, and re-size the toolbar without getting this prompt.
- If DaqView cannot connect to your hardware on start-up, it will automatically switch to the Simulated Instrument mode when loaded from DaqViewXL.

This and the following chapters are written for users who wish to write their own programs to control and acquire data from the TempBook/66. This introductory chapter covers basic TempBook operation from a programmer's perspective and the options available for API drivers and languages. Further detail is included through examples in the individual language support chapters, the command reference section, as well as the chapters on thermocouple linearization and software calibration & zero compensation.

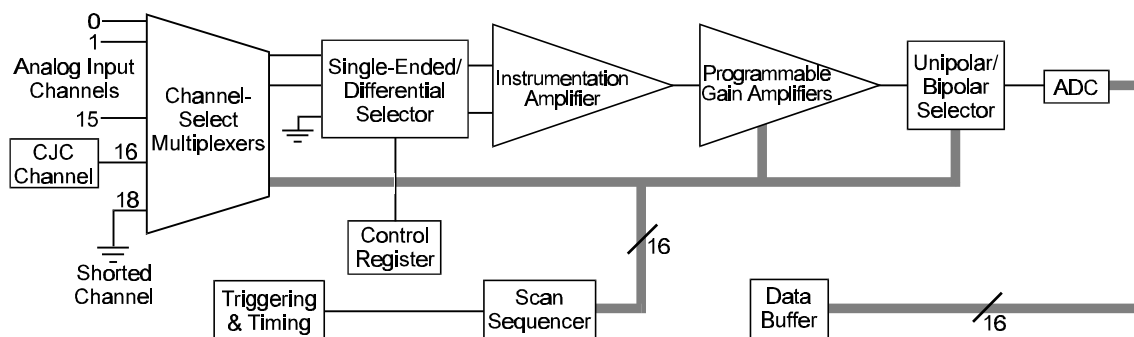
A Programmer's View of TempBook Operations

The TempBook provides flexible, high-speed, multi-channel data acquisition capabilities through the use of sophisticated analog and digital electronic circuitry. This circuitry allows up to 16 analog input channels to be read at an aggregate 100 kHz sampling frequency. Each of these channels can be read as a unipolar (0 to +V) or bipolar ($\pm V$) signal in one of 8 input voltage ranges. These ranges are determined by dividing the standard 0 to 10V unipolar or $\pm 5V$ bipolar input range by the available gains of $\times 1, 2, 5, 10, 20, 50, 100,$ or 200. Additionally, each input can be read as a single-ended or differential signal (selecting differential limits the number of available channels to 8).

The conversion of these input signals to a digital code is accomplished by a high-performance 12-bit analog-to-digital converter (ADC). (The resultant digital code can then be converted into a voltage via a DAC). The output of the ADC is an unsigned value which is left-justified within a 16-bit data word. In unipolar and bipolar modes, the output code is related to the input voltage as shown in the table (12-bit data format and standard input range shown).

Bipolar	Unipolar
0x000 = -5V	0x000 = 0V
0x800 = 0V	0x800 = 5V
0xFFF = 4.9976V	0xFFF = 9.9976V

The analog section of the TempBook consists of the following: channel select multiplexers, a single-ended / differential selector, an instrumentation amplifier, programmable gain amplifiers, a unipolar / bipolar selector, and the 12-bit ADC.



Block Diagram

The analog electronics are controlled by a scan-sequencer and control registers. The scan sequencer is implemented with a 16-bit \times 512-location FIFO RAM. Each entry in the scan sequencer contains channel, gain, and polarity information. The single-ended/differential selection is controlled by a static control register entry. Once triggered, the scan sequencer is stepped through at a constant rate of 100 kHz until all sequencer entries have been read. At each step, an ADC conversion is performed and the resultant output is stored in a data buffer implemented with another 16-bit \times 512-location FIFO RAM.

A scan is initiated by a trigger from a software command, a TTL input, or an internal pacer clock. In addition to the source, a mode (one-shot or continuous-trigger) can be selected.

- In **one-shot mode**, the scan sequencer is stepped through once (scanned) each time a trigger is received.
- In **continuous mode**, the scan sequencer waits for the selected trigger for the initial scan but subsequent scans are initiated by the pacer clock.

Note: the selection of one-shot or continuous mode has no effect if the pacer clock is selected as the trigger source.

The pacer clock is generated by dividing an internal 1 MHz or 100 kHz clock by a programmable 32-bit counter. The pacer clock source selection is made by an internal jumper setting, and the counter is implemented by cascading P1 and P2 of an 8254 counter/timer chip.

The scan sequencer must be loaded prior to any data acquisition. When using the high-level data acquisition routines (as in the **adcex1** example programs), a single command can combine the scan sequencer setup, trigger selection, pacer clock programming, and data collection. When using the low-level data acquisition routines (as in **adcex2** and **adcex3**), these operations are broken out into separate commands.

Note: When connecting thermocouple and other low-level signals in addition to high-level signals, the low-level signals should be programmed first into the scan sequence with the high-level signals following in ascending order of signal magnitude.

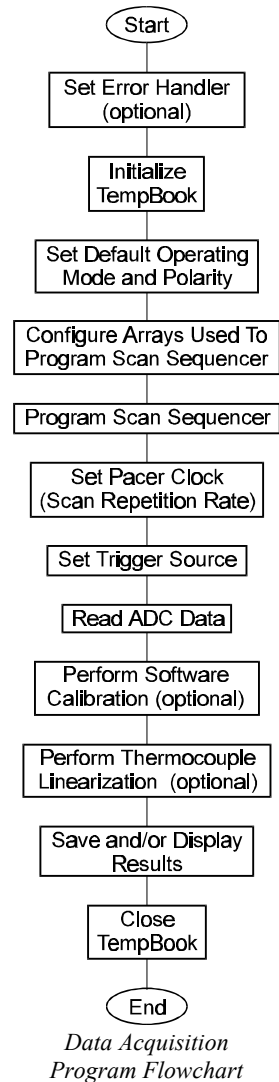
In addition to the 16 externally available analog input channels, two internal channels are provided for offset correction and thermocouple cold junction compensation. The external analog input channels are addressed as channels 0 - 15, the CJC channel is at address 16, and the internal shorted channel is at 18. The CJC channel must be included in the scan group when reading thermocouples. This channel reading is used by the thermocouple linearization functions. The shorted channel, when read at the same gain as an analog input channel, can be used to remove offset errors present at run time in the analog electronics. These topics are covered in greater detail in the *Thermocouple Linearization* and *Zero Compensation* chapters.

Besides zero compensation, software compensation can be used to improve measurement accuracy. The offset and gain errors present after factory calibration are characterized and recorded within a unit-specific calibration-constants file. The calibration-constants file is read by the data acquisition program at run time. The collected ADC data can then be corrected for offset and gain errors. This topic is covered in greater detail in the *Software Calibration and Zero Compensation* chapter.

In addition to analog input, the TempBook provides digital I/O and counter/timer function. Eight bits each of digital I/O are provided which can be accessed as register-addressable I/O ports. Additionally, the 8 bits of digital input can be read as part of the scan group by programming the scan sequencer with the appropriate channel definition. The digital inputs are then returned within the ADC data buffer and are right justified within the 16-bit data words. When the digital inputs are read in this way, the timing relationship between the trigger event and the analog & digital data is fixed.

A user accessible counter / timer is provided through the P0 port of the 8254. This port can be configured into one of several modes including one-shot and pulse-train generation as well as event counting. The clock input to this counter / timer port can be taken from an external or internal 100 kHz source. These topics are covered in greater detail in the Command Reference chapter under the **tbkConfCntr0** and **tbkSetTrig** commands.

The flowchart diagram shows the operation of a typical TempBook data acquisition program.



Driver Options

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

TempBook applications can be written to either the Standard TempBook API or to the Enhanced Daq* API. Standard API functions have the **t**bk****... prefix. The Enhanced API is a new format which can be generically used with the TempBook, WaveBook, DaqBook, DaqBoard, and Daq PC-Card product lines. Enhanced API functions share the **daq**... prefix. Generally,

- If starting with an existing TempBook Application written to Windows 3.1, the quickest port is to use or re-write code to the Standard API.
- If writing a new application, it is best to write code to the Enhanced API due to its improved performance and enhanced feature set (see following).

Standard API (**t**bk****)

The standard API was originally written for the TempBook’s Windows 3.1 driver. However, it can be used under Windows 95 in 16-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

Enhanced API (**daq**)

The Enhanced API for 32-bit systems has several features that are not present in the standard API:

- Multi-device - can concurrently handle up to 4 devices (including WaveBooks, Daq* products, and/or TempBooks)
- 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, we recommend you use the Enhanced API whenever 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++ (v2.2 and greater) Borland C++ (v4.0 and greater)
BASIC Microsoft Visual Basic (v4.0 and previous) QuickBASIC	BASIC Microsoft Visual Basic (v4.0 and greater)
Pascal Turbo Pascal	Delphi Borland Delphi (v2.0)

Note: The enhanced API commands do not work exactly like the standard API commands; refer to chapters 10 and 11.

Accessing TempBook from a Windows Program

The structure of a Windows program generally dictates that actions take place in response to messages such as an operator key-press, mouse action, menu selection, etc. This discussion covers the basic actions needed to control the TempBook. How these actions are combined and coordinated in response to Windows messages is up to the application designer.

Accessing TempBook from a C-for-Windows Program

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

The library, TEMPBOOK.LIB, must be included in the application's makefile or project file so that the linker will find the TempBook functions. This is a large memory model library; the appropriate compiler and linker options for a large memory model program must be invoked. See the documentation for your specific C compiler for a description on using header files, libraries, and memory models.

To use the example program located in the TEMPBOOK\DOS\C directory, create a makefile or project file which consists of the TBKEX.C source file, TBKEX.RC resource file, TBKEX.DEF definition file, TBKEX.ICO icon file and the TEMPBOOK.LIB library.

High-Level Analog Input

The following excerpt from TBKEX.C shows the usage of several high level analog input routines.

```
unsigned sample, buf [10], data[7], data2[80];
int i, scan, chan;
```

Set default operating mode to single-ended, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(0, 1);
```

Get one A/D sample from channel 0 at unity gain and print the unsigned integer that is returned. Since the TempBook contains a 12-bit A/D converter, the least significant 4 bits are undefined. Shifting the value to the right 4 times right justifies the 12 bit value in the 16 bit variable.

```
tbkRd(0, &sample, TgainX1);
sprintf(tempstr, "Result of tbkRd : %4d\r\n\r\n", sample>4);
```

Get 10 samples from channel 0, triggered by the pacer clock with a 1kHz sampling frequency at unity gain. Once the data has been collected, print the 10 samples that were placed in the buffer.

```
tbkRdN(0, buf, 10, TtsPacerClock, 0, 1000, TgainX1);
sprintf(tempstr, "Results of tbkRdN:");
strcat (response, tempstr);
for(i=0;i<8;i++){
    sprintf(tempstr, "%4d ", buf[i]>4);
    strcat (response, tempstr);
}
```

Get 1 scan of channels 0 to 7 at unity gain. Once the data has been collected, print all 8 channel values.

```
tbkRdScan(0, 7, data, TgainX1);
sprintf(tempstr"\r\n\r\nResults of tbkRdscan:\r\n");
strcat (response, tempstr);
for(i=0;i<8;i++) {
    sprintf(tempstr"Channel: %2d Data: %4d\r\n", i, data[i]>4);
    strcat (response, tempstr);
}
```

Get 10 scans of channels 0 to 7, triggered by the pacer clock with a 1kHz sampling frequency and unity gain, then print the scan data.

```
tbkRdScanN(0, 7, data2, 10, TtsPacerClock, 0, 1000, TgainX1);
sprintf(tempstr"\r\nResults of tbkRdScanN Channels 0 - 7:");
strcat (response, tempstr);

for (scan=0 ; scan<8 ; scan++){
    sprintf(tempstr"\r\nScan %d: ", scan);
    strcat (response, tempstr);

    for (chan=0 ; chan<8 ; chan++){
        sprintf(tempstr"%4d ", data2[(scan * 8) + chan]>4);
        strcat (response, tempstr);
    }
}
```

Low-Level Analog Input

The following excerpt from TBKEX.C shows the usage of several low-level analog input routines.

```
unsigned int buf[80], scan, chan;
```

Set the default operating mode to single-ended, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(0, 1);
```

Setup the scan sequencer for channels 0 to 7 with each channel in the scan at unity gain.

```
tbkSetMux(0, 7, TgainX1);
```

Set the scan frequency for 1kHz.

```
tbkSetFreq(1000);
```

Make the Pacer Clock the trigger source. This will also arm and trigger the system.

```
tbkSetTrig(TtsPacerClock, 0, 0, 1);
```

Read 10 scans of data into the language buffer buf, then print the 8 scan values.

```
tbkRdNFore(buf, 10);
sprintf(tempstr"\r\nResults of tbkRdNFore Channels 0 - 7:\r\n");
strcat (response, tempstr);

for (scan=0 ; scan<8 ; scan++)
    {
        sprintf(tempstr"\r\nScan %d: ", scan);
        strcat (response, tempstr);

        for (chan=0 ; chan<8 ; chan++){
            sprintf(tempstr"%4d ", buf[(scan * 8) + chan]>4);
            strcat (response, tempstr);
        }
    }
```

Analog Input in the Background

The following excerpt from TBKEX.C shows the usage of background acquisition functions. These functions setup the TempBook to collect data in the background while your program continues to process new lines of code in the foreground.

```
unsigned int data[80], chans[8], i, scan, chan;
unsigned chargains[8], polarities[8], active;
unsigned longcount;
```

Initialize the TempBook on LPT1 with interrupt 7.

```
tbkInit(LPT1, 7);
```

Set the default operating mode to single-ended, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(0, 1);
```

The TempBook has a sophisticated channel-gain sequencer that allows every channel in the defined scan to have a different gain and unipolar/bipolar setting. The following array assignment will be used to setup the sequencer to sample channels 0 through 7 in bipolar mode at unity gain. If desired, each channel could have been assigned a different gain and/or unipolar/bipolar setting.

```
for(i=0;i<8;i++){
    chans[i] = i;
    gains[i] = TgainX1;
    polarities[i] = 1;
}
```

Once the arrays are loaded with the desired channel numbers and their associated gain and unipolar/bipolar settings, the following function uses them to load the sequencer.

```
tbkSetScan(chans, gains, polarities, 8);
```

Set the Clock to 1 Hz. (This assumes that the time base selection jumper is in the default 1 MHz position)

```
tbkSetClk(1000, 1000);
```

Make the Pacer Clock the trigger source.

```
tbkSetTrig(TtsPacerClock, 1, 0, 0);
```

Setup the background acquisition of 10 scans. As the data is collected, place it into the array called data. Regardless of the status of the acquisition, the program will immediately return from this function call and proceed to the next line in our code. The data will be collected via interrupts and placed in the specified buffer in the background.

```
tbkRdNBack(data, 10, 0, 1);
```

At any point in the program, you can check the status of the background acquisition. The next lines of code poll the background status continuously until it is no longer active, then it exits the do loop and proceeds through the remainder of the program.

```
/* Check if acquisition is complete */
do
{
    tbkGetBackStat(&active, &count);
    sprintf(tempstr"Transfer in progress : %2d scans acquired.\r",count);
    strcat(response, tempstr);
}
while (active != 0);
sprintf(tempstr"\r\nAcquisition complete.\r\n\r\n");
```

Since the background acquisition is complete, print the data in the buffer.

```
sprintf(tempstr"Data Acquired:\r\n");
strcat(response, tempstr);
for (scan=0 ; scan<8 ; scan++){
    sprintf(tempstr, "\nScan %d:", scan);
    strcat(response, tempstr);
    for (chan=0 ; chan<8 ; chan++){
        sprintf(tempstr" %4d", data[(scan * 8) + chan]>4);
        strcat(response, tempstr);
    }
}
```

General Purpose Digital I/O Functions

The following excerpt from TBKEX.C shows the usage of the general purpose digital I/O functions.

```
unsigned char bit, in_bit, out_bit, out_byte, in_byte;
```

Read the state of the digital input bit 3, and place its value in in_bit.

```
tbkRdBit(3, &in_bit);
```

Depending on its state, print a message.

```
if (in_bit) {
    sprintf(tempstr, "Digital input #3 is set\r\n\r\n");
    strcat(response, tempstr);
} else {
    sprintf(tempstr, "Digital input #3 is clear\r\n\r\n");
    strcat(response, tempstr);
}
```

Set the digital output bit 5 to a high state.

```
tbkWtBit(5, 1);
```

The following lines of code use the byte manipulation functions to perform a "walking-bit" test on the digital output port.

```
for (bit=0 ; bit<8 ; bit++)
{
    out_byte = 0x01<bit;// Put a 1 in the bit(th) location of a byte
    tbkWtByte(out_byte);// Write that byte to the digital output port
    sprintf(tempstr, "Digital output byte written 0x%2x\r\n", out_byte);
    strcat(response, tempstr);
}
```

Read the value from the digital input port (DI0 - DI7 inclusive), then print the result.

```
tbkRdByte(&in_byte);
sprintf(tempstr, "Digital input byte is 0x%2x\r\n", in_byte);
strcat(response, tempstr);
```

High-Speed Digital Input

The following excerpt from TBKEX.C shows the usage of the high-speed digital input function calls.

The high-speed digital port can be specified as a channel in the analog scan sequence, just like an analog input channel. In this way the high-speed digital port data is acquired synchronously with the analog input data and is placed in the same data buffer as the analog input data. This program sets up a scan which includes analog channels and the high speed digital port.

```
unsigned chans[9], data[9], i, chan;
unsigned char gains[9], polarity[9];
```

Configure a scan consisting of analog input channels 0 through 7.

```
for (i=0 ; i<8 ; i++){
    chans[i] = i;        // Analog input channels 0 - 7
    gains[i] = TgainX1;  // Unity gain
}
```

Include the high speed port as the last channel in the scan. Note that the high speed channel can be placed anywhere in the scan.

```
chans[8] = TchHighSpeedDig; // High speed digital inputs
gains[8] = TgainX1;         // Put any gain, it doesn't matter
```

Initialize the TempBook on LPT1 with interrupt 7.

```
tbkInit(LPT1, 7);
```

Set the default operating mode to single-ended, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(0, 1);
```

Load the scan sequencer using a NULL pointer for the polarities array which indicates the use of the default, global polarity.

```
tbkSetScan(chans, gains, 0, 9);
```

Set the trigger source to software trigger.

```
tbkSetTrig(TtsSoftware, 0, 0, 0);
```

Trigger a scan.

```
tbkSoftTrig();
```

Read the A/D FIFO buffer and print the results.

```
tbkRdNFore(data, 1);
sprintf("Analog input channels 0 - 7:\n");
strcat(response, tempstr);

for (chan=0 ; chan<8 ; chan++)
    sprintf("  %4d", data[chan]>4);
    strcat(response, tempstr);
}
sprintf(tempstr, "\nHigh speed digital inputs DI0 - DI7:\n");
sprintf(response, tempstr);
```

Counter/Timer Functions

The following excerpt from TBKEX.C shows the usage of the counter/timer functions.

The counter/timer port available through the termination card is the P0 port of an 8254 counter/timer chip. This port can be configured through software to perform several functions which are described in detail in the command reference section under the `tbkConfCntr0` command. This example demonstrates the usage of three of the counter/timer modes. Although the invocation of these modes is demonstrated there is no way to observe the function of the port without the connection of external test equipment or circuitry.

Configure CTR0 to use the internal 100 kHz clock.

```
tbkSetTrig(TtsSoftware, 0, 1, 0);
```

Configure CTR0 to mode 0, High on Terminal Count and write a count value of 100 to counter 0. After this the counter 0 output (OUT0) will go high after 100 pulses are received on the counter 0 gate input (GAT0).

```
tbkConfCntr0(Tc0cHighTermCnt);
tbkWtCntr0(100);
```

Configure CTR0 to mode 1, Hardware Retriggerable One-Shot and write a count value of 1000 to counter 0. After this a rising edge on the counter 0 input (GAT0) will cause the output to go high for 10 msec.*/

```
tbkConfCntr0(Tc0cOneShot);
tbkWtCntr0(1000);
```

Configure CTR0 to mode 3, Square Wave Generator and write a count value of 20 to counter 0. After this a square wave of 5kHz frequency should be present on the counter 0 output (OUT0).

```
tbkConfCntr0(Tc0cSquareWave);
tbkWtCntr0(20);
```

High-Level Thermocouple Data Acquisition

The following excerpt from TBKEX.C demonstrates the use of the TempBook's high-level thermocouple temperature data acquisition routines. These functions have combined scan sequencer setup, ADC data collection, and thermocouple linearization.

```
int i, temp, temps[10];
unsigned buf[1200];
```

Set the default operating mode to differential, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(1, 1);
```

Get one temperature sample from a type J thermocouple on channel 0, then print the result.

```
tbkRdTemp(0, TbkTypeJ, &temp);
sprintf(tempstr, "\r\nResults of tbkRdTemp\r\n");
strcat(response, tempstr);
sprintf(tempstr, "Temperature: %4.1f \r\n", (float)temp/10.0);
strcat(response, tempstr);
```

Get one temperature value from a type J thermocouple on channel 0 which is the average of 10 acquired values, then print the result. This has the effect of reducing the noise content of your signal. The 10 readings will be taken at 1kHz, with only one temperature value returned by the function.

```
tbkRdTempN(0, TbkTypeJ, 10, &temp, buf, 1000, 0);
sprintf(tempstr, "\r\nResults of tbkRdTempN\r\n");
strcat(response, tempstr);
sprintf(tempstr, "Temperature: %4.1f \r\n", (float)temp/10.0);
strcat(response, tempstr);
```

Get one temperature value from a type J thermocouple on channels 0 through 7, then print the result.

```
tbkRdTempScan(0, 7, TbkTypeJ, temps);
sprintf(tempstr, "\r\nResults of tbkRdTempScan\r\n");
strcat(response, tempstr);
for (i=0 ; i<8 ; i++){
    sprintf(tempstr, "Channel %d Temperature: %4.1f \r\n", i,
(float)temps[i]/10.0);
    strcat(response, tempstr);
}
```

Get 8 temperature values from type J thermocouple on channels 0 through 7 which are the average of 10 acquired values, then print the result. This has the effect of reducing the noise content of your signal. The 10 readings will be taken at 1kHz, with only one temperature value for each channel returned by the function.

```
tbkRdTempScanN(0, 7, TbkTypeJ, 10, temps, buf, 1000, 0);
sprintf(tempstr, "\r\nResults of tbkRdTempScanN\r\n");
strcat(response, tempstr);
for (i=0 ; i<8 ; i++){
    sprintf(tempstr, "Channel %d Temperature: %4.1f \r\n", i,
(float)temps[i]/10.0);
    strcat(response, tempstr);
}
```

Thermocouple Linearization

The following excerpt from TBKEX.C, demonstrates the use of the TempBook's thermocouple linearization routines.

```
unsigned i, chans[11], data[1100];
unsigned char gains[11];
int temp[8];
```

The following lines of code assign channels and gains to the arrays that will be used to create a scan sequence. The position of the CJC in the scan and the assignment of the gains for all of the channels must conform to the conventions used by the linearization routines. See the command reference section for more information on scan configuration.

```
/* Configure chans array */
chans[0] = 18; /* Shorted channel*/
chans[1] = 18; /* Shorted channel */
chans[2] = 16; /* CJC channel */
chans[3] = 0; /* Thermocouple on channel 0 */
chans[4] = 1; /* Thermocouple on channel 1 */
chans[5] = 2; /* Thermocouple on channel 2 */
chans[6] = 3; /* Thermocouple on channel 3 */
chans[7] = 4; /* Thermocouple on channel 4 */
chans[8] = 5; /* Thermocouple on channel 5 */
chans[9] = 6; /* Thermocouple on channel 6 */
chans[10] = 7; /* Thermocouple on channel 7 */

/* Configure gains array */
gains[0] = TbkBiCJC; /* Bipolar CJC gain setting */
gains[1] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[2] = TbkBiCJC; /* Bipolar CJC gain setting */
gains[3] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[4] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[5] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[6] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[7] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[8] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[9] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[10] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
```

Set the default operating mode to differential, bipolar. These parameters will affect all scanned channels.

```
tbkSetMode(1, 1);
```

Configure the scan sequencer, with the desired channels and gains. A zero for the polarity argument forces the use of the default polarity for all of the channels.

```
tbkSetScan(chans, gains, 0, 11);
```

Set the pacer clock for a 1 msec period.

```
tbkSetFreq(1000);
```

Use the Pacer Clock as the trigger source.

```
tbkSetTrig(TtsPacerClock, 0, 0, 0);
```

The following function tells all of the TbkTC functions to use zero correction. Zero correction compensates for offset drift in the electronics due to ambient temperature changes and/or age.

```
tbkTCAutoZero(1);
```

Read 100 scans and place the raw data in the supplied buffer.

```
tbkRdNFore(data, 100);
```

Reset the trigger to prevent a buffer overrun.

```
tbkSetTrig(TtsSoftware, 1, 0, 0);
```

Configure and perform Thermocouple Linearization with block averaging on the raw ADC counts. The raw counts are in the data buffer, data. The converted temperature values will be returned in the buffer temp. With block averaging enabled one temperature, the average of all 100 scans, will be returned for each channel.

```
tbkTCSetup(11, 2, 8, TbkTypeJ, 1, 0);
tbkTCConvert(data, 100, temp, 8);
```

The converted temperatures can now be printed to the screen.

```
for (i=0 ; i<8 ; i++){
    sprintf(tempstr, "%8.1f ", (float)temp[i] / 10.0);
    strcat(response, tempstr);
}
sprintf(tempstr, "\r\n");
strcat(response, tempstr);
SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
EmptyMessageQueue(myDlg);
```

Sample Programs

High-Level Analog Input

```

/*****
File:Adcx1.c
Description:This example demonstrates the use of TempBook/66s highest level
ADC functions. These functions have combined scan sequencer setup and ADC
data collection.
Functions Used:tbkRd(unsigned chan, unsigned *sample, unsigned char gain)
tbkRdN(unsigned chan, unsigned *buf, unsigned count, unsigned char trigger,
unsigned char oneShot, float freq, unsigned char gain) tbkRdScan(unsigned
startChan, unsigned endChan, unsigned * buf, unsigned char gain)
tbkRdScanN(unsigned startChan, unsigned endChan, unsigned * buf, unsigned
count, unsigned char trigger, unsigned char oneShot, float freq, unsigned
char gain)
*****/

unsigned sample, buf[10], data[7], data2[80];
int i, scan, chan;    sprintf(response, "\r\nAdcx1.c\r\n\r\n");

    /* Set the default operating mode to single-ended, bipolar */
tbkSetMode(0, 1);

    /* Get 1 ADC sample from channel 0 at unity gain. */
tbkRd(0, &sample, TgainX1);

    /* Print results using a 12 bit data format */
    sprintf(tempstr, "Result of tbkRd : %4d\r\n\r\n", sample>4);
    strcat(response, tempstr);

    /* Get 10 samples from channel 0, trigged by the pacer clock with a */
    /* 1000 Hz sampling frequency at unity gain. */
tbkRdN(0, buf, 10, TtsPacerClock, 0, 1000, TgainX1);

    /* Print the results using a 12 bit data format */

    sprintf(tempstr, "Results of tbkRdN:");
    strcat(response, tempstr);

    for(i=0; i<8; i++)    {
        sprintf(tempstr, "%4d ", buf[i]>4);
        strcat(response, tempstr);
    }

    /* Get 1 sample from channels 0 through 7 at unity gain. */

```



```

tbkRdScan(0, 7, data, TgainX1);

    /* Print the results using a 12 bit data format */

sprintf(tempstr,"\r\n\r\nResults of tbkRdscan:\r\n");
strcat(response,tempstr);

for(i=0;i<8;i++)    {
    sprintf(tempstr,"Channel: %2d Data: %4d\r\n", i, data[i]>4);
    strcat(response,tempstr);
}

    /* Get 10 samples from channels 0 - 7, triggered by the pacer clock
       with a 1000 Hz sampling frequency and unity gain. */

tbkRdScanN(0, 7, data2, 10, TtsPacerClock, 0, 1000, TgainX1);

    /* Print the results using a 12 bit data format */

sprintf(tempstr,"\r\nResults of tbkRdScanN Channels 0 - 7:");
strcat(response,tempstr);

for (scan=0 ; scan <8; scan++)    {
    sprintf(tempstr,"\r\nScan %d:  ", scan);
    strcat(response,tempstr);

    for (chan=0 ; chan<8 ; chan++) {

        sprintf(tempstr,"%4d ", data2[(scan * 8) + chan]>4)
;        strcat(response,tempstr);
    }
}

    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response)
; EmptyMessageQueue(myDlg);
}

```

Low-Level Analog Input

```

/*****
File:Adcex2.c
Description:This example demonstrates the use of the TempBook/66's lower
level ADC functions. These functions allow separate scan
sequencer setup, trigger source selection, and ADC data collection.
Functions Used:bkSetMux(unsigned startChan, unsigned endChan, unsigned char
gain)
tbkSetFreq(float freq) tbkSetTrig(unsigned char trigger, unsigned char
oneShot, unsigned char ctr0mode, unsigned char pacerMode)
tbkRdNFore(unsigned _far *buf, unsigned count)
*****/

    unsigned int buf[80], scan, chan;
    sprintf(response,"\r\nAdcex2.c\r\n");

    /* Set the default mode of operation to single-ended bipolar */
    tbkSetMode(0, 1);

    /* Set the scan sequencer for channels 0 - 7 at unity gain */
    tbkSetMux(0, 7, TgainX1);

    /* Set the scan frequency for 1000 Hz */
    tbkSetFreq(1000);

    /* Set the trigger source for the Pacer Clock. Note that the trigger
       is armed immediately */
    tbkSetTrig(TtsPacerClock, 0, 0, 1);

    /* Read 10 scans of data */

```

```

tbkRdNFore(buf, 10);

    /* Print the results using a 12 bit data format */
    sprintf(tempstr, "\r\nResults of tbkRdNFore Channels 0 - 7:\r\n");
    strcat(response, tempstr);

    for (scan=0 ; scan<8 ; scan++)    {
        sprintf(tempstr, "\r\nScan %d: ", scan);    strcat(response, tempstr);
        for (chan=0 ; chan<8 ; chan++) {
            sprintf(tempstr, "%4d ", buf[(scan * 8) + chan]>4);
            strcat(response, tempstr);
        }
    }

    sprintf(tempstr, "\r\n");
    strcat(response, tempstr);

    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```

Analog Input in the Background

```

/*****
File: Adcex3.c
Description: This example demonstrates the use of the TempBook/66's
            background transfer routines.
Functions Used: tbkRdNBack(unsigned _far *buf, unsigned count, unsigned char
cycle,
unsigned char updateSingle)
tbkGetBackStat(unsigned char _far *active, unsigned long _far *count)
*****/

unsigned int    data[80], chans[8], i, scan, chan;
unsigned char   gains[8], polarities[8], active;
unsigned long   count;

sprintf(response, "\r\nAdcex3.c\r\n\r\n");

    /* Set the default mode of operation to single-ended bipolar */
    tbkSetMode(0, 1);

    /* Set channels, gains, & polarities arrays to channels 0-7,
       bipolar, at unity gain. */

    for(i=0; i<8; i++)
    {
        chans[i] = i;
        gains[i] = TgainX1;
        polarities[i] = 1;
    }

    /* Load the scan sequencer FIFO */
    tbkSetScan(chans, gains, polarities, 8);

    /* Set Clock : 1Hz - xtal set to 1MHz */
    tbkSetClk(1000, 1000);

    /* Set pacer clock trigger source */
    tbkSetTrig(TtsPacerClock, 1, 0, 0);

    /* Read 10 scans of data in the background, cycle off, */
    /* update on each scan */
    tbkRdNBack(data, 10, 0, 1);

    /* Check if acquisition is complete */
    do    {

```

```

        tbkGetBackStat(&active, &count);
        sprintf(tempstr,"Transfer in progress : %2d scans acquired.\r",count);
        strcat(response,tempstr);
    }
while (active != 0);
sprintf(tempstr,"\r\nAcquisition complete.\r\n\r\n");
strcat(response,tempstr);

    /* Print results using a 12 bit data format */

    sprintf(tempstr,"Data Acquired:\r\n");
    strcat(response,tempstr);

    for (scan=0 ; scan<8 ; scan++)
        {
        sprintf(tempstr,"\r\nScan %d:", scan);
        strcat(response,tempstr);

        for (chan=0 ; chan <8; chan++) {
            sprintf(tempstr," %4d", data[(scan * 8) + chan]>4);
            strcat(response,tempstr);
        }
    }

    sprintf(tempstr,"\r\n");
    strcat(response,tempstr);

    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```

General Purpose Digital I/O

```

/*****
File: Adcex3.c
Description: This example demonstrates the use of the TempBook/66's general
purpose digital I/O ports.
Functions Used: tbkRdBit(unsigned char bitNum, unsigned char *bitVal)
tbkRdByte(unsigned char *byteVal)
tbkWtBit(unsigned char bitNum, unsigned char bitVal)
tbkWtByte(unsigned char byteVal)
*****/

*/ unsigned char bit, in_bit, out_byte, in_byte;
sprintf(response,"\r\nDigex1.c\r\n\r\n");

    /* Read the state of digital input DI3 */
    tbkRdBit(3, &in_bit);

    if (in_bit) {
        sprintf(tempstr,"Digital input #3 is set\r\n\r\n");
        strcat(response,tempstr); } else {
        sprintf(tempstr,"Digital input #3 is clear\r\n\r\n");
        strcat(response,tempstr); }

    /* Set digital output DO5 */
    tbkWtBit(5, 1);
    /* Perform a 'walking bit' test by setting each digital
    output in order from DO0 to DO7 */
    for (bit=0 ; bit<8 ; bit++)
        {
        out_byte = 0x01<bit;
        // Put a 1 in the bit(th) location of a byte
        tbkWtByte(out_byte);

        // Write that byte to the digital output port

```

```

        sprintf(tempstr,"Digital output byte written 0x%2x\r\n", out_byte);
        strcat(response,tempstr);
    }

    /* Read the value from the digital input port (DI0 - DI7 inclusive) */
    tbkRdByte(&in_byte);

    sprintf(tempstr,"Digital input byte is 0x%2x\r\n", in_byte);
    strcat(response,tempstr);
    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```

High-Speed Digital Input

```

/*****
File: Digex2.c
Description:This example demonstrates the use of the TempBook's high speed
digital inputs.
Functions Used:tbkSetScan(unsigned *chans, unsigned char *gains, unsigned
char *polarity, unsigned count)
tbkRdNFore(unsigned _far *buf, unsigned count)
*****/

unsigned chans[9], data[9], i, chan;
unsigned char gains[9], polarity[9];

sprintf(response,"\nDigex2.c\n\n");

/* Configure the channels and gains arrays for analog inputs 0-7 */
/* plus the high speed digital inputs */

for (i=0 ; i<8 ; i++)
{
    chans[i] = i;          /* Analog input channels 0 - 7 */
    gains[i] = TgainX1;   /* Unity gain */
    polarity[i] = 1;      /* Bipolar */
}

chans[8] = TchHighSpeedDig; /* High speed digital inputs */
gains[8] = TgainX1;        /* Put any gain, it doesn't matter */
polarity[8] = 1;          /* Put any polarity, it doesn't matter */

/* Set default operation to single-ended, bipolar */

tbkSetMode(0, 1);

/* Load the scan sequencer, NULL pointer for polarities array */
/* indicates use default polarity */

tbkSetScan(chans, gains, 0, 9);

/* Set software trigger source */

tbkSetTrig(TtsSoftware, 0, 0, 0);

/* Trigger a scan */

tbkSoftTrig();

/* Read the ADC FIFO buffer */

tbkRdNFore(data, 1);

/* Print the results */

sprintf(tempstr,"Analog input channels 0 - 7:\n");

```

```

    strcat(response,tempstr);

    for (chan=0 ; chan<8 ; chan++) {
        sprintf(tempstr," %4d", data[chan]>4);
        strcat(response,tempstr);
    }

    sprintf(tempstr,"\nHigh speed digital inputs DI0 - DI7:\n");
    strcat(response,tempstr);

    sprintf(tempstr," 0x%x", (char)data[8]);
    strcat(response,tempstr);

    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```

Counter Timer Functions

```

/*****
File:          Ctrex1.c
Description:   This example demonstrates the use of the TempBook's
              counter timer functions.
Functions Used:
              tbkConfCntr0(unsigned char config)
              tbkWtCntr0(unsigned cntr0)
*****/

    sprintf(response,"\r\nctrex1.c\r\n\r\n");

    /* Configure CTR0 to use the internal 100kHz clock */

    tbkSetTrig(TtsSoftware, 0, 1, 0);

    /* Configure CTR0 to mode 1, Hardware Retriggerable One-Shot and write
       a count value of 1000 to counter 0. After this a rising edge on the
       counter 0 input (GAT0) will cause the output to go high
       for 10 msec. */

    tbkConfCntr0(Tc0cOneShot);

    tbkWtCntr0(1000);

    /* Configure CTR0 to mode 3, Square Wave Generator and write a count
       value of 20 to counter 0. After this a square wave of 5kHz
       frequency should be present on the counter 0 output (OUT0) */

    tbkConfCntr0(Tc0cSquareWave);

    tbkWtCntr0(20);
    /* Configure CTR0 to use an external clock */
    tbkSetTrig(TtsSoftware, 0, 0, 0);
    /* Configure CTR0 to mode 0, High on Terminal Count and write a count
       value of 100 to counter 0. After this the counter 0 output (OUT0)
       will go high after 100 pulses are received on the counter 0 clock
       input (CLK0) */
    tbkConfCntr0(Tc0cHighTermCnt);
    tbkWtCntr0(100);
    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```

High-Level Thermocouple Measurement

```
*****
File:
Description:
This example demonstrates the use of the TempBook's high level
    thermocouple temperature data acquisition routines. These
    functions have combined scan sequencer setup, ADC data
    collection, and thermocouple linearization.
Functions Used:
tbkRdTemp(unsigned chan, unsigned tcType, int * temp)
    tbkRdTempN( unsigned chan, unsigned tcType, unsigned count, int *
temp,
unsigned * buf, float freq, unsigned avg ) tbkRdTempScan(unsigned startChan,
unsigned endChan,unsigned tcType, int * temp) tbkRdTempScanN(unsigned
startChan,
unsigned endChan, unsigned tcType, unsigned count, int * temp, unsigned *
buf,
float freq, unsigned avg)
*****/

int i, temp, temps[10];
unsigned buf[1200];

sprintf(response, "\r\nTempex1.c\r\n");

    /* Set the default mode of operation to differential bipolar */

tbkSetMode(1, 1);

    /* Get 1 ADC sample from a type J thermocouple on channel 0 and
convert the reading to a temperature. Print the result. */

tbkRdTemp(0, TbkTypeJ, &temp);
sprintf(tempstr, "\r\nResults of tbkRdTemp\r\n");
strcat(response, tempstr);

sprintf(tempstr, "Temperature: %4.1f \r\n", (float)temp/10.0);
strcat(response, tempstr);

    /* Get 10 ADC samples from a type J thermocouple on channel 0 and
convert the readings to a single temperature using block averaging.
Print the results. */

tbkRdTempN(0, TbkTypeJ, 10, &temp, buf, 1000, 0);

sprintf(tempstr, "\r\nResults of tbkRdTempN\r\n");
strcat(response, tempstr);

sprintf(tempstr, "Temperature: %4.1f \r\n", (float)temp/10.0);
strcat(response, tempstr);

    /* Get 1 ADC sample each from type J thermocouples on channels 0
through
    7 and convert the readings to temperatures. Print the results */

tbkRdTempScan(0, 7, TbkTypeJ, temps);

sprintf(tempstr, "\r\nResults of tbkRdTempScan\r\n");
strcat(response, tempstr);

for (i=0 ; i<8 ; i++) {
    sprintf(tempstr, "Channel %d Temperature: %4.1f \r\n", i,
(float)temps[i]/10.0);
    strcat(response, tempstr);
}

```

```

        /* Get 10 ADC samples each from type J thermocouples on channels 0
        through 7 and convert the readings to temperatures using block
        averaging. Print the results */

        tbkRdTempScanN(0, 7, TbkTypeJ, 10, temps, buf, 1000, 0);

        sprintf(tempstr, "\r\nResults of tbkRdTempScan\r\n");
        strcat(response, tempstr);

        for (i=0 ; i<8 ; i++) {
            sprintf(tempstr, "Channel %d Temperature: %4.1f \r\n", i,
(float)temps[i]/10.0);      strcat(response, tempstr);
        }

        SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
        EmptyMessageQueue(myDlg);
    }

```

Low-Level Thermocouple Linearization

```

*****
File:          Description:
This example demonstrates the use of the TempBook's thermocouple
linearization
routines.
Functions Used:
tbkTCSetup(unsigned nscan, unsigned cjcPosition, unsigned ntc,
unsigned tcType, unsigned char bipolar, unsigned avg)
tbkTCConvert(unsigned _far *counts, unsigned scans,
int _far *temp, unsigned ntemp)
*****/

/* unsigned i, chans[11], data[1100];
   unsigned char gains[11];   int temp[8];
   sprintf(response, "\r\nTempex2.c\r\n");

        /* Configure chans array */
chans[0] = 18; /* Shorted channel*/
chans[1] = 18; /* Shorted channel*/
chans[2] = 16; /* CJC channel */
chans[3] = 0; /* Thermocouple on channel 0 */
chans[4] = 1; /* Thermocouple on channel 1 */
chans[5] = 2; /* Thermocouple on channel 2 */
chans[6] = 3; /* Thermocouple on channel 3 */
chans[7] = 4; /* Thermocouple on channel 4 */
chans[8] = 5; /* Thermocouple on channel 5 */
chans[9] = 6; /* Thermocouple on channel 6 */
chans[10] = 7; /* Thermocouple on channel 7 */

        /* Configure gains array */
gains[0] = TbkBiCJC; /* Bipolar CJC gain setting */
gains[1] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[2] = TbkBiCJC; /* Bipolar CJC gain setting */
gains[3] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[4] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[5] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[6] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[7] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[8] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[9] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */
gains[10] = TbkBiTypeJ; /* Bipolar Type J Thermocouple gain setting */

        /* Set the default mode of operation to differential bipolar */
        tbkSetMode(1, 1);

        /* Configure the scan sequencer, will use default polarity. */
        tbkSetScan(chans, gains, 0, 11);

```

```

    /* Set the pacer clock for a 1 msec period */
    tbkSetFreq(1000);

    /* Configure for pacer clock trigger, continuous mode. */
    tbkSetTrig(TtsPacerClock, 0, 0, 0);

    /* Tell tbkTC.. functions to use zero correction */
    tbkTCAutoZero(1);

    /* Read 100 scans of ADC data */
    tbkRdNFore(data, 100);

    /* Reset the trigger to prevent a buffer overrun */
    tbkSetTrig(TtsSoftware, 1, 0, 0);

    /* Configure and Perform Thermocouple Linearization with block
    averaging */
    tbkTCSetup(11, 2, 8, TbkTypeJ, 1, 0);
    tbkTCConvert(data, 100, temp, 8);

    /* The converted temperatures can now be printed to the screen */
    sprintf(tempstr, "\r\nThermocouple temperatures: Channels 0 through
7\r\n\r\n");  strcat(response, tempstr);

    for (i=0 ; i<8 ; i++) {
        sprintf(tempstr, "%8.1f ", (float)temp[i] / 10.0);
        strcat(response, tempstr);
    }
    sprintf(tempstr, "\r\n");
    strcat(response, tempstr);

    SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response);
    EmptyMessageQueue(myDlg);
}

```


Command Summary, C Language (Standard API)

Command	Description
Initialization, Communication and Error Handling:	
tbkInit (uint lptPort, uchar lptIntr)	Establish communication with the TempBook at the specified LPT port and interrupt
tbkSelectPort (uint lptPort)	Select an initialized TempBook as the current TempBook
tbkClose (void)	End communication with the TempBook
tbkSetErrHandler (tbkErrorHandlerFPT tbkErrHandler)	Specify a user-defined routine to call when error occurs
tbkDefaultHandler (int tbkErrnum)	Default error handling routine
tbkGetProtocol (int *protocol)	Get the current parallel port communication protocol
tbkSetProtocol (int protocol)	Set the parallel port communication protocol
Separate Scan Sequence, Pacer Clock and Trigger Commands:	
tbkSoftTrig (void)	Issue a software trigger to the TempBook
tbkSetFreq (float freq)	Set the pacer clock frequency
tbkGetFreq (float _far *freq)	Read the pacer clock frequency
tbkSetClk (uint ctrl1, uint ctrl2)	Set the pacer clock frequency divider registers
tbkSetMode (uchar di_se, uchar polarity)	Specify input signal type (single-ended or differential)
tbkSetMux (uint startChan, uint endChan, uchar gain)	Setup the scan sequencer for a range of channels at the same gain
tbkGetScan (uint *chans, uchar *gains, uchar *polarity, uint *count)	Read the scan sequencer contents
Digital I/O and Counter/Timer Functions:	
tbkSetScan (uint *chans, uchar *gains, uchar *polarity, uint count)	Setup the scan sequencer with specific channels, gains, and polarities
tbkSetTrig (uchar trigger, uchar oneShot, uchar ctr0mode, uchar pacerMode)	Set the trigger source for analog data acquisition
Separate Data Read and Background Transfer Commands:	
tbkRdFore (uint _far *sample)	Read a single ADC sample in the foreground and increment the scan sequencer
tbkRdNFore (uint _far *buf, uint count)	Read multiple scans in the foreground
tbkRdNBack (uint _far *buf, uint count, uchar cycle, uchar updateSingle)	Read multiple scans in the background using interrupts
tbkGetBackStat (uchar _far *active, uint long _far *count)	Determine if a background transfer is still in progress
tbkStopBack (void)	Stop the background transfer
Data Read Commands with Combined Scan Sequencer, Pacer Clock and Trigger Setup and Thermocouple Linearization:	
tbkRd (uint chan, uint *sample, uchar gain)	Read a single sample from the specified channel
tbkRdN (uint chan, uint *buf, uint count, uchar trigger, uchar oneShot, float uchar gain)	Read multiple samples from the specified channel
tbkRdScan (uint startChan, uint endChan, uint *buf, uchar gain)	Read a single sample from the specified range of channels
tbkRdScanN (uint startChan, uint endChan, uint *buf, uint count, uchar trigger, uchar oneShot, float freq, uchar gain)	Read multiple samples from the specified range of channels
tbkRdTemp (uint chan, uint tcType, int temp)	Read the thermocouple temperature once from the specified channel
tbkRdTempN (uint chan, uint tcType, uint count, int *temp, uint *buf, float freq, uint avg)	Read the thermocouple temperature from the specified channel multiple times with optional averaging
tbkRdTempScan (uint startChan, uint endChan, uint tcType, int temp)	Read the thermocouple temperature once from each channel in a range
tbkRdTempScanN (uint startChan, uint endChan, uint tcType, uint count, int *temp, uint *buf, float freq, uint avg)	Read the thermocouple temperature from each channel in a range multiple times using optional averaging
Thermocouple Linearization Commands:	
tbkTCSetup (uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg)	Specify scan information used by tbkTCCConvert
tbkTCCConvert (uint _far *counts, uint scans, int _far *temp, uint ntemp)	Perform thermocouple linearization on scan data
tbkTCSetupConvert (uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg, uint _far *counts, uint scans, int _far *temp, uint ntemp)	Combined tbkTCSetup and tbkTCCConvert
tbkTCAutoZero (uint zero)	Tell the thermocouple linearization functions that auto zeroing will be used

Software Calibration and Zero Compensation Commands:	
tbkReadCalFile (char *calfile)	Read the calibration constants text file
tbkCalSetup (uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint bipolar, uint noOffset)	Specify scan information used by tbkCalConvert
tbkCalConvert (uint *counts, uint scans)	Perform software calibration on scan data
tbkCalSetupConvert (uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint bipolar, uint noOffset, uint *counts, uint scans)	Combined tbkCalSetup and tbkCalConvert
tbkZeroSetup (uint nscan, uint zeroPos, uint readingsPos, uint nReadings)	Specify scan information used by tbkZeroConvert function
tbkZeroConvert (uint *counts, uint scans)	Perform zero compensation on scan data
tbkZeroSetupConvert (uint nscan, uint zeroPos, uint readingsPos, uint nReadings, uint *counts, uint scans)	Combined tbkZeroSetup and tbkZeroConvert
tbkConfCntr0 (uchar config)	Set the operating mode of the counter/timer
tbkWtCntr0 (uint cntr0)	Write a value to the counter/timer count down register
tbkRdCntr0 (uint _far *cntr0, uchar mode)	Read the counter/timer hold register
tbkRdBit (uchar bitNum, uchar *bitVal)	Read a specific digital input bit
tbkRdByte (uchar *byteVal)	Read all digital inputs
tbkWtBit (uchar bitNum, uchar bitVal)	Write to a specific digital output bit
tbkWtByte (uchar byteVal)	Write to all digital outputs
Pretrigger Operation Commands:	
tbkSetTrigPreT (uchar source, uint channels, uint level, uint preCount, uint postCount)	Setup the trigger source and level for a pretrigger operation
tbkRdNForePreT (uint _far *buf, uint count, uint _far *retcount, uchar _far active)	Read multiple scans for a pretrigger operation in the foreground
tbkRdNForePreTWait (uint _far *buf, uint count, uint _far *retcount)	Read multiple scans for a pretrigger operation in the foreground continuing until the trigger event occurs
tbkRdNBackPreT (uint _far *buf, uint count, uchar cycle)	Read multiple scans for a pretrigger operation in the background

Note: The enhanced API commands do not work exactly like the standard API commands used in this chapter.

This section describes how to use the TempBook's software calibration and zero compensation functions to correct for gain and offset errors. To use the calibration constants shipped with the board, DaqView users should follow the instructions given on the calibration document containing these constants. The program will automatically use these constants.

Programmers wishing to use the TempBook's thermocouple linearization functions with auto-zero compensation (rather than calling the zero compensation functions manually) should refer to the *Thermocouple Linearization* chapter of this manual.

Both software calibration and zero compensation increase the accuracy of the TempBook and its expansion cards by correcting for gain and offset errors. For example, when using a TempBook with software calibration, accuracy better than 1°C can be achieved. The calibration operation removes static gain and offset errors that are inherent in the hardware. This operation uses calibration constants, usually measured at the factory, to adjust for gain and optionally offset errors. The calibration constants do not change during the execution of a program but are different for each card and programmable gain setting.

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. By reading a shorted channel on the same card at the same gain as the desired channel, the offset can be removed at run-time. **Note:** The TempBook has channel 18 permanently shorted for performing zero compensation.

Software Calibration

Software calibration functions are designed to adjust TempBook 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 re-compiling the program.

The calibration process is divided up into three 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.

All of the functions prototypes, return error codes and definitions are located in the TempBook.H header file (C language) or similar header file (other languages).

Initializing the Calibration Constants

Each TempBook is shipped with a disk containing a calibration constants file. The file is named `serial_no.cal` where *serial_no* is the serial number of the TempBook for which the constants were generated. This file should be copied into the directory from which the user's program will be run. For convenience, this file can be renamed `tempbook.cal` which is the default calibration filename.

The initialization function for reading in the calibration constants from the calibration text file is **tbkReadCalFile**. The C language version of **tbkReadCalFile** is similar to that of other languages and operates as follows:

```
int tbkReadCalFile(char *calfile)
```

<code>char *calfile</code>	calfile contains the path (optional) and filename of the calibration file. If calfile is NULL or empty (""), the default calibration file TempBook.CAL will be read.
----------------------------	--

This function, which is 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 a non-zero error code will be returned by the **tbkReadCalFile** function.

Calibration Setup and Conversion

Once the cal constants have been read from the cal file, they can be used by the **tbkCalSetup** and **tbkCalConvert** functions. The **tbkCalSetup** function 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 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 chapter.

int

tbkCalSetup ()

uint nscan	The number of readings in a single scan.
uint readingsPos	The position of the readings to be calibrated within the scan.
uint nReadings	The number of readings to calibrate.
uint chanType	The type of channel from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel and 0 when reading any other channel.
uint chanGain	The gain setting of the channels to be calibrated.
uint bipolar	Non-zero if the TempBook is configured for bipolar readings.
uint noOffset	If non-zero, the offset cal constant will not be used to calibrate the readings.

The **tbkCalConvert** function performs the actual calibration of one or more scans according to the previously called **tbkCalSetup** function. This function will modify the array of data passed to it.

int

tbkCalConvert ()

uint *counts	The raw data from one or more scans.
uint scans	The number of scans of raw data in the counts array.

For convenience, both the setup and convert steps can be performed with one call to **tbkCalSetupConvert**. This is useful when the calibration needs to be performed multiple times because data was read from channels at different gains.

int

tbkCalSetupConvert ()

uint nscan	The number of readings in a single scan.
uint readingsPos	The position of the readings to be calibrated within the scan.
uint 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 and 0 when reading any other channel.
uint chanGain	The gain setting of the channels to be calibrated.
uint bipolar	Non-zero if the TempBook 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.

Calibration Example

In this example, several TempBook channels will be read and calibrated. This example assumes that the calibration file has been created according to the initializing calibration constants section of this chapter.

Although all of the channels in this example are read at the same gain, the same principles apply to calibration of channel readings at different gains. In that situation, channels to be read are grouped by gain within the scan sequence. Then, **tbkCalSetup** and **tbkCalConvert** (or **tbkCalSetupConvert**) would be called once for each group of channels at a particular gain.

```
void main(void)
{
    unsigned sample, buf[10], data[8];
    int i, scan, chan;
    printf("\nAdcex4.c/n");

    /* Set error handler and initialize TempBook*/
    tbkSetErrorHandler(myhandler);
    tbkInit(LPT1, 7);
```

```

        /*Read the calibration constants from the calibration constant text file
        assuming the default name `tempbook.cal'*/
tbkReadCalFile("");

        /* Set the default operating mode to differential, bipolar */
tbkSetMode(1,1);

        /* Get 1 sample from channels 0 through 7 at unity gain */
tbkRdScan(0, 7, data, TgainX1);

        /* Print the uncalibrated samples using a 12-bit format */
printf("\nUncalibrated Results of tbkRdScan:\n");
for(i=0; i<8; i++)
    printf("Channel: %2d Data: %4d\n", i, data[i]>4);

        /* Setup and perform offset and gain software calibration of data */
tbkCalSetup(8, /* 8 readings within a scan */
            0, /* First reading to be cal'd at position 0 */
            8, /* Calibrate 8 readings per scan */
            0, /* Channel type is 0 for non-CJC */
            TgainX1 /* Reading taken at X1 gain */
            1, /* Readings are bipolar */
            0); /* Perform zero as well as gain calibration */
tbkCalConvert (data, /*Pointer to array of readings */
              1); /* 1 scan in that array */

        /* Print the calibrated samples using a 12-bit format */
printf("\nCalibrated Results of tbkRdScan:\n");
for(i=0; i<8; i++)
    printf("Channel: %2d Data: %4d\n", i, data[i]>4);

        /* Close and exit */
tbkClose
}

```

Zero Compensation

The zero compensation functions require a shorted channel to be sampled at the same gain as the channels to be compensated.

The **tbkZeroSetup** 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. This function does not do the conversion. A non-zero return value indicates an invalid parameter error.

```
int tbkZeroSetup ()
```

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 per scan to be zero compensate.

The **tbkZeroConvert** function compensates one or more scans according the previously called **tbkZeroSetup** function. This function will modify the array of data passed to it.

```
int tbkZeroConvert ()
```

uint *counts	The raw data from one or more scans.
uint scans	The number of scans of raw data in the counts array.

For convenience, both the setup and convert steps can be performed with one call to **tbkZeroSetupConvert**. This is useful when the zero compensation needs to be performed multiple times because data was read from channels at different gains.

```
int tbkZeroSetupConvert ()
```

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.
uint nReadings	The number of readings per scan to be zero compensated.
uint *counts	The raw data from one or more scans.
uint scans	The number of scans of raw data in the counts array.

Zero Compensation Example

In this example, several TempBook channels will be read and zero compensated. Although all of the channels in this example are read at the same gain, the same principles apply to calibration of channel readings at different gains. In that situation, channels to be read are grouped by gain within the scan sequence. Also within the scan sequence, the TempBook's internal shorted channel would be read once at each gain used for channel measurement. Then, `tbkZeroSetup` and `tbkZeroConvert` (or `tbkZeroSetupConvert`) would be called once for each group of channels at a particular gain making sure that the corresponding shorted channel reading for that gain is passed.

```
void main(void)
{
    unsigned int data[9], chans[9], i;
    unsigned char gains[9], polarities[9];
    printf("\nAdcx5.c\n");

    /* Initialize the channels, gains, and polarities arrays*/
    chans[0] = 18;          /* Shorted channel */
    gains[0] = TgainX1     /* Same gain as analog input channels */
    polarities[0] = 1;    /* Bipolar */
    for(i=0; i<8; i++)
    {
        chans[i] = i-1;   /* Analog input channels 0 - 7 */
        gains[i] = TgainX1 /* Unity gain */
        polarities[i] = 1; /* Bipolar */
    }

    /*Set error handler and initialize TempBook */
    tbkSetErrorHandler(myhandler);
    tbkInit(LPT1, 7);

    /* Set the default operating mode to differential, bipolar */
    tbkSetMode(1,1);

    /* Set the scan sequencer */
    tbkSetScan(chans, gains, polarities, 9);

    /* Configure trigger source and issue software trigger */
    tbkSetTrig(TtsSoftware, 1, 0, 0);
    tbkSoftTrig();

    /* Read the ADC data */
    tbkRdNFore(data, 1);

    /* Print the uncompensated samples using a 12-bit format.*/
    printf("\nResults of tbkRdNFore before zero compensation:\n");
    for(i=0; i<8; i++)
        printf("Channel: %2d Data: %4d\n", i, data[i]>4);

    /* Setup and perform zero compensation of the data */
    tbkZeroSetup(9, /* 9 readings within a scan */
                0, /* Shorted channel at position 0 */
                1, /* First reading to be compensated at position 1 */
                8, /* Compensate 8 readings per scan */
    tbkZeroConvert(data, /*Pointer to array of readings */
                  1); /* 1 scan in that array */
}
```

```

        /* Print the compensated samples using a 12-bit format */
printf("\nResults of tbkRdNFore after zero compensation:\n");
for(i=0; i<8; i++)
    printf("Channel: %2d Data: %4d\n", i, data[i]>4);

    /* Close and exit */
tbkClose
}

```

Automatic Zero Compensation

The **tbkTCAutoZero** function will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation when making thermocouple measurements. 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.

```
int tbkTCAutoZero ( )
```

uint zero	If non-zero, will enable auto zero compensation in the tbkTC... functions
-----------	---

Note: See the thermocouple linearization section for a description and example of automatic zero compensation for thermocouple measurement.

Note: The enhanced API commands do not work exactly like the standard API commands used in this chapter.

The TempBook software includes two groups of functions for obtaining thermocouple temperatures:

- Low-level data conversion functions provide thermocouple linearization for previously acquired ADC data. Functions include: `tbkTCSetup`, `tbkTCConvert`, `tbkTCSetupConvert`, `tbkTCAutoZero`.
- High-level thermocouple measurement functions provide combined scan sequencer setup, triggering, data collection, and linearization. Functions include: `tbkRdTemp`, `tbkRdTempN`, `tbkRdTempScan`, `tbkRdTempScanN`.

Both of these function groups support types J, K, T, E, N28, N14, S, R and B thermocouples. The TempBook accepts thermocouple attachment on differential input channels 0 through 7. In addition, the TempBook provides a cold-junction compensation circuit on channel 16 and a permanently shorted input on channel 18 for performing zero compensation.

Two software techniques can be used to increase the measurement accuracy: software calibration and zero compensation. Software calibration uses gain and offset calibration constants, unique to each unit, to compensate for inherent errors. Zero compensation is a method by which offset voltages in the input amplifier stages can be removed at run-time. This is done by measuring a shorted channel at the same gain as the signal measurement to find the offset, and subtracting this from the actual reading. Both of these methods are described in the *Zero Compensation and Calibration* chapter.

The thermocouple linearization functions have a special auto-zero compensation feature that will perform zero compensation on the raw thermocouple data before linearizing. This auto-zero feature is enabled by default but can be disabled using the `tbkTCAutoZero` function.

Low-Level Thermocouple Data Conversion Functions

The low-level thermocouple linearization functions are designed to convert ADC data which was collected in a specific scan sequence format.

- When not using the auto-zero feature, the scan sequence should consist of a CJC reading followed by thermocouple readings. If different thermocouple types are being read, the readings should be grouped by type with a CJC reading preceding each group. The thermocouple linearization functions must then be called once for each thermocouple type.
- When using the auto-zero feature, the scan sequence should consist of 2 shorted channel reading followed by the CJC and thermocouple readings. The first shorted channel reading should be taken at the CJC gain and the second at the thermocouple gain. If different thermocouple types are being read, the reading should be grouped by thermocouple type with two shorted channel and one CJC readings preceding each group. The thermocouple linearization functions must then be called once for each thermocouple type.

The scan is not restricted to thermocouple measurements. The scan may include other signals such as voltage or digital input. The linearization functions will ignore this other data within the scan group.

The gain settings for the CJC and thermocouple types depend on the bipolar/unipolar setting of the TempBook as specified in the table below. **Note:** Unipolar operations are not recommended for thermocouple measurement unless the temperatures to be measured are guaranteed to be greater than the TempBook temperature.

TempBook Gain Codes				
Type	Unipolar Gain Code	Unipolar Gain	Bipolar Gain Code	Bipolar Gain
CJC	TbkUniCJC	50	TbkBiCJC	100
J	TbkUniTypeJ	100	TbkBiTypeJ	200
K	TbkUniTypeK	100	TbkBiTypeK	200
T	TbkUniTypeT	200	TbkBiTypeT	200
E	TbkUniTypeE	50	TbkBiTypeE	100
N28	TbkUniTypeN28	100	TbkBiTypeN28	200
N14	TbkUniTypeN14	100	TbkBiTypeN14	200
S	TbkUniTypeS	200	TbkBiTypeS	200
R	TbkUniTypeR	200	TbkBiTypeR	200
B	TbkUniTypeB	200	TbkBiTypeB	200

The low-level temperature conversion functions take data from one or more scans from the TempBook. They examine the CJC and thermocouple readings within that scan and, after optional averaging, convert them to temperatures which are stored as output. For example, assume the readings in the table at right.

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

This shows that the first two readings of each scan are a CJC zero reading and a TC zero reading. The third reading is from the CJC, and the remaining three readings are from three type J thermocouples. If the auto-zero feature is not used, the first two readings will be ignored. Otherwise, they will be used to remove any offset errors in the CJC and thermocouple reading before proceeding. When not using averaging, the CJC readings are combined with the thermocouple readings to produce one temperature result for each thermocouple reading. Reducing the 24 original readings to 12 temperatures (see table at right).

Scan	Result		
	0	1	2
1	J1a	J1b	J1c
2	J2a	J2b	J2c
3	J3a	J3b	J3c
4	J4a	J4b	J4c

The conversion process is divided into two steps: setup and conversion. The setup step describes the characteristics of the temperature measurement, and the conversion step actually converts the data from raw readings to temperatures. All of the functions return error codes which are defined in TempBook.H which includes the function prototypes and definitions of the thermocouple type codes.

The setup function is **tbkTCSetup**. The C-language version of **tbkTCSetup** is very similar to that of the other programming languages and is described below. **Note:** **uint** is short for **unsigned int**, and **uchar** is short for **unsigned char**. Further details in *Command Reference* chapter.

```
tbkTCSetup( uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar
bipolar, uint avg)
```

uint nscan	The number of readings in a single scan of TempBook data. The tbkTC 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 nscan-1. Each scan of temperature data must include a reading of the CJC. The CJC readings must be taken with the appropriate unipolar or bipolar CJC gain setting. Valid range: 0 to nscan-2 with no zero compensation; 2 to nscan-2 with zero compensation.
uint ntc	The number of thermocouple readings 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: J, K, T, E, N28, N14, S, R, or B. Valid range: One of the predefined values: TbkTCTypeJ , TbkTCTypeK , TbkTCTypeT , TbkTCTypeE , TbkTCTypeN28 , TbkTCTypeN14 , TbkTCTypeS , TbkTCTypeR or TbkTCTypeB .
uchar bipolar	Non-zero if the TempBook is configured for bipolar readings.
uint avg	The type of averaging to be performed. Valid range: any unsigned integer. Note: Since the thermocouple voltage may be small compared to the ambient electrical noise, averaging may sometimes 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. Each scan's 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 next table.

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	$(1D+1E+1F)/3$	$(2D+2E+2F)/3$

The conversion function is **tbkTCConvert**: (Note: Further details in *Command Reference* chapter.)

tbkTCConvert(uint *counts, uint scans, int *temp, uint ntemp)

uint *counts	A array of one or more scans of raw data as received from the TempBook. The ADC data bits are in the 12 most significant bits of the 16-bit integers. 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	The converted temperature results. The integer values are 10 times the temperatures in degrees C; e.g., 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 temp array. This is checked by the functions to avoid writing past the end of the temp array. Valid range: If avg is 0, then ntc or greater. If avg is non-zero, then scans * ntc or greater.

For convenience both setup and conversion can be performed at once by **tbkTCSetupConvert**: (Note: Further details in *Command Reference* chapter.)

tbkTCSetupConvert(uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg, uint *counts, uint scans, int *temp, uint ntemp)

The auto-zero feature can be enabled disabled using the **tbkTCAutoZero** function: (Note: Further details in *Command Reference* chapter.)

tbkTCAutoZero (uint zero)

uint zero	Non-zero to enable auto-zeroing; zero (0) to disable auto-zeroing.
-----------	---

Note: see the tempex2 example programs in the individual language support chapters fore examples of using these functions.

High-Level Thermocouple Measurement Functions

The high-level thermocouple measurement functions are designed to provide combined scan sequencer setup, triggering, data collection, and linearization. There are 4 functions in this group:

tbkRdTemp	Read a single thermocouple channel once.
tbkRdTempN	Read a single thermocouple channel multiple times.
tbkRdTempScan	Read a range of thermocouple channels once.
tbkRdTempScanN	Read a range of thermocouple channels multiple times.

Note: see tempex1 sample programs using these functions in the individual language-support chapters.

Single-Channel Measurement (tbkRdTemp)

The **tbkRdTemp** function uses software triggering to immediately acquire one sample from the specified analog input channel. This function also collects CJC and shorted channel readings for linearization and zero compensation. The CJC and thermocouple readings are then zero compensated and the thermocouple reading is linearized. The converted temperature is placed in a variable supplied by the calling program. (Note: Further details in *Command Reference* chapter.)

tbkRdTemp (uint chan, uint tcType, int *temp)

uint chan	The channel number to which the thermocouple is attached.
uint tcType	The type of thermocouple attached.
int*temp	A variable in which to store the measured temperature

Multiple Measurements from a Single Channel (tbkRdTempN)

The **tbkRdTempN** function uses pacer clock triggering to acquire multiple samples from the specified analog input channel and CJC and shorted channel readings for linearization and zero compensation. The CJC and thermocouple readings are zero compensated and the thermocouple readings are linearized. The converted temperatures are placed in an array supplied by the calling program. If block averaging is used, a single temperature is returned; otherwise, a number of temperatures equal to the number of scans specified are returned. (**Note:** Further details in *Command Reference* chapter.)

```
tbkRdTempN( uint chan, uchar tcType, uint count, int * temp, uint * buf,
float freq, uint avg )
```

uint chan	The channel number to which the thermocouple is attached.
uint tcType	The type of thermocouple attached.
uint count	The number of scans to read.
int *temp	A variable in which to store the measured temperature.
uint* buf	An array for the temporary storage of raw scan data (must be at least 4*count in length).
float freq	The scan interval frequency.
uint avg	Type of averaging to be used. 0 - block averaging 1 - no averaging 2 - moving averaging

Multiple-Channel Measurement (tbkRdTempScan)

The **tbkRdTempScan** function uses software triggering to immediately acquire one sample from the specified range of analog input channels. All of these channels must be of the same thermocouple type,. This function also collects CJC and shorted channel readings for linearization and zero compensation. The CJC and thermocouple readings are then zero compensated and the thermocouple readings are linearized. The converted temperatures are then placed in an array supplied by the calling program. (**Note:** Further details in *Command Reference* chapter.)

```
tbkRdTempScan(uint startChan, uint endChan, uchar tcType, int * temp)
```

uint startchan	The starting channel number of the range of thermocouple channels to read.
uint endChan	The ending channel number of the range of thermocouple channels to read.
uint tcType	The type of thermocouple attached.
int *temp	An array in which to store the measured temperatures (must be at least endChan - startChan + 1 in length).

Multiple Measurements from Multiple Channels (tbkRdTempScanN)

The **tbkRdTempScanN** function uses pacer clock triggering to acquire multiple samples from the specified range of analog input channels. All of these channels must be of the same thermocouple type. This function also collects CJC and shorted channel readings for linearization and zero compensation. The CJC and thermocouple readings are then zero compensated and the thermocouple readings are linearized. The converted temperatures are placed in an array supplied by the calling program. If block averaging is specified, then a single temperature is returned for each channel; otherwise, a number of temperatures equal to the number of scans time the number of channels specified are returned. (**Note:** Further details in *Command Reference* chapter.)

```
tbkRdTempScanN(uint startChan, uint endChan, uchar tcType, uint count, int *
temp, uint * buf, float freq, uint avg)
```

uint startchan	The starting channel number of the range of thermocouple channels to read.
uint endChan	The ending channel number of the range of thermocouple channels to read.
uint tcType	The type of thermocouple attached.
uint count	The number of scans to read.
int *temp	A variable in which to store the measured temperature.
uint *buf	An array for the temporary storage of raw scan data (must be at least [endChan - startChan + 4] *count in length).
float freq	The scan interval frequency.
uint avg	Type of averaging to be used. 0 - block averaging 1 - no averaging 2 - moving averaging

Overview

The first part of this chapter describes the TempBook/66 driver commands (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 as defined in the driver header files. Then, the prototype commands are described in alphabetical order as indexed below. At the end of the chapter (beginning on page 9-32), several reference tables define parameters for: A/D Channel Descriptions, A/D Gain Definitions, A/D Trigger Source Definitions, Pre-Trigger Functions, Thermocouple Types, and the API Error Codes.

These TempBook software commands are described on the following pages:

Function	Description	Page
High- and Low-Level A/D Functions		
tbkConfCntr0	Configure the counter 0 mode	9-5
tbkGetBackStat	Read the status of a background A/D transfer	9-6
tbkGetFreq	Read the current pacer clock frequency	9-6
tbkGetScan	Read the current scan configuration	9-7
tbkRd	Configure an A/D acquisition and read one sample from a channel	9-8
tbkRdCntr0	Read the current value of the counter 0	9-9
tbkRdFore	Read a single A/D sample and increment the channel mux	9-10
tbkRdN	Configure an A/D acquisition and read multiple scans from a channel	9-10
tbkRdNBack	Read count A/D scans in the background using interrupts	9-11
tbkRdNBackPreT	Reads multiple A/D scans, initiated by <code>tbkAdcSetTrigPreT</code> command, in the background	9-12
tbkRdNFore	Read count A/D samples in the foreground (polled mode)	9-12
tbkRdNForePreT	Read multiple A/D scans, initiated by <code>tbkAdcSetTrigPreT</code> command, in the foreground	9-13
tbkRdNForePreTWait	Read multiple A/D scans, initiated by <code>tbkAdcSetTrigPreT</code> command, in the foreground without returning until the acquisition completes	9-14
tbkRdScan	Configure an A/D acquisition and read one scan	9-14
tbkRdScanN	Configure an A/D acquisition and read multiple scans	9-15
tbkRdTemp	Take a single thermocouple reading from the given analog input channel	9-15
tbkRdTempN	Take multiple thermocouple readings from the given analog input channel	9-16
tbkSetMode	Configure gain amp single-/differential and polarity modes	9-21
tbkRdTempScan	Take thermocouple readings from analog input channels 'startChan' through 'endChan'	9-17
tbkRdTempScanN	Take multiple thermocouple readings from analog input channels 'startChan' through 'endChan'	9-18
tbkSetClk	Set the pacer clock counters	9-20
tbkSetFreq	Configure the pacer clock frequency in Hz	9-21
tbkSetMux	Configure a scan specifying start and end channels	9-22
tbkSetScan	Configure up to 512 channels making up an A/D or HS digital input scan	9-24
tbkSetTrig	Configure an A/D trigger	9-25
tbkSetTrigPreT	Set the trigger of analog level triggering & initiates the collection of pre-trigger data acquisition	9-26
tbkSoftTrig	Send a software trigger command to the TempBook	9-27
tbkStopBack	Stop a background A/D transfer	9-27
tbkWtCntr0	Write a value to counter 0	9-30
Digital I/O Functions		
tbkRdBit	Read a bit on a digital input port	9-9
tbkRdByte	Read a byte from a digital input port	9-9
tbkWtBit	Program a bit on a digital output port	9-29
tbkWtByte	Output a byte to a digital output port	9-30
Thermocouple Functions		
tbkTCConvert	Convert raw A/D readings to temperature readings	9-28
tbkTCSetup	Set up parameters for subsequent temperature conversions	9-28
tbkTCSetupConvert	Set up and convert raw A/D readings into temperature readings	9-29

Software Calibration and Zero Compensation Functions		
tbkTCAutoZero	Configure the thermocouple linearization functions to automatically perform zero compensation	9-27
tbkCalConvert	Perform the actual calibration of one or more scans	9-3
tbkCalSetup	Configure the order and type of data to be calibrated	9-3
tbkCalSetupConvert	Perform both the setup and convert steps with one call	9-4
tbkReadCalFile	Read all the calibration constants from the specified file	9-19
tbkTCConvert	Performs zero compensation on one or more scans	9-28
tbkTCSetup	Configure data for zero compensation	9-28
tbkTCSetupConvert	Perform both the setup and convert steps with one call	9-29
General Functions		
tbkClose	End communication with the TempBook	9-4
tbkGetProtocol	Returns the current parallel port communications protocol	9-7
tbkInit	Initialize a single TempBook	9-8
tbkSelectPort	Select an initialized TempBook/66 as the current TempBook	9-19
tbkSetErrHandler	Sets the handler that will be executed upon an error condition	9-20
tbkSetProtocol	Specifies the type of parallel-port implementation and protocol available on the computer	9-23
tbkDefaultHandler		9-6

Commands in Alphabetical Order

The following pages give the details for each TempBook/66 command listed in alphabetical order. Each section starts with a table that summarizes the main features of the command. An explanation follows (and in some cases a programming example or related information).

tbkCalConvert

DLL Function	<code>tbkCalConvert(uint *counts, uint scans);</code>
C	<code>tbkCalConvert(unsigned *counts, unsigned scans);</code>
QuickBASIC	<code>BtbkCalConvert%(counts%, ByVal scans%)</code>
Visual Basic	<code>VBtbkCalConvert% (counts%(), ByVal scans%)</code>
Turbo Pascal	<code>ftbkCalConvert(counts:WordP; scans:word):integer;</code>
Parameters	
<code>uint *counts</code>	The raw data from one or more scans.
<code>uint scans</code>	The number of scans of raw data in the counts array.
Returns	<code>TerrZCInvParam</code> - Invalid parameter value <code>TerrNoError</code> - No error
See Also	<code>tbkReadCalFile</code> , <code>tbkCalSetup</code> , <code>tbkCalSetupConvert</code>
Program References	None

The **tbkCalConvert** function performs the actual calibration of one or more scans according to the previously called **tbkCalSetup** function. This function will modify the array of data passed to it. See the Software Calibration and Zero Compensation chapter for a complete description of calibration.

tbkCalSetup

DLL Function	<code>tbkCalSetup(uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint bipolar, uint noOffset);</code>
C	<code>tbkCalSetup(unsigned nscan, unsigned readingsPos, unsigned nReadings, unsigned chanType, unsigned chanGain, unsigned bipolar, unsigned noOffset);</code>
QuickBASIC	<code>BtbkCalSetup%(ByVal nscan%, ByVal readingsPos%, ByVal nReadings%, ByVal chanType%, ByVal chanGain%, ByVal bipolar%, ByVal noOffset%)</code>
Visual Basic	<code>VBtbkCalSetup% (ByVal nscan%, ByVal readingsPos%, ByVal nReadings%, ByVal chanType%, ByVal chanGain%, ByVal bipolar%, ByVal noOffset%)</code>
Turbo Pascal	<code>tbkCalSetup(nscan:word;readingsPos:byte;nReadings:byte;chanType:word;chanGain:word;:word;noOffset:word):integer;</code>
Parameters	
<code>uint nscan</code>	The number of readings in a single scan.
<code>uint readingsPos</code>	The position of the readings to be calibrated within the scan.
<code>uint nReadings</code>	The number of readings to calibrate.
<code>uint chanType</code>	The type of channel from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel and 0 when calibrating any other channel.
<code>uint chanGain</code>	The gain setting of the channels to be calibrated.
<code>uint bipolar</code>	Non-zero if the readings are bipolar.
<code>uint noOffset</code>	If non-zero, the offset cal constant will not be used to calibrate the readings.
Returns	<code>TerrZCInvParam</code> - Invalid parameter value <code>TerrNoError</code> - No error
See Also	<code>tbkReadCalFile</code> , <code>tbkCalConvert</code> , <code>tbkCalSetupConvert</code>
Program References	None

The **tbkCalSetup** function will configure the order and type of data to be calibrated. This function requires all data to be calibrated to be from 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. See the Software Calibration and Zero Compensation chapter for a complete description of calibration.

tbkCalSetupConvert

DLL Function	tbkCalSetupConvert(uint nscan, uint readingsPos, uint nReadings, uint chanType, uint chanGain, uint bipolar, uint noOffset, uint *counts, uint scans);
C	tbkCalSetupConvert(unsigned nscan, unsigned readingsPos, unsigned nReadings, unsigned chanType, unsigned chanGain, unsigned bipolar, unsigned noOffset, unsigned *counts, unsigned scans);
QuickBASIC	BtbkCalSetupConvert% (ByVal nscan%, ByVal readingsPos%, ByVal nReadings%, ByVal chanType%, ByVal chanGain%, ByVal bipolar%, ByVal noOffset%, counts%, ByVal scans%)
Visual Basic	VBtbkCalSetupConvert% (ByVal nscan%, ByVal readingsPos%, ByVal nReadings%, ByVal chanType%, ByVal chanGain%, ByVal bipolar%, ByVal noOffset%, counts%(), ByVal scans%)
Turbo Pascal	tbkCalSetupConvert(nscan:word;readingsPos:byte;nReadings:byte;chanType:word;chanGain:word;bipolar:word;noOffset:word;counts:WordP;scans:word):integer;
Parameters	
uint nscan	The number of readings in a single scan.
uint readingsPos	The position of the readings to be calibrated within the scan.
uint nReadings	The number of readings to calibrate.
uint chanType	The type of channel from which the readings to be calibrated are read. This should be set to 1 when calibrating a CJC channel and 0 when calibrating any other channel.
uint chanGain	The gain setting of the channels to be calibrated.
uint bipolar	Non-zero if the readings are bipolar.
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	TerrZCInvParam - Invalid parameter value TerrNoError - No error
See Also	tbkReadCalFile, tbkCalSetup, tbkCalConvert
Program References	None

For convenience, both the setup and convert steps can be performed with one call to **tbkCalSetupConvert**. This is useful when the calibration needs to be performed multiple times because data was read at different gains or polarities. See the Software Calibration and Zero Compensation chapter for a complete description of calibration.

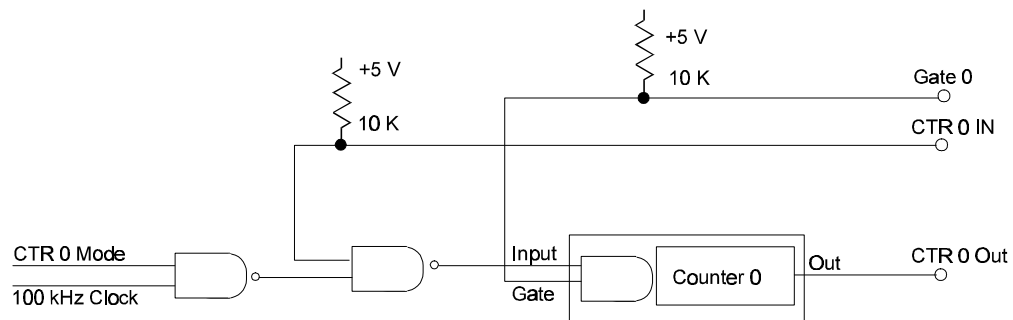
tbkClose

DLL Function	int tbkClose(void)
C	tbkClose(void);
QuickBASIC	BtbkClose% ()
Visual Basic	VBtbkClose% ()
Turbo Pascal	tbkClose:integer;
Parameters	None
Returns	TerrNoError - No error
See Also	tbkInit
Program References	

tbkClose is used to end communications with the TempBook/66. If **tbkClose** is called, **tbkInit** must be called before calling any other function.

tbkConfCntr0

DLL Function	<code>int tbkConfCntr0(uchar config);</code>																					
C	<code>tbkConfCntr0(unsigned char config);</code>																					
QuickBASIC	<code>BtbkConfCntr0% (ByVal config%)</code>																					
Visual Basic	<code>VBtbkConfCntr0% (ByVal config%)</code>																					
Turbo Pascal	<code>tbkConfCntr0(config:byte):integer;</code>																					
Parameters																						
uchar config	The configuration of Counter 0 (see table below for definitions)																					
	<table border="1"> <thead> <tr> <th>Description</th> <th>Value</th> <th>Note</th> </tr> </thead> <tbody> <tr> <td>Tc0cHighTermCnt</td> <td>0x30</td> <td>High on terminal count</td> </tr> <tr> <td>Tc0cOneShot</td> <td>0x32</td> <td>Hardware retriggerable one-shot</td> </tr> <tr> <td>Tc0cDivByNCtr</td> <td>0x34</td> <td>Rate Generator</td> </tr> <tr> <td>Tc0cSquareWave</td> <td>0x36</td> <td>Square wave</td> </tr> <tr> <td>Tc0cSoftTrigStrobe</td> <td>0x38</td> <td>Software triggered strobe</td> </tr> <tr> <td>Tc0cHardTrigStrobe</td> <td>0x3A</td> <td>Hardware triggered strobe</td> </tr> </tbody> </table>	Description	Value	Note	Tc0cHighTermCnt	0x30	High on terminal count	Tc0cOneShot	0x32	Hardware retriggerable one-shot	Tc0cDivByNCtr	0x34	Rate Generator	Tc0cSquareWave	0x36	Square wave	Tc0cSoftTrigStrobe	0x38	Software triggered strobe	Tc0cHardTrigStrobe	0x3A	Hardware triggered strobe
Description	Value	Note																				
Tc0cHighTermCnt	0x30	High on terminal count																				
Tc0cOneShot	0x32	Hardware retriggerable one-shot																				
Tc0cDivByNCtr	0x34	Rate Generator																				
Tc0cSquareWave	0x36	Square wave																				
Tc0cSoftTrigStrobe	0x38	Software triggered strobe																				
Tc0cHardTrigStrobe	0x3A	Hardware triggered strobe																				
Returns	TerrNoError - No error																					
See Also	tbkWtCntr0, tbkRdCntr0, tbkSetTrig																					
Program References	None																					



tbkConfCntr0 programs the control register of Counter 0 in one of six 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 **tbkSetTrig** command.

Mode 0, high on terminal count, is typically used to count events. After the initial count value (see **tbkWtCntr0**) is set, the counter will decrement on each pulse of the Counter 0 input. The count value at any time can be read using **tbkRdCntr0**. Counter 0 output (pin 2 of P1), which is initially low, will go high when the counter decrements to 0.

Mode 1, hardware retriggerable one-shot, is used to generate a pulse following the occurrence of a rising edge of the Counter 0 gate. The output, which is initially high, will go low after the hardware trigger is received until the count decrements to 0.

Mode 2, rate generator, acts as 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.

Mode 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.

Mode 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.

Mode 5, hardware triggered strobe, is similar to mode 4 except the strobe is initiated by a hardware trigger (rising edge of Counter 0 gate).

tbkDefaultHandler

DLL Function	int tbkDefaultHandler(uchar tbkErrnum);
C	tbkDefaultHandler(int tbkErrnum);
QuickBASIC	BtbkDefaultHandler% (ByVal tbkErrnum%)
Visual Basic	VBtbkDefaultHandler% (tbkErrnum%)
Turbo Pascal	tbkDefaultHandler(tbkErrnum:integer);
Parameters	
tbkErrnum	The error code of the detected error.
Returns	Nothing
See Also	tbkSetErrHandler
Program References	None

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

tbkGetBackStat

DLL Function	int tbkGetBackStat(uchar *active, ulong *count);
C	tbkGetBackStat(unsigned char _far *active, unsigned long _far *count);
QuickBASIC	BtbkGetBackStat%(active%, count%)
Visual Basic	VBtbkGetBackStat% (active%, count%)
Turbo Pascal	tbkGetBackStat(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	TerrOverrun - Internal data buffer overrun TerrFIFOFull - ADC FIFO Overrun TerrNoError - No error
See Also	tbkRdNBack, tbkStopBack
Program References	None

tbkGetBackStat 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 **tbkRdNBack** function.

tbkGetBackStat can return two possible error codes. **TerrFIFOFull** is returned if the data FIFO in the TempBook/66 is filled before the user can read the data out. In which case, the data read may be invalid. If the **tbkRdNBack** is called with the cycle flag enabled, a **TerrOverrun** can be returned. This means that 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.

tbkGetFreq

DLL Function	int tbkGetFreq(float *freq);
C	tbkGetFreq(float _far *freq);
QuickBASIC	BtbkGetFreq% (freq!)
Visual Basic	VBtbkGetFreq% (freq!)
Turbo Pascal	tbkGetFreq(freq:FloatP):integer;
Parameters	
float *freq	A variable to hold the currently defined sampling frequency in Hz Valid values: 100000.0 - 0.0002
Returns	TerrNoError - No error
See Also	tbkSetFreq, tbkSetClk
Program References	None

tbkGetFreq reads the sampling frequency of the pacer clock. Note: **tbkGetFreq** assumes that the 100 kHz/1 MHz jumper is set to the default position of 1MHz.

tbkGetProtocol

DLL Function	int tbkGetProtocol(int *protocol)																		
C	tbkGetProtocol(int *protocol);																		
QuickBASIC	BtbkGetProtocol% (protocol%)																		
Visual Basic	VBtbkGetProtocol% (protocol%)																		
Turbo Pascal	tbkGetProtocol(protocol:DataP):integer;																		
Parameters																			
protocol	A pointer to a value that will be set to the current protocol chosen from the protocol codes listed below (additional EPP implementation codes may be described in the README file).																		
	<table border="1"> <thead> <tr> <th>Name</th> <th>Description</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>TbkProtocolNone</td> <td>TempBook/66Communication Disabled</td> <td>0</td> </tr> <tr> <td>TbkProtocol8</td> <td>8-bit I/O</td> <td>1</td> </tr> <tr> <td>TbkProtocol4</td> <td>4-bit I/O</td> <td>2</td> </tr> <tr> <td>TbkProtocol4FPort</td> <td>Far Point F/Port EPP Interface</td> <td>10</td> </tr> <tr> <td>TbkProtocolSL</td> <td>82360 SL EPP Interface</td> <td>20</td> </tr> </tbody> </table>	Name	Description	Value	TbkProtocolNone	TempBook/66Communication Disabled	0	TbkProtocol8	8-bit I/O	1	TbkProtocol4	4-bit I/O	2	TbkProtocol4FPort	Far Point F/Port EPP Interface	10	TbkProtocolSL	82360 SL EPP Interface	20
Name	Description	Value																	
TbkProtocolNone	TempBook/66Communication Disabled	0																	
TbkProtocol8	8-bit I/O	1																	
TbkProtocol4	4-bit I/O	2																	
TbkProtocol4FPort	Far Point F/Port EPP Interface	10																	
TbkProtocolSL	82360 SL EPP Interface	20																	
Returns	An error number, or 0 is no error.																		
See Also	tbkInit, tbkSetProtocol																		
Program References	None																		

tbkGetProtocol returns the current parallel port communications protocol. **tbkInit** initially sets the protocol set to either **TbkProtocol8** or **TbkProtocol4**, indicating either 8-bit or 4-bit standard parallel port protocol. **tbkSetProtocol** may be used to specify other protocols.

tbkGetScan

DLL Function	int tbkGetScan(uint *chans, uchar *gains, uchar *polarities, uint *count);
C	tbkGetScan(unsigned *chans, unsigned char *gains, unsigned char *polarity, unsigned *count);
QuickBASIC	BtbkGetScan% (chans%, gains%, polarity%, ByVal count%)
Visual Basic	VBtbkGetScan% (chans%(), gains%(), polarity%(), count%)
Turbo Pascal	tbkGetScan(chans:WordP; gains:ByteP; polarity:ByteP; count:word):integer;
Parameters	
uint *chans	An array to hold up to 512 channel numbers or 0 if the channel information is not desired. See table at end of chapter for valid values.
uchar *gains	An array to hold up to 512 gain values or 0 if the channel gain information is not desired. See table at end of chapter for valid values.
uchar polarity	Zero value causes TempBook to default to Unipolar mode. Non-zero value causes default Bipolar mode. All ADC conversions except those started with tbksetscan will use the default polarity.
uint count	A variable to hold the number of values returned in the chans and gains arrays.
Returns	TerrNoError - No error
See Also	tbkSetScan, tbkSetMux
Program References	None

tbkGetScan reads the current scan sequence consisting of up to 512 channels, gains and polarities.

tbkInit

DLL Function	<code>int tbkInit(uchar lptPort, uchar lptIntr);</code>										
C	<code>tbkInit(unsigned int lptPort, unsigned char lptIntr);</code>										
QuickBASIC	<code>BtbkInit% (ByVal lptPort%, ByVal lptIntr%)</code>										
Visual Basic	<code>VBtbkInit% (lptPort%, lptIntr%)</code>										
Turbo Pascal	<code>tbkInit(lptPort:byte; lptIntr:byte):integer;</code>										
Parameters											
<code>uchar lptIntr</code>	The LPT interrupt level (7 for LPT1)										
<code>uchar lptPort</code>	The LPT port number (See table below for definitions.)										
	<table border="1"> <thead> <tr> <th>Description</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>LPT1</td> <td>0x00</td> </tr> <tr> <td>LPT2</td> <td>0x01</td> </tr> <tr> <td>LPT3</td> <td>0x02</td> </tr> <tr> <td>LPT4</td> <td>0x03</td> </tr> </tbody> </table>	Description	Value	LPT1	0x00	LPT2	0x01	LPT3	0x02	LPT4	0x03
Description	Value										
LPT1	0x00										
LPT2	0x01										
LPT3	0x02										
LPT4	0x03										
Returns	TerrNotOnLine - No communication with TempBook TerrBadChannel - Invalid LPT channel TerrNoTempBook - No TempBook/66detected TerrNoError - No error										
See Also	<code>tbkSelectPort, tbkClose</code>										
Program References	None										

tbkInit is used to perform multiple functions: initialize subroutine library variables, establish communications with a TempBook unit, reset the TempBook hardware to power-on conditions, and select the TempBook as the current TempBook. **tbkInit** can be called to reinitialize the TempBook only after the **tbkClose** command is called to terminate communications with the TempBook.

tbkInit 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
- Reset the counter/timers

Note: **tbkInit** must be called before any other TempBook function

tbkRd

DLL Function	<code>int tbkRd(uint chan, uint *sample, uchar gain);</code>
C	<code>tbkRd(unsigned chan, unsigned *sample, unsigned char gain);</code>
QuickBASIC	<code>BtbkRd% (ByVal chan%, sample%, ByVal gain%)</code>
Visual Basic	<code>VBtbkRd% (chan%, sample%, gain%)</code>
Turbo Pascal	<code>tbkRd(chan:word; sample:WordP; gain:byte):integer;</code>
Parameters	
<code>uint chan</code>	A single channel number (see table at end of chapter for valid values).
<code>uint *sample</code>	A pointer to a value where an A/D sample is stored
<code>uchar gain</code>	The channel gain (see table at end of chapter for valid values)
Returns	TerrFIFOFull - Buffer Overrun TerrInvGain - Invalid gain TerrInvChan - Invalid channel TerrNoError - No Error
See Also	<code>tbkRdN, tbkSetMux, tbkSetTrig, tbkSoftTrig, tbkRdFore</code>
Program References	None

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

tbkRdBit

DLL Function	<code>int tbkRdBit(uchar bitNum, uchar *bitVal);</code>
C	<code>tbkRdBit(unsigned char bitNum, unsigned char *bitVal);</code>
QuickBASIC	<code>BtbkRdBit% (ByVal bitNum%, bitVal%)</code>
Visual Basic	<code>VBtbkRdBit% (bitNum%, bitVal%)</code>
Turbo Pascal	<code>tbkRdBit(bitNum:byte; bitVal:ByteP):integer;</code>
Parameters	
<code>uchar bitNum</code>	The bit number of the specified digital I/O port to read Valid values: 0 - 7
<code>uchar *bitVal</code>	A variable to hold the value of the specified bit (non-zero if asserted, 0 if unasserted)
Returns	<code>TerrInvBitNum</code> - Invalid bit number <code>TerrNoError</code> - No error
See Also	<code>tbkWtByte</code> , <code>tbkRdByte</code> , <code>tbkWtBit</code>
Program References	None

`tbkRdBit` reads the state of a single digital input bit.

tbkRdByte

DLL Function	<code>int tbkRdByte(uchar *byteVal);</code>
C	<code>tbkRdByte(unsigned char *byteVal);</code>
QuickBASIC	<code>BtbkRdByte%(digIn%)</code>
Visual Basic	<code>VBtbkRdByte% (digIn%)</code>
Turbo Pascal	<code>tbkRdByte(byteVal:DataP):integer;</code>
Parameters	
<code>uchar *byteVal</code>	A variable to hold the value of the digital input byte
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkWtByte</code> , <code>tbkWtBit</code> , <code>tbkRdBit</code>
Program References	None

`tbkRdByte` reads the 8-bit digital input byte.

tbkRdCntr0

DLL Function	<code>int tbkRdCntr0(uint *cntr0, uchar latch);</code>
C	<code>tbkRdCntr0(unsigned _far *cntr0, unsigned char mode);</code>
QuickBASIC	<code>BtbkRdCntr0% (cntr0%, ByVal latch%)</code>
Visual Basic	<code>VBtbkRdCntr0% (cntr0%, latch%)</code>
Turbo Pascal	<code>tbkRdCntr0(cntr0:WordP; mode:Byte):integer;</code>
Parameters	
<code>uint *cntr0</code>	The value read back from the Counter 0 hold register Valid values: 0 - 65535
<code>uchar latch</code>	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	<code>TerrNoError</code> - No error
See Also	<code>tbkConfCntr0</code> , <code>tbkWtCntr0</code>
Program References	None

`tbkRdCntr0` reads the hold register of counter 0. This function is normally used with mode 0 of counter 0 (see `tbkConfCntr0`) to read the current count value.

tbkRdFore

DLL Function	int tbkRdFore(uint *sample);
C	tbkRdFore(unsigned _far *sample);
QuickBASIC	BtbkRdFore% (sample%)
Visual Basic	VBtbkRdFore% (sample%)
Turbo Pascal	tbkRdFore(sample:WordP):integer;
Parameters	
uint *sample	A pointer to a value where an A/D sample is stored Valid values: (See tbkSetTag)
Returns	TerrFIFOFull - Buffer overrun TerrNoError - No error
See Also	tbkReadFIFO, tbkSetTag, tbkSetClk, tbkSetTrig, tbkSetScan
Program References	None

tbkRdFore will read one sample from the A/D data FIFO. This function, unlike the **tbkRd** function, 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, hanging the computer.

tbkRdN

DLL Function	int tbkRdN(uint chan, uint *buf, uint count, uchar trigger, uchar oneShot, float freq, uchar gain);
C	tbkRdN(unsigned chan, unsigned *buf, unsigned count, unsigned char trigger, unsigned char oneShot, float freq, unsigned char gain);
QuickBASIC	BtbkRdN% (ByVal chan%, buf%, ByVal count%, ByVal trigger%, ByVal oneShot%, ByVal freq!, ByVal gain%)
Visual Basic	VBtbkRdN% (chan%, buf%(), count%, trigger%, oneShot%, freq!, gain%)
Turbo Pascal	tbkRdN(chan:word; buf:WordP; count:word; trigger:byte; oneShot:byte; freq:real; gain:byte):integer;
Parameters	
unit chan	A single channel number (see table at end of chapter for valid values)
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 (see table at end of chapter for valid values)
uchar one shot	A flag that if non-zero enables one-shot trigger mode, otherwise enables continuous mode.
float freq	The sampling frequency in Hz (100000.0 to 0.0002)
uchar gain	The channel gain (see table at end of chapter for valid values)
Returns	TerrFIFOFull - Buffer overrun TerrInvGain - Invalid gain TerrIncChan - Invalid channel TerrInvTrigSource - Invalid trigger TerrInvLevel - Invalid level
See Also	tbkRd, tbkRdScan, tbkRdScanN, tbkRdNFore, tbkSetFreq, tbkSetMux, tbkSetClk, tbkSetTrig
Program References	None

tbkRdN is used to take multiple scans from a single A/D channel. This function will configure the pacer clock, arm the trigger and acquire 'count' scans from the specified A/D channel.

tbkRdNBack

DLL Function	<code>int tbkRdNBack(uint *buf, uint count, uchar cycle, uchar updateSingle);</code>
C	<code>tbkRdNBack(unsigned _far *buf, unsigned count, unsigned char cycle, unsigned char updateSingle);</code>
QuickBASIC	<code>BtbkRdNBack% (buf%, ByVal count%, ByVal cycle%, ByVal updateSingle%)</code>
Visual Basic	<code>VBtbkRdNBack% (buf%(), count%, cycle%, updateSingle%)</code>
Turbo Pascal	<code>tbkRdNBack(buf:WordP; count:word; cycle:byte; updateSingle:byte):integer;</code>
Parameters	
<code>uint *buf</code>	An array where the A/D scans will be placed
<code>uint count</code>	The number of scans to be taken Valid values: 1 - 32767
<code>uchar cycle</code>	A flag that if non-zero will enable continuous operation, or if 0 will disable it
<code>uchar updateSingle</code>	A flag that if non-zero will enable single scans to be read into buf or if 0 will enable buf to be updated in a block of 256 scans
Returns	<code>TerrMultBackXfer</code> - Background read already in progress <code>TerrNoError</code> - No error
See Also	<code>tbkGetBackStat</code> , <code>tbkBackStop</code> , <code>tbkSetTag</code> , <code>tbkSetClk</code> , <code>tbkSetTrig</code>
Program References	None

tbkRdNBack 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 **tbkGetBackStat** function or stop the transfer with the **tbkBackStop** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes that 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. This allows the user to read large amounts of data without calling **tbkRdNBack** multiple 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 indefinitely. In this mode, the user should get the total number of scans written into 'buf' using **tbkGetBackStat** 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.

Note: the Visual Basic chapter includes an example program which demonstrates how to use the cycle mode of **tbkRdNBack**.

The `updateSingle` flag allows the user to control whether the TempBook/66 updates 'buf' one sample at a time or in blocks of 256 scans. Enabling `updateSingle` allows the user to read A/D data during slow acquisitions as the data is acquired. Because the `updateSingle` flag is directly tied to the number of interrupts that will be generated on the computer, the flag should not be enabled if the acquisition rate is greater than roughly 500 scans per second (sampling rate * # 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 the computer does not hang.

tbkRdnBackPreT

DLL Function	<code>int tbkRdnBackPreT(uint *buf, uint count, uchar cycle);</code>
C	<code>tbkRdnBackPreT(unsigned int _far *buf, unsigned int count, unsigned char cycle);</code>
QuickBASIC	<code>BtbkRdnBackPreT% (buf%, ByVal count%, ByVal cycle%)</code>
Visual Basic	<code>VBtbkRdnBackPreT% (buf%(), count%, cycle%)</code>
Turbo Pascal	<code>tbkRdnBackPreT(buf:WordP; count:word; cycle:byte):integer;</code>
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	TerrMultBackXfer - Background read already in progress TerrNoError - No error
See Also	tbkGetBackStat , tbkBackStop , tbkSetTag , tbkSetTrigPreT
Program References	PRETEX3 (all languages)

tbkRdnBackPreT reads multiple A/D scans, initiated by the **tbkSetTrigPreT** 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 **tbkGetBackStat** function or stop the transfer with the **tbkBackStop** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes that the pre-trigger acquisition has already been setup using the **tbkSetTrigPreT** command.

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 **tbkRdnBackPreT** 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 **tbkGetBackStat** 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, a number of **tbkRdnBack** calls may be necessary to read all the data collected during the pre-trigger mode acquisition.

tbkRdnFore

DLL Function	<code>int tbkRdnFore(uint *buf, uint count);</code>
C	<code>tbkRdnFore(unsigned _far *buf, unsigned count);</code>
QuickBASIC	<code>BtbkRdnFore% (buf%, ByVal count%)</code>
Visual Basic	<code>VBtbkRdnFore% (buf%(), count%)</code>
Turbo Pascal	<code>tbkRdnFore(buf:WordP; count:word):integer;</code>
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	TerrFIFOFull - Buffer overrun TerrNoError - No error
See Also	tbkSetClk , tbkSetTrig
Program References	

tbkRdnFore reads multiple A/D scans in the foreground. Unlike **tbkRdnBack**, 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.

tbkRdNForePreT

DLL Function	<code>int tbkRdNForePreT(uint *buf, uint count, uint *retcount, uchar *active);</code>
C	<code>tbkRdNForePreT(unsigned int _far *buf, unsigned int count, unsigned int _far *retcount, unsigned char _far *active);</code>
QuickBASIC	<code>BtbkRdNForePreT% (buf%, ByVal count%, retcount%, active%)</code>
Visual Basic	<code>VBtbkRdNForePreT% (buf%(), count%, retcount%, active%)</code>
Turbo Pascal	<code>tbkRdNForePreT(buf:WordP; count:word; retcount: WordP; active:ByteP):integer;</code>
Parameters	
<code>uint *buf</code>	An array where the A/D samples will be placed
<code>uint count</code>	The number of scans to be taken Valid values: 1 - 32767
<code>uint *retcount</code>	Pointer to an integer representing the number of scans actually taken
<code>uchar *active</code>	Pointer to a flag indicating whether or not the pre-trigger acquisition is still active
Returns	<code>TerrFIFOFull</code> - Buffer overrun <code>TerrNoError</code> - No error
See Also	<code>tbkSetTrigPreT</code> , <code>tbkSetTag</code> , <code>tbkRdNForePreTWait</code> , <code>tbkRdNBackPreT</code>
Program References	PRETEX1 (all languages)

tbkRdNForePreT reads multiple A/D scans, initiated by the **tbkSetTrigPreT** command, in the foreground. Unlike the **tbkRdNBackPreT** 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.

This function may be called subsequent to configuring a pre-trigger acquisition using the **tbkSetTrigPreT** command. Once this command has been called, it will return only when one of two possible conditions are met: 1) The specified number of scans has been collected, or, 2) the trigger has been detected and the acquisition has completed. In the latter case, the returned 'active' flag will be 0 and the number of scans actually collected will be returned in 'retcount'.

Note: If the A/D converter has not been configured to acquire data, this function may wait indefinitely, hanging the computer.

tbkRdNForePreTWait

DLL Function	<code>int tbkRdNForePreTWait(uint *buf, uint count, uint *retcount);</code>
C	<code>tbkRdNForePreTWait(unsigned int _far *buf, unsigned int count, unsigned int _far *retcount);</code>
QuickBASIC	<code>BtbkRdNForePreTWait% (buf%, ByVal count%, retcount%)</code>
Visual Basic	<code>VBtbkRdNForePreTWait% (buf%(), count%, retcount%)</code>
Turbo Pascal	<code>tbkRdNForePreTWait(buf:WordP; count:word; retcount:WordP):integer;</code>
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	<code>TerrFIFOFull</code> - Buffer overrun <code>TerrNoError</code> - No error
See Also	<code>tbkSetTrigPreT</code> , <code>tbkRdNForePreT</code> , <code>tbkRdNBackPreT</code>
Program References	PRETEX2 (all languages)

tbkRdNForePreTWait reads multiple A/D scans, initiated by the **tbkSetTrigPreT** command, in the foreground. Unlike the **tbkRdNForePreT** 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 **tbkSetTrigPreT** command. Once this command has been called, it will return only when the trigger has been detected and the acquisition has completed. The amount specified in the 'count' parameter specifies the length of the supplied buffer in scans. Unlike **tbkAdcRdNForePreT** 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.

tbkRdScan

DLL Function	<code>int tbkRdScan(uint startChan, uint endChan, uint *buf, uchar gain);</code>
C	<code>tbkRdScan(unsigned startChan, unsigned endChan, unsigned * buf, unsigned char gain);</code>
QuickBASIC	<code>BtbkRdScan% (ByVal startChan%, ByVal endChan%, buf%, ByVal gain%)</code>
Visual Basic	<code>VBtbkRdScan% (startChan%, endChan%, buf%(), gain%)</code>
Turbo Pascal	<code>tbkRdScan(startChan:word; endChan:word; buf:WordP; gain:byte):integer;</code>
Parameters	
uint startChan	The starting channel of the scan group (see table at end of chapter for valid values)
uint endChan	The ending channel of the scan group (see table at end of chapter for valid values)
uint *buf	An array where the A/D scans will be placed
uchar gain	The channel gain (see table at end of chapter for valid values)
Returns	<code>TerrInvGain</code> - Invalid gain <code>TerrInvChan</code> - Invalid channel <code>TerrNoError</code> - No error
See Also	<code>tbkRd</code> , <code>tbkRdN</code> , <code>tbkRdNScan</code> , <code>tbkSetMux</code> , <code>tbkRdFore</code> , <code>tbkSetClk</code> , <code>tbkSetTrig</code>
Program References	

tbkRdScan reads a single sample from multiple channels. This function will use a software trigger to immediately trigger and acquire one scan consisting of each channel starting with 'startChan' and ending with 'endChan'.

tbkRdScanN

DLL Function	<code>int tbkRdScanN(uint startChan, uint endChan, uint *buf, uint count, uchar trigger, uchar oneShot, float freq, uchar gain);</code>
C	<code>tbkRdScanN(unsigned startChan, unsigned endChan, unsigned * buf, unsigned count, unsigned char trigger, unsigned char oneShot, float freq, unsigned char gain);</code>
QuickBASIC	<code>BtbkRdScanN% (ByVal startChan%, ByVal endChan%, buf%, ByVal count%, ByVal trigger%, ByVal oneShot%, ByVal freq!, ByVal gain%)</code>
Visual Basic	<code>VBtbkRdScanN% (startChan%, endChan%, buf%(), count%, trigger%, oneShot%, freq!, gain%)</code>
Turbo Pascal	<code>tbkRdScanN(startChan:word; endChan:word; buf:WordP; count:word; trigger:byte; oneShot:byte; freq:real; gain:byte):integer;</code>
Parameters	
<code>uint startChan</code>	The starting channel of the scan group (see table at end of chapter for valid values)
<code>uint endChan</code>	The ending channel of the scan group (see table at end of chapter for valid values)
<code>uint *buf</code>	An array where the A/D scans will be placed
<code>uint count</code>	The number of scans to be read Valid values: 1 - 65536
<code>uchar trigger</code>	The trigger source (see table at end of chapter for valid values)
<code>uchar oneShot</code>	A flag that if non-zero enables one-shot trigger mode
<code>float freq</code>	The sampling frequency in Hz Valid values: 100000.0 - 0.0002
<code>uchar gain</code>	The channel gain (see table at end of chapter for valid values)
Returns	<code>TerrInvGain</code> - Invalid gain <code>TerrInvChan</code> - Invalid channel <code>TerrInvTrigSource</code> - Invalid trigger <code>TerrInvLevel</code> - Invalid Level <code>TerrFIFOFull</code> - Buffer Overrun <code>TerrNoError</code> - No error
See Also	<code>tbkRd</code> , <code>tbkRdN</code> , <code>tbkRdScan</code> , <code>tbkRdNFore</code> , <code>tbkSetClk</code> , <code>tbkSetTrig</code>
Program References	

tbkRdScanN 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'.

tbkRdTemp

DLL Function	<code>tbkRdTemp(uint chan, uchar tcType, int * temp)</code>
C	<code>tbkRdTemp(unsigned chan, unsigned tcType, int * temp);</code>
QuickBASIC	<code>BtbkRdTemp% (ByVal chan%, ByVal tcType%, temp%)</code>
Visual Basic	<code>VBtbkRdTemp% (chan%, tcType%, temp%)</code>
Turbo Pascal	<code>tbkRdTemp(chan:word; tcType:word; temp:DataP):integer;</code>
Parameters	
<code>uint chan</code>	Single channel number (see table at end of chapter for valid values)
<code>uint tcType</code>	Thermocouple type (see table at end of chapter for valid values)
<code>int *temp</code>	Pointer to a value where the converted temperature is to be stored
Returns	<code>TerrInvChan</code> - Invalid Channel <code>TerrTCE_Type</code> - Invalid Thermocouple Type
See Also	<code>tbkTCSetup</code> , <code>tbkTCConvert</code> , <code>tbkTCSetupConvert</code>
Program References	TEMPEX1 (all languages)

tbkRdTemp is used to take a single thermocouple reading from the given analog input channel. The reading will be zero corrected, span corrected, and linearized to yield a temperature reading in tenths of a degree Celsius. This function will use software triggering to immediately trigger and acquire one sample.

If a calibration constant file with the default name, "**tempbook.cal**", is not visible to the calling program then no span correction is performed.

tbkSetMode must be called before this function to configure the TempBook for differential operation in either unipolar or bipolar mode.

tbkRdTempN

DLL Function	tbkRdTempN(uint chan, uchar tcType, uint count, int * temp, uint * buf, float freq, uint avg)
C	tbkRdTempN(unsigned chan, unsigned tcType, unsigned count, int * temp, unsigned * buf, float freq, unsigned avg);
QuickBASIC	BtbkRdTempN% (ByVal chan%, ByVal tcType%, ByVal count%, temp%, buf%, ByVal freq!, ByVal avg%)
Visual Basic	VBtbkRdTempN% (chan%, tcType%, count%, temp%(), buf%(), freq!, avg%)
Turbo Pascal	tbkRdTempN(chan:word; tcType:word; count:word; temp:DataP; buf:WordP; freq:real; avg:word):integer;
Parameters	
uint chan	Single channel number (see table at end of chapter for valid values)
uint tcType	Thermocouple type (see table at end of chapter for valid values)
uint count	Number of scans to read.
int *temp	Pointer to a value where the converted temperature is to be stored
uint* buf	Pointer to an array where the raw ADC counts are to be stored.
float freq	Sampling frequency in Hz Valid values: 100000.0 - 0.0002
uint avg	Type of averaging to be used. 0 - block averaging 1 - no averaging 2 - moving averaging
Returns	TerrInvChan - Invalid Channel TerrTCE_Type - Invalid Thermocouple Type TerrInvCount - More than 1 scan specified with freq = 0
See Also	tbkRdTemp, tbkRdTempScan, tbkRdTempScanN, tbkTCSetup, tbkTCConvert, tbkTCSetupConvert
Program References	TEMPEX1 (all languages)

tbkRdTempN is used to take multiple thermocouple readings from the given analog input channel. The readings will be zero corrected, span corrected, and linearized to yield temperature reading(s) in tenths of a degree Celsius. This function will use pacer clock triggering to acquire samples at the rate defined in the parameter **freq**. The parameter **avg** is used to specify no, block, or moving averaging.

A pointer to a buffer array must be provided for storage of the raw ADC counts. The array dimension must be at least 4 * count.

If a calibration constant file with the default name “**tempbook.cal**” is not visible to the calling program, then no span correction is performed.

tbkSetMode must be called before this function to configure the TempBook for differential operation in either unipolar or bipolar mode.

tbkRdTempScan

DLL Function	tbkRdTempScan(uint startChan, uint endChan, uchar tcType, int * temp)
C	tbkRdTempScan(unsigned startChan, unsigned endChan, unsigned tcType, int * temp);
QuickBASIC	BtbkRdTempScan% (ByVal startChan%, ByVal endChan%, ByVal tcType%, temp%)
Visual Basic	VBtbkRdTempScan% (startChan%, endChan%, tcType%, temp%())
Turbo Pascal	tbkRdTempScan(startChan:word; endChan:word; tcType:word; temp:DataP):integer;
Parameters	
uint startchan	Single channel number (see table at end of chapter for valid values)
uint endChan	Ending channel of the scan group (see table at end of chapter for valid values)
uint tcType	Thermocouple type (see table at end of chapter for valid values)
int *temp	An array where the A/D scan will be placed.
Returns	TerrInvChan - Invalid Channel TerrTCE_Type - Invalid Thermocouple Type
See Also	tbkRdTemp, tbkRdTempN, tbkRdTempScanN, tbkTCSetup, tbkTCConvert, tbkTCSetupConvert
Program References	TEMPEX1 (all languages)

tbkRdTempScan is used to take thermocouple readings from analog input channels 'startChan' through 'endChan'. The readings will be zero corrected, span corrected, and linearized to yield temperature readings in tenths of a degree Celsius. This function will use software triggering to immediately trigger and acquire one scan.

If a calibration constant file with the default name "**tempbook.cal**" is not visible to the calling program, then no span correction is performed.

tbkSetMode must be called before this function to configure the TempBook for differential operation in either unipolar or bipolar mode.

tbkRdTempScanN

DLL Function	tbkRdTempScanN(uint startChan, uint endChan, uchar tcType, uint count, int * temp, uint * buf, float freq, uint avg)
C	tbkRdTempScanN(unsigned startChan, unsigned endChan, unsigned tcType, unsigned count, int * temp, unsigned * buf, float freq, unsigned avg);
QuickBASIC	BtbkRdTempScanN% (ByVal startChan%, ByVal endChan%, ByVal tcType%, ByVal count%, temp%, buf%, ByVal freq!, ByVal avg%)
Visual Basic	VBtbkRdTempScanN% (startChan%, endChan%, tcType%, count%, temp%(), buf%(), freq!, avg%)
Turbo Pascal	tbkRdTempScanN(startChan:word; endChan:word; tcType:word; count:word; temp:DataP; buf:WordP; freq:real; avg:word):integer;
Parameters	
uint startchan	Single channel number (see table at end of chapter for valid values)
uint endChan	Ending channel of the scan group (see table at end of chapter for valid values)
uint tcType	Thermocouple type (see table at end of chapter for valid values)
uint count	Number of scans to read.
int *temp	Pointer to a value where the converted temperatures are to be stored
uint *buf	Pointer to an array where the raw ADC counts are to be stored.
float freq	Sampling frequency in Hz Valid values: 100000.0 - 0.0002
uint avg	Type of averaging to be used.
Returns	TerrInvChan - Invalid Channel TerrTCE_Type - Invalid Thermocouple Type TerrInvCount - More than 1 scan specified with freq = 0
See Also	tbkRdTemp, tbkRdTempN, tbkRdTempScan, tbkTCSetup, tbkTCConvert, tbkTCSetupConvert
Program References	TEMPEX1 (all languages)

tbkRdTempScanN is used to take multiple thermocouple readings from analog input channels 'startChan' through 'endChan'. The readings will be zero-corrected, span-corrected, and linearized to yield temperature readings in tenths of a degree Celsius. This function will use pacer-clock triggering to acquire samples at the rate defined in the parameter **freq**. The parameter **avg** is used to specify no, block, or moving averaging.

A pointer to a buffer array must be provided for storage of the raw ADC counts. The array dimension must be at least count * (endChan - startChan + 4).

If a calibration constant file with the default name "tempbook.cal" is not visible to the calling program, then no span correction is performed.

tbkSetMode must be called before this function to configure the TempBook for differential operation in either unipolar or bipolar mode.

tbkReadCalFile

DLL Function	tbkReadCalFile(char *calfile);
C	tbkReadCalFile(char *calfile);
QuickBASIC	BtbkReadCalFile% (ByVal calfile\$)
Visual Basic	VBtbkReadCalFile% (ByVal calfile\$)
Turbo Pascal	tbkReadCalFile(calfile : string) : 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 TEMPBOOK.CAL will be read.
Returns	TerrNoError - No error TerrInvCalfile - Error occurred while opening or reading calibration file
See Also	tbkCalSetup, tbkCalConvert, tbkCalSetupConvert
Program References	None

tbkReadCalFile is the initialization function for reading in the calibration constants from the calibration text file. This function, which is 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 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 **tbkReadCalFile** function.

See the Software Calibration and Zero Compensation chapter for a complete description of calibration.

tbkSelectPort

DLL Function	int tbkSelectPort(uchar lptPort);	
C	tbkSelectPort(unsigned int lptPort);	
QuickBASIC	BtbkSelectPort% (ByVal lptPort%)	
Visual Basic	VBtbkSelectPort% (lptPort%)	
Turbo Pascal	tbkSelectPort(lptPort:byte):integer;	
Parameters		
uchar lptPort	The LPT port number (see table below for definitions.)	
	Description	Value
	LPT1	0x00
	LPT2	0x01
	LPT3	0x02
	LPT4	0x03
Returns	TerrNotOnLine - No communications with TempBook TerrBadChannel - Invalid LPT channel TerrNoTempBook - No TempBook/66 detected TerrNoError - No error	
See Also		
Program References		

tbkSelectPort selects an initialized TempBook. This function causes any subsequent function calls to be performed on this TempBook. Because **tbkInit** initializes then selects a TempBook, **tbkSelectPort** is only needed when using multiple TempBooks.

Note: **tbkInit** must be called with the corresponding LPT port before **tbkSelectPort** can select it.

tbkSetClk

DLL Function	<code>int tbkSetClk(uint ctr1, uint ctr2);</code>
C	<code>tbkSetClk(unsigned ctr1, unsigned ctr2);</code>
QuickBASIC	<code>BtbkSetClk% (ByVal ctr1%, ByVal ctr2%)</code>
Visual Basic	<code>VBtbkSetClk% (ctr1%, ctr2%)</code>
Turbo Pascal	<code>tbkSetClk(ctr1:word; ctr2:word):integer;</code>
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	TerrInvClock - Invalid clock TerrNoError - No error
See Also	<code>tbkSetFreq</code> , <code>tbkGetFreq</code>
Program References	

tbkSetClk 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 frequency is defined to be $xtal/ctr1*ctr2$ where $xtal$ is the frequency of the board crystal (either 1 MHz or 100 kHz).

tbkSetErrorHandler

DLL Function	<code>tbkSetErrorHandler(tbkSetErrorHandlerFTP tbkErrorHandler);</code>
C	<code>tbkSetErrorHandler(tbkErrorHandlerFPT tbkErrorHandler);</code>
QuickBASIC	<code>BtbkSetErrorHandler% (ByVal tbkErrorHandler&)</code>
Visual Basic	<code>VBtbkSetErrorHandler% (tbkErrorHandler%)</code>
Turbo Pascal	<code>tbkSetErrorHandler(tbkErrorHandler:ErrorFuncT):integer;</code>
Parameters	
tbkErrorHandler	This is a function that takes an integer (error code) and returns nothing, or NULL to disable.
Returns	TerrNoError - No error, or an error number
See Also	<code>tbkDefaultHandler</code>
Program References	

*** For C and Pascal Only - If the driver detects an error condition during its operation, it automatically calls a default system error handler. This command allows the user to supply an error handler that is automatically called when a system error is detected.

*** For Visual Basic and QuickBASIC - If the driver detects an error condition during its operation, it will pass the error code as the return value of each function. This command allows the user to set a BASIC error number which will be generated when an error occurs. The error can then be handled using the standard ONERROR feature of BASIC.

tbkSetFreq

DLL Function	<code>int tbkSetFreq(float freq);</code>
C	<code>tbkSetFreq(float freq);</code>
QuickBASIC	<code>BtbkSetFreq% (ByVal freq!)</code>
Visual Basic	<code>VBtbkSetFreq% (freq!)</code>
Turbo Pascal	<code>tbkSetFreq(freq:real):integer;</code>
Parameters	
<code>float freq</code>	The sampling frequency in Hz Valid values: 100000.0 - 0.0002
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkGetFreq</code> , <code>tbkSetClk</code>
Program References	None

tbkSetFreq calculates then sets the frequency of the pacer clock using the specified frequency 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. **tbkRdFreq** can be used to read the exact frequency setting of the pacer clock. **tbkSetClk** 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.

tbkSetMode

DLL Function	<code>tbkSetMode(uchar di_se, uchar polarity);</code>
C	<code>tbkSetMode(unsigned char di_se, unsigned char polarity);</code>
QuickBASIC	<code>BtbkSetMode% (ByVal di_se%, ByVal polarity%)</code>
Visual Basic	<code>VBtbkSetMode% (ByVal di_se%, ByVal polarity%)</code>
Turbo Pascal	<code>tbkSetMode(di_se:byte; polarity:byte):integer;</code>
Parameters	
<code>uchar di_se</code>	Zero value causes TempBook to go to single-ended mode (power-on default). Non-zero value causes differential mode.
<code>uchar polarity</code>	Zero value causes TempBook to default to Unipolar mode. Non-zero value causes default Bipolar mode. All ADC conversions except those started with <code>tbkSetScan</code> will use the default polarity.
Returns	<code>TerrNoError</code> - No error
See Also	
Program References	None

tbkSetMode 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 a pair of selected channels.

Polarity is unipolar or bipolar:

- Unipolar maximum voltage range is 0 to +10 V
- Bipolar maximum voltage range is -10 to +10 V.

tbkSetMux

DLL Function	<code>int tbkSetMux(uint startChan, uint endChan, uchar gain);</code>
C	<code>tbkSetMux(unsigned startChan, unsigned endChan, unsigned char gain);</code>
QuickBASIC	<code>BtbkSetMux% (ByVal startChan%, ByVal endChan%, ByVal gain%)</code>
Visual Basic	<code>VBtbkSetMux% (startChan%, endChan%, gain%)</code>
Turbo Pascal	<code>tbkSetMux(startChan:word; endChan:word; gain:byte):integer;</code>
Parameters	
<code>uint startChan</code>	The starting channel of the scan group (see table at end of chapter for valid values)
<code>uint endChan</code>	The ending channel of the scan group (see table at end of chapter for valid values)
<code>uchar gain</code>	The gain value for all channels (see table at end of chapter for valid values)
Returns	<code>TerrInvGain</code> - Invalid gain <code>TerrIncChan</code> - Invalid channel <code>TerrNoError</code> -No error
See Also	<code>tbkSetScan</code> , <code>tbkGetScan</code>
Program References	

tbkSetMux sets a simple scan sequence of local A/D channels from 'startChan' to 'endChan' all with the specified gain value. This provides a simple alternative to **tbkSetScan** if consecutive channels need to be acquired.

tbkSetProtocol

DLL Function	int tbkSetProtocol(int protocol)																								
C	tbkSetProtocol(int protocol);																								
QuickBASIC	BtbkSetProtocol% (ByVal protocol%)																								
Visual Basic	VBtbkSetProtocol% (protocol%)																								
Turbo Pascal	tbkSetProtocol(protocol:integer):integer;																								
Parameters																									
protocol	One of the predefined protocol codes listed below (additional protocol codes may be described in the README file).																								
	<table border="1"> <thead> <tr> <th>Name</th> <th>Description</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>TbkProtocol8</td> <td>8-bit I/O</td> <td>1</td> </tr> <tr> <td>TbkProtocol4</td> <td>4-bit I/O</td> <td>2</td> </tr> <tr> <td>TbkProtocolFPort</td> <td>Far Point F/Port EPP Interface</td> <td>10</td> </tr> <tr> <td>TbkProtocolSL</td> <td>82360 SL EPP Interface</td> <td>20</td> </tr> <tr> <td>TbkProtocolSMC666</td> <td>SMC 37C666 EPP mode</td> <td>30</td> </tr> <tr> <td>TbkProtocolEPPBIOS</td> <td>EPP bios mode</td> <td>40</td> </tr> <tr> <td>TbkProtocolPastEPP</td> <td>WBK20/21 Fast EPP mode</td> <td>50</td> </tr> </tbody> </table>	Name	Description	Value	TbkProtocol8	8-bit I/O	1	TbkProtocol4	4-bit I/O	2	TbkProtocolFPort	Far Point F/Port EPP Interface	10	TbkProtocolSL	82360 SL EPP Interface	20	TbkProtocolSMC666	SMC 37C666 EPP mode	30	TbkProtocolEPPBIOS	EPP bios mode	40	TbkProtocolPastEPP	WBK20/21 Fast EPP mode	50
Name	Description	Value																							
TbkProtocol8	8-bit I/O	1																							
TbkProtocol4	4-bit I/O	2																							
TbkProtocolFPort	Far Point F/Port EPP Interface	10																							
TbkProtocolSL	82360 SL EPP Interface	20																							
TbkProtocolSMC666	SMC 37C666 EPP mode	30																							
TbkProtocolEPPBIOS	EPP bios mode	40																							
TbkProtocolPastEPP	WBK20/21 Fast EPP mode	50																							
Returns	An error number, or 0 if no error.																								
See Also	tbkInit, tbkGetProtocol																								
Program References																									

tbkSetProtocol specifies to the TempBook/66 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 TempBook/66 to communicate using the specified protocol. Since establishing the protocol may affect the settings of the TempBook, **tbkSetProtocol** should only be invoked immediately after **tbkInit** has established communications with and reset the TempBook. Switching protocols during normal TempBook/66 operation is not recommended.

Two types of parallel port implementations are supported by the TempBook: standard and enhanced. Standard parallel ports, using the TempBook/66 manufacturer's proprietary protocols, are capable of receiving data either 4 or 8 bits at a time. When possible, 8-bit operation is preferred (it is much faster), 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 TempBook/66 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 TempBook/66 is initialized by **tbkInit**, it is initially configured for a standard parallel port protocol: either 8-bit, if possible, or the slower 4-bit protocol. After **tbkInit** has completed, **tbkSetProtocol** may be used to switch to another supported protocol.

If **tbkSetProtocol** 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, **tbkSetProtocol** will not return an error indication unless it is unable to establish any protocol.

In any case, **tbkGetProtocol** may be used to check the current operating protocol.

tbkSetScan

DLL Function	<code>int tbkSetScan(uint *chans, uchar *gains, uchar polarities, uint count);</code>
C	<code>tbkSetScan(unsigned *chans, unsigned char *gains, unsigned char *polarity, unsigned count);</code>
QuickBASIC	<code>BtbkSetScan% (chans%, gains%, polarity%, ByVal count%)</code>
Visual Basic	<code>VBtbkSetScan% (chans%(), gains%(), polarity%(), count%)</code>
Turbo Pascal	<code>tbkSetScan(chans:WordP; gains:ByteP; polarity:ByteP; count:word):integer;</code>
Parameters	
<code>uint *chans</code>	An array of up to 512 channel numbers (see table at end of chapter for valid values)
<code>uchar *gains</code>	An array of up to 512 gain values (see table at end of chapter for valid values)
<code>uchar polarities</code>	An array of up to 512 polarity values. Zero value causes TempBook/66 to select Unipolar mode. Non-zero values causes Bipolar mode.
<code>uint count</code>	The number of values in the chans and gains arrays Valid values: 1 - 512
Returns	<code>TerrNotCapable</code> - No high speed digital <code>TerrInvGain</code> - Invalid gain <code>TerrInvChan</code> - Invalid channel <code>TerrNoError</code> - No error
See Also	<code>tbkGetScan</code> , <code>tbkSetMux</code>
Program References	

tbkSetScan configures a scan sequence consisting of multiple channels, polarities 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. Channels can be entered multiple times at the same or different gain. The high-speed digital I/O port can also be included although its gain value will be ignored.

tbkSetTrig

DLL Function	<code>int tbkSetTrig(uchar trigger, uchar one shot, uchar ctr0 mode, uchar pacer Mode);</code>
C	<code>tbkSetTrig(unsigned char trigger, unsigned char oneShot, unsigned char ctr0mode, unsigned char pacerMode);</code>
QuickBASIC	<code>BtbkSetTrig% (ByVal trigger%, ByVal oneShot%, ByVal ctr0mode%, ByVal pacerMode%)</code>
Visual Basic	<code>VBtbkSetTrig% (ByVal trigger%, ByVal oneShot%, ByVal ctr0mode%, ByVal pacerMode%)</code>
Turbo Pascal	<code>tbkSetTrig(trigger:byte; oneShot:byte; ctr0Mode:byte; pacerMode:byte):integer;</code>
Parameters	
<code>uchar trigger</code>	The trigger source (see table at end of chapter for valid values)
<code>uchar one Shot</code>	A flag that if non-zero enables one-shot trigger mode, otherwise enables continuous mode
<code>ctr0mode</code>	A non-zero flag selects an internal 100 kHz clock to be the input to counter 0. If the flag is zero, only the external clock is the input to counter 0. See figure in <code>tbkConfCntr0</code> for detailed diagram.
<code>pacer Mode</code>	A flag that if zero, disables the external TTL Trigger 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	<code>TerrInvTrigSource</code> - Invalid trigger <code>TerrInvLevel</code> - Invalid level <code>TerrNoError</code> - No error
See Also	<code>tbkConfCntr0</code>
Program References	

tbkSetTrig sets and arms the trigger of the A/D converter. Eight trigger sources and several mode flags can be used to generate a wide variety of acquisitions. The **tbkSetTrig** command will stop any current acquisitions, empty the TempBook/66 of any data previously acquired and arm the TempBook/66 using the specified trigger source.

The pacer clock trigger source can be used to acquire data at a constant frequency. The sampling rate can be set using the **tbkSetClk** or **tbkSetFreq** functions. The one-shot flag has no meaning when using this trigger source.

The software trigger source allows the user to trigger the A/D from software using the **tbkSoftTrig** function. When the one-shot mode is enabled, a single scan will be initiated by the software trigger. In the continuous mode (one-shot disabled), sending a software trigger will cause the A/D converter to sample at the rate of the pacer clock.

An external TTL pulse 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 the trig input. The pulse will initiate a single scan in one-shot mode and a continuous acquisition at the pacer clock frequency in continuous mode.

Setting the counter 0 mode flag true will enable an onboard 100 kHz clock to be ANDed with the counter 0 input to produce the input to counter 0. If nothing is connected to counter 0 input, the line will float high essentially clocking counter 0 off the 100KHz clock. If this 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 to the trig input 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 the trig input. If it is zero, the pacer clock will be enabled.

tbkSetTrigPreT

DLL Function	int tbkSetTrigPreT(uchar trigger, uint channel, uint level, uint precount, uint postcount);
C	tbkSetTrigPreT(unsigned char source, unsigned int channel, unsigned int level, unsigned int preCount, unsigned int postCount);
QuickBASIC	BtbkSetTrigPreT% (ByVal source%, ByVal channel%, ByVal level%, ByVal preCount%, ByVal postCount%)
Visual Basic	VBtbkSetTrigPreT% (source%, channels%, level%, preCount%, postCount%)
Turbo Pascal	tbkSetTrigPreT(source:byte; channel:word; level:word; preCount:word; postCount:word):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 (0-4095)
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	TerrInvTrigSource - Invalid trigger TerrInvLevel - Invalid level TerrNoError - No error
See Also	tbkSetFreq, tbkSetClk, tbkRdNForePreT, tbkRdNForePreTWait, tbkRdNBackPreT
Program References	PRETEX1, PRETEX2, PRETEX3 (ALL LANGUAGES)

tbkSetTrigPreT sets the trigger for analog level triggering and initiates the collection of a pre-trigger data acquisition. The **tbkSetTrigPreT** command will stop any current acquisition, empty the TempBook/66 of any data previously acquired, arm the TempBook/66 using the specified analog level trigger source and will immediately begin the collection of the specified amount of pre-trigger data.

This command allows the configuration of a data acquisition that includes both pre-trigger and post-trigger data. The specified 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. This means that the specified pre-trigger amount represents the minimum amount of pre-trigger data which will actually be collected. The specified 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 **tbkSetClk** or **tbkSetFreq** commands.

The four analog trigger sources, rising or falling slope with either 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. It does not necessarily represent the actual physical channel number.

When setting up a pre-trigger acquisition, a specific command set must be used to retrieve the data. This command set includes **tbkRdNForePreT**, **tbkRdNForePreTWait** and **tbkRdNBackPreT**. For more information on these commands, refer to the command description for each specific command in this chapter.

tbkSoftTrig

DLL Function	<code>int tbkSoftTrig(void);</code>
C	<code>tbkSoftTrig(void);</code>
QuickBASIC	<code>BtbkSoftTrig% ()</code>
Visual Basic	<code>VBtbkSoftTrig% ()</code>
Turbo Pascal	<code>tbkSoftTrig:integer;</code>
Parameters	None
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkSetTrig</code>
Program References	

tbkSoftTrig is used to send a software trigger command to the TempBook. 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.

tbkStopBack

DLL Function	<code>int tbkStopBack(void);</code>
C	<code>tbkStopBack(void);</code>
QuickBASIC	<code>BtbkStopBack% ()</code>
Visual Basic	<code>VBtbkStopBack% ()</code>
Turbo Pascal	<code>tbkStopBack:integer;</code>
Parameters	None
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkRdNBack</code> , <code>tbkGetBackStat</code>
Program References	

tbkStopBack stops a background operation initiated by the `tbkRdNBack` function.

tbkTCAutoZero

DLL Function	<code>tbkTCAutoZero(uint zero);</code>
C	<code>tbkTCAutoZero(unsigned zero);</code>
QuickBASIC	<code>BtbkTCAutoZero% (ByVal zero%)</code>
Visual Basic	<code>VBtbkTCAutoZero% (ByVal zero%)</code>
Turbo Pascal	<code>tbkTCAutoZero(zero:word):integer;</code>
Parameters	
<code>uint zero</code>	If non-zero will enable auto zero compensation in the <code>tbkTC...</code> functions
Returns	<code>TerrZCInvParam</code> - Invalid parameter value <code>TerrNoError</code> - No error
See Also	<code>tbkZeroSetup</code> , <code>tbkZeroConvert</code> , <code>tbkZeroSetupConvert</code> , <code>tbkTCSetup</code> , <code>tbkTCConvert</code> , <code>tbkTcSetupConvert</code>
Program References	None

The **tbkTCAutoZero** function will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the TempBook. 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.

tbkTCConvert

DLL Function	tbkTcConvert (uint *counts, uint scans ,int *temp, uint ntemp);
C	tbkTCConvert(unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);
QuickBASIC	BtbkTCConvert% (counts%, ByVal scans%, temp%, ByVal ntemp%)
Visual Basic	VBtbkTCConvert% (counts%(), ByVal scans%, temp%(), ByVal ntemp%)
Turbo Pascal	tbkTCConvert(counts:WordP; scans:word; temp:DataP; 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
Returns	TerrTCE_NOSETUP - Setup was not called TerrTCE_PARAM - Param out of range TerrNoError - No Error
See Also	tbkTCSetup, tbkTCSetupConvert
Program References	None

tbkTCConvert takes raw A/D readings and converts them to temperature readings in tenths of degrees Celsius. Note: Total number of conversions (scan * chans/scan) must be less than 32 K.

tbkTCSetup

DLL Function	tbkTCSetup(uint nscan, uint cjcPosition, uint ntc , uint tcType, uchar bipolar, uint avg);
C	tbkTCSetup(unsigned nscan, unsigned cjcPosition, unsigned ntc, unsigned tcType, unsigned char bipolar, unsigned avg);
QuickBASIC	BtbkTCSetup% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%)
Visual Basic	VBtbkTCSetup% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%)
Turbo Pascal	tbkTCSetup(nscan, cjcPosition, ntc, tcType:word; bipolar:byte; avg:word):integer;
Parameters	
uint nscan	Number of readings in a scan. Valid range: 1- 512
uint cjcPosition	Position of CJC reading within a scan. Valid range: 0 - (nscan-1) 2 -(nscan-1), if auto-zeroing is used.
uint ntc	Number of thermocouples immediately following CJC. Valid range: 1 - (nscan-cjcposition-1)
uint tcType	Type of thermocouple (see table at end of chapter for valid types)
uint bipolar	Zero for unipolar, non-zero for bipolar.
uint avg	Type of averaging to be used.
Returns	TerrTCE_PARAM - Parameter out of range TerrTCE_TYPE - Invalid thermocouple type TerrNoError - No Error
See Also	tbkTCConvert, tbkTCSetupConvert
Program References	None

tbkTCSetup sets up parameters for subsequent temperature conversions.

tbkTCSetupConvert

DLL Function	<code>int tbkTCSetupConvert(uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg, uint *counts, uint scans, int *temp,);</code>
C	<code>tbkTCSetupConvert(unsigned nscan, unsigned cjcPosition, unsigned ntc, unsigned tcType, unsigned char bipolar, unsigned avg, unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);</code>
QuickBASIC	<code>BtbkTCSetupConvert% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%, counts%, ByVal scans%, temp%, ByVal ntemp%)</code>
Visual Basic	<code>VBtbkTCSetupConvert% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%, counts%(), ByVal scans%, temp%(), ByVal ntemp%)</code>
Turbo Pascal	<code>tbkTCSetupConvert(nscan, cjcPosition, ntc, tcType:word; bipolar:byte; avg:word; counts:WordP; scans:word; temp:DataP; ntemp:word):integer;</code>
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.
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 (see table at end of chapter for valid types)
uint bipolar	Non-zero if the TempBook/66 is configured for bipolar readings.
uint avg	The type of averaging to be performed: block, none or moving.
uint *counts	The raw data 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	TerrTCE_PARAM - Parameter out of range TerrTCE_TYPE - Invalid thermocouple type TerrNoError - No Error
See Also	<code>tbkTCSetup</code> , <code>tbkTCConvert</code>
Program References	None

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

tbkWtBit

DLL Function	<code>int tbkWtBit(uchar bitNum, uchar bitVal);</code>
C	<code>tbkWtBit(unsigned char bitNum, unsigned char bitVal);</code>
QuickBASIC	<code>BtbkWtBit% (ByVal bitNum%, ByVal bitVal%)</code>
Visual Basic	<code>VBtbkWtBit% (bitNum%, bitVal%)</code>
Turbo Pascal	<code>tbkWtBit(bitNum:byte; bitVal:byte):integer;</code>
Parameters	
uchar bitNum	The bit number to assert/unassert Valid values: 0 - 7
uchar bitVal	A flag that if non-zero will assert the specified bit, if 0 the bit is unasserted
Returns	TerrInvBitNum - Invalid bit number TerrNoError - No error
See Also	<code>tbkWtByte</code> , <code>tbkRdByte</code> , <code>tbkRdBit</code>
Program References	

tbkWtBit sets or clears a single digital output bit.

tbkWtByte

DLL Function	<code>int tbkWtByte(uchar byteVal);</code>
C	<code>tbkWtByte(unsigned char byteVal);</code>
QuickBASIC	<code>BtbkWtByte% (ByVal digOut%)</code>
Visual Basic	<code>VBtbkWtByte% (ByVal digOut%)</code>
Turbo Pascal	<code>tbkWtByte(byteVal:byte):integer;</code>
Parameters	
<code>uchar byteVal</code>	The value to write to the specified port Valid values: 0 - 255 for 8-bit ports 0-15 for 4-bit ports
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkRdByte</code> , <code>tbkWtBit</code> , <code>tbkRdBit</code>
Program References	

`tbkWtByte` writes a byte to the 8-bit digital output port.

tbkWtCntr0

DLL Function	<code>int tbkWtCntr0(uint cntr0);</code>
C	<code>tbkWtCntr0(unsigned cntr0);</code>
QuickBASIC	<code>BtbkWtCntr0% (ByVal cntr0%)</code>
Visual Basic	<code>VBtbkWtCntr0% (ByVal cntr0%)</code>
Turbo Pascal	<code>tbkWtCntr0(cntr0:word):integer;</code>
Parameters	
<code>uint cntr0</code>	The value to write to the count down register of Counter 0 Valid values: 0 - 65535
Returns	<code>TerrNoError</code> - No error
See Also	<code>tbkConfCntr0</code> , <code>tbkRdCntr0</code>
Program References	

`tbkWtCntr0` loads the count down register of Counter 0. See `tbkAdcConfCntr0` for various applications of counter 0.

tbkZeroConvert

DLL Function	<code>tbkZeroConvert(uint *counts, uint scans);</code>
C	<code>tbkZeroConvert(unsigned *counts, unsigned scans);</code>
QuickBASIC	<code>BtbkZeroConvert% (counts%, ByVal scans%)</code>
Visual Basic	<code>VBtbkZeroConvert% (counts%(), ByVal scans%)</code>
Turbo Pascal	<code>tbkZeroConvert(counts:WordP; scans:word):integer;</code>
Parameters	
<code>uint *counts</code>	The raw data from one or more scans.
<code>uint scans</code>	The number of scans of raw data in the counts array.
Returns	<code>TerrZCInvParam</code> - Invalid parameter value <code>TerrNoError</code> - No error
See Also	<code>tbkZeroSetup</code> , <code>tbkZeroSetupConvert</code> , <code>tbkAutoZero</code>
Program References	None

The `tbkZeroConvert` function compensates one or more scans according the previously called `tbkZeroSetup` function. This function will modify the array of data passed to it. See the Software Calibration and Zero Compensation chapter for a complete description of zero compensation.

tbkZeroSetup

DLL Function	tbkZeroSetup(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings);
C	tbkZeroSetup(unsigned nscan, unsigned zeroPos, unsigned readingsPos, unsigned nReadings);
QuickBASIC	BtbkZeroSetup% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%)
Visual Basic	VBtbkZeroSetup% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%)
Turbo Pascal	tbkZeroSetup(nscan:word; zeroPos:word; readingsPos:word; nReadings:word):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	TerrZCInvParam - Invalid parameter value TerrNoError - No error
See Also	tbkZeroConvert, tbkZeroSetupConvert, tbkAutoZero
Program References	None

The **tbkZeroSetup** 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. This function does not do the conversion. A non-zero return value indicates an invalid parameter error. See the Software Calibration and Zero Compensation chapter for a complete description of zero compensation.

tbkZeroSetupConvert

DLL Function	tbkZeroSetupConvert(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings, uint *counts, uint scans);
C	tbkZeroSetupConvert(unsigned nscan, unsigned zeroPos, unsigned readingsPos, unsigned nReadings, unsigned *counts, unsigned scans);
QuickBASIC	BtbkZeroSetupConvert% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%, counts%, ByVal scans%)
Visual Basic	VBtbkZeroSetupConvert% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%, counts%(), ByVal scans%)
Turbo Pascal	tbkZeroSetupConvert(nscan:word; zeroPos:word; readingsPos:word; nReadings:word; counts:WordP; scans:word):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	TerrZCInvParam - Invalid parameter value TerrNoError - No error
See Also	tbkZeroSetup, tbkZeroConvert, tbkAutoZero
Program References	None

For convenience, both the setup and convert steps can be performed with one call to **tbkZeroSetupConvert**. This is useful when the zero compensation needs to be performed multiple times because data was read from channels at different gains or polarities. See the Software Calibration and Zero Compensation chapter for a complete description of zero compensation.

API Reference Tables

These tables provide information for programming with the TempBook/66 Application Programming Interface. The tables are organized as follows:

API Parameter Reference Tables	
Table Title	Page
A/D Channel Descriptions	9-32
A/D Gain Definitions	9-32
A/D Trigger Source Definitions	9-32
Pretrigger Functions Trigger Source Definitions	9-33
Thermocouple Types	9-33
API Error Codes - C Languages	9-33
API Error Codes - QuickBASIC	9-34
API Error Codes - Turbo Pascal	9-35
API Error Codes - Visual Basic	9-36

A/D Channel Descriptions

A/D Channel	Source
0 to 15*	Analog input channels 0 to 15
16	CJC Channel
18	Shorted Channel
272	High-speed digital Inputs

*Note: In differential mode only channels 0 to 7 are valid.

A/D Gain Definitions

BASE UNIT	
Description	Value
TgainX1	0x00
TgainX2	0x10
TgainX5	0x20
TgainX10	0x30
TgainX20	0x11
TgainX50	0x21
TgainX100	0x31
TgainX200	0x32
TbkBiCJC	0x21
TbkBiTypeJ	0x31
TbkBiTypeK	0x31
TbkBiTypeT	0x32
TbkBiTypeE	0x21
TbkBiTypeN28	0x31
TbkBiTypeN14	0x31
TbkBiTypeS	0x32
TbkBiTypeR	0x32
TbkBiTypeB	0x32
TbkUniCJC	0x31
TbkUniTypeJ	0x32
TbkUniTypeK	0x32
TbkUniTypeT	0x32
TbkUniTypeE	0x31
TbkUniTypeN28	0x32
TbkUniTypeN14	0x32
TbkUniTypeS	0x32
TbkUniTypeR	0x32
TbkUniTypeB	0x32

A/D Trigger Source Definitions

Definition	Value	Trigger
TtsPacerClock	0x00	8254 Pacer Clock
TtsSoftware	0x10	Software
TtsTTLFall	0x20	External TTL falling edge
TtsTTL Rise	0x30	External TTL rising edge

Pretrigger Functions Trigger Source Definitions

Definition	Value	Trigger
TtsAnalogFallNeg	0x40	Falling below a negative setpoint
TtsAnalogRiseNeg	0x50	Rising above a negative setpoint
TtsAnalogRisePos	0x60	Rising above a positive setpoint
TDtsAnalogFallPos	0x70	Falling below positive setpoint

Thermocouple Types

Description	Value
TbkTypeJ	18
TbkTypeK	19
TbkTypeT	20
TbkTypeE	21
TbkTypeN28	22
TbkTypeN14	23
TbkTypeS	24
TbkTypeR	25
TbkTypeB	26

API Error Codes - C Languages

Error Name	Error Code	Description
TerrNoError	0x00	No error
TerrBadChannel	0x01	Specified LPT channel was out-of-range
TerrNotOnLine	0x02	Requested TempBook is not on-line
TerrNoTempBook	0x03	TempBook is not on the requested channel
TerrBadAddress	0x04	Bad function address
TerrFIFOFull	0x05	FIFO Full detected, possible data corruption
TerrInvChan	0x10	Invalid analog input channel
TerrInvCount	0x11	Invalid count parameter
TerrInvTrigSource	0x12	Invalid trigger source parameter
TerrInvGain	0x14	Invalid channel gain parameter
TerrInvPort	0x17	Invalid port parameter
TerrInvChip	0x18	Invalid chip parameter
TerrInvBitNum	0x1A	Invalid bit number parameter
TerrInvClock	0x1B	Invalid clock parameter
TerrInvTod	0x1C	Invalid time-of-day parameter
TerrInvGateCtrl	0x20	Invalid gate control parameter
TerrInvOutputCtrl	0x21	Invalid output control parameter
TerrInvInterval	0x22	Invalid interval parameter
TerrTypeConflict	0x23	An integer was passed to a function requiring a character
TerrMultBackXfer	0x24	A second background transfer was requested
TerrInvDiv	0x25	Invalid Fout divisor
TerrTCE_TYPE	0x26	TC type out-of-range
TerrTCE_TRANGE	0x27	Temperature out-of-CJC-range
TerrTCE_VRANGE	0x28	Voltage out-of-TC-range
TerrTCE_PARAM	0x29	Unspecified parameter value error
TerrTCE_NOSETUP	0x2A	tbkTCConvert called before tbkTCSetup
TerrOverrun	0x2C	A buffer overrun occurred
TerrZCInvParam	0x2D	Invalid zero compensation parameter
TerrZCNoSetup	0x2E	tbkZeroConvert called before tbkZeroSetup
TerrInvCalFile	0x2F	Cannot open the specified calibration file
TerrMemLock	0x30	Cannot lock allocated memory from Windows
TerrMemHandle	0x31	Cannot get a memory handle from Windows
TerrNoPreTActive	0x32	No pre-trigger configured

API Error Codes - QuickBASIC

Error Name	Error Code	Description
CONST TerrNoError%	&H00	No error
CONST TerrBadChannel%	&H01	Specified LPT channel was out-of-range
CONST TerrNotOnLine%	&H02	Requested TempBook is not on-line
CONST TerrNoTempBook%	&H03	TempBook is not on the requested channel
CONST TerrBadAddress%	&H04	Bad function address
CONST TerrFIFOFull%	&H05	FIFO Full detected, possible data corruption
CONST TerrInvChan%	&H10	Invalid
CONST TerrInvCount%	&H11	Invalid count parameter
CONST TerrInvTrigSource%	&H12	Invalid trigger source parameter
CONST TerrInvGain%	&H14	Invalid channel gain parameter
CONST TerrInvPort%	&H17	Invalid port parameter
CONST TerrInvChip%	&H18	Invalid chip parameter
CONST TerrInvBitNum%	&H1A	Invalid bit number parameter
CONST TerrInvClock%	&H1B	Invalid clock parameter
CONST TerrInvTod%	&H1C	Invalid time-of-day parameter
CONST TerrInvGateCtrl%	&H20	Invalid gate control parameter
CONST TerrInvOutputCtrl%	&H21	Invalid output control parameter
CONST TerrInvInterval%	&H22	Invalid interval parameter
CONST TerrTypeConflict%	&H23	An integer was passed to a function requiring a character
CONST TerrMultBackXfer%	&H24	A second background transfer
CONST TerrInvDiv%	&H25	Invalid Fout divisor
CONST TerrTCE.TYPE	&H26	TC type out of range
CONST TerrTCE.TRANGE	&H27	Temperature out-of-CJC-range
CONST TerrTCE.VRANGE	&H28	Voltage out-of-TC-range
CONST TerrTCE.PARAM	&H29	Unspecified parameter value error
CONST TerrTCE.NOSETUP	&H2A	tbkTCCovert called before tbkTCSetup
CONST TerrOverrun%	&H2C	A buffer overrun occurred
CONST TerrZCInvParam	&H2D	Invalid zero compensation parameter
CONST TerrZCNoSetup	&H2E	tbkZeroCovert called before tbkZeroSetup
CONST TerrInvCalFile	&H2F	Cannot open the specified calibration file
CONST TerrMemLock	&H30	Cannot lock allocated memory from Windows
CONST TerrMemHandle	&H31	Cannot get a memory handle from Windows
CONST TerrNoPreTActive	&H32	No pre-trigger configured

API Error Codes - Turbo Pascal

Error Name	Error Code	Description
TerrNoError	0	No error
TerrBadChannel	1	Specified LPT channel was out-of-range
TerrNotOnLine	2	Requested TempBook is not on-line
TerrNoTempBook	3	TempBook is not on the requested channel
TerrBadAddress	4	Bad function address
TerrFIFOFull	5	FIFO Full detected, possible data corruption
TerrInvChan	16	Invalid
TerrInvCount	17	Invalid count parameter
TerrInvTrigSource	18	Invalid trigger source parameter
TerrInvGain	20	Invalid channel gain parameter
TerrInvPort	23	Invalid port parameter
TerrInvChip	24	Invalid chip parameter
TerrInvBitNum	26	Invalid bit number parameter
TerrInvClock	27	Invalid clock parameter
TerrInvTod	28	Invalid time-of-day parameter
TerrInvGateCtrl	32	Invalid gate control parameter
TerrInvOutputCtrl	33	Invalid output control parameter
TerrInvInterval	34	Invalid interval parameter
TerrTypeConflict	35	An integer was passed to a function requiring a character
TerrMultBackXfer	36	A second background transfer was requested
TerrInvDiv	37	Invalid Fout divisor
TerrTCE_TYPE	38	TC type out-of-range
TerrTCE_TRANGE	39	Temperature out-of-CJC-range
TerrTCE_VRANGE	40	Voltage out-of-TC-range
TerrTCE_PARAM	41	Unspecified parameter value error
TerrTCE_NOSETUP	42	tbkTCConvert called before tbkTCSetup
TerrNot Capable	43	TempBook not capable of function
TerrOverrun	44	A buffer overrun occurred
TerrZCInvParam	45	Invalid zero compensation parameter
TerrZCNoSetup	46	tbkZeroConvert called before tbkZeroSetup
TerrInvCalFile	47	Cannot open the specified calibration file
TerrMemLock	48	Cannot lock allocated memory from Windows
TerrMemHandle	49	Cannot get a memory handle from Windows
TerrNoPreTActive	50	No pre-trigger configured

API Error Codes - Visual Basic

Error Name	Error Code	Description
Global Const TerrNoError%	&H00	No error
Global Const TerrBadChannel%	&H01	Specified LPT channel was out-of-range
Global Const TerrNotOnLine%	&H02	Requested TempBook is not on-line
Global Const TerrNoTempBook%	&H03	TempBook is not on the requested channel
Global Const TerrBadAddress%	&H04	Bad function address
Global Const TerrFIFOFull%	&H05	FIFO Full detected, possible data corruption
Global Const TerrInvChan%	&H10	Invalid
Global Const TerrInvCount%	&H11	Invalid count parameter
Global Const TerrInvTrigSource%	&H12	Invalid trigger source parameter
Global Const TerrInvGain%	&H14	Invalid channel gain parameter
Global Const TerrInvPort%	&H17	Invalid port parameter
Global Const TerrInvChip%	&H18	Invalid chip parameter
Global Const TerrInvBitNum%	&H1A	Invalid bit number parameter
Global Const TerrInvClock%	&H1B	Invalid clock parameter
Global Const TerrInvTod%	&H1C	Invalid time-of-day parameter
Global Const TerrInvGateCtrl%	&H20	Invalid gate control parameter
Global Const TerrInvOutputCtrl%	&H21	Invalid output control parameter
Global Const TerrInvInterval%	&H22	Invalid interval parameter
Global Const TerrTypeConflict%	&H23	An integer was passed to a function requiring a character
Global Const TerrMultBackXfer%	&H24	A second background transfer
Global Const TerrInvDiv%	&H25	Invalid Fout divisor
Global Const TerrTCE_TYPE%	&H26	TC type out-of-range
Global Const TerrTCE_TRANGE%	&H27	Temperature out-of-CJC-range
Global Const TerrTCE_VRANGE%	&H28	Voltage out-of-TC-range
Global Const TerrTCE_PARAM%	&H29	Unspecified parameter value error
Global Const TerrTCE_NOSETUP%	&H2A	tbkTCConvert called before tbkTCSetup
Global Const TerrOverrun%	&H2C	A buffer overrun occurred
Global Const TerrZCInvParam	&H2D	Invalid zero compensation parameter
Global Const TerrZCNoSetup	&H2E	tbkZeroConvert called before tbkZeroSetup
Global Const TerrInvCalFile	&H2F	Cannot open the specified calibration file
Global Const TerrMemLock	&H30	Cannot lock allocated memory from Windows
Global Const TerrMemHandle	&H31	Cannot get a memory handle from Windows
Global Const TerrNoPreTActive	&H32	No pre-trigger configured

Overview

The enhanced Application Programming Interface (API) allows you to create custom software to satisfy your TempBook data acquisition requirements. Two chapters explain the enhanced API: this chapter gives you the basic concepts needed to write effective programs, and chapter 11 describes the API functions in detail. This chapter explains *how to combine the API functions into useful routines* and is divided into 3 parts:

- **Data Acquisition Environment** outlines related concepts and defines system 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 example program excerpt.
- **Summary Guide of Selected API Functions** is an easy-to-read table that describes when to use the basic API functions.

Note: The TempBook enhanced API is a subset of the **DaqX** API which provides a common interface for 32-bit data acquisition applications (TempBook, WaveBook, DaqBook, DaqBoard, Daq PC-Card, etc). This manual describes the commands that pertain to the TempBook.

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 11 (*daqCommand 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 (*Programmer's Guide*). Language support varies as follows:

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

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; chapters 6 to 8 demonstrate examples using the standard API).

Hardware Capabilities and Constraints

To program the system effectively, you must understand your 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. 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

Several data acquisition concepts are listed here; you must apply these concepts as needed in your situation. Some of these concepts include:

- **Device and parameter identification.** Refer to the related reference tables in chapter 11.
- **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.
- **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 the trigger functions in chapter 11.
- **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 2048 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. Channels sampled in a scan 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	10-3
Acquisition	Foreground Acquisition with One-Step Commands	10-5
	Temperature Acquisition Using One-Step Commands	10-7
	Counted Acquisition Using Linear Buffers	10-9
	Indefinite Acquisition, Direct-To-Disk Using Circular Buffers	10-11
	Multiple Hardware Scans, Software Triggering	10-14
	Background Acquisition	10-16
Data Handling	Temperature Acquisition Using TC Conversion Functions	10-18
	Double Buffering	10-21
	Direct-to-Disk Transfers	10-23
	Transfers With Driver-Allocated Buffers	10-26

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&("TempBook0")
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&("TempBook0") < 0) Then
    "Cannot open "TempBook0"
```

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.)

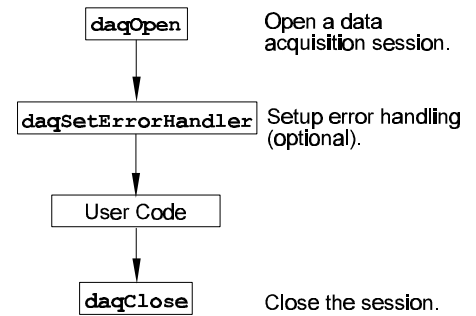
```
handle& = VBdaqOpen&("TempBook0")
ret& = VBdaqSetErrorHandler&(handle&, 100)
```

```
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). First, some constants need to be defined and variables dimensioned.

```

Const freq! = 1000!
Const gain& = DgainX1&
Const flags& = DafAnalog&+DafUnipolar&
Const scans& = 9
Const channels& = 8
Const rising& = DatdRisingEdge
Const HYSTERESIS& = 0.1
Dim buf%(scans& * channels&)
Dim handle&
Dim i&, j&
Dim sample%
Dim ret&
    
```

```

`1000Hz sample rate
`gain of x1
`unipolar mode on
`number of scans to acquire
`number of channels to scan
`XXXX I have no idea
`with a hysteresis of .1
`array buffer to hold data
`handle for TempBook device
`counter variables
`hold a single reading
`function return value
    
```

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 bipolar 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 $\times 1$. The `gain&` constant is defined as `DgainX1&`, defined constant from `DaqX.bas` and included at the beginning of this program. Likewise, the `flags&` constant parameter is defined to be the sum of the `DafAnalog` and `DafUnipolar` flags, which are also defined in `DaqX.bas`.

```

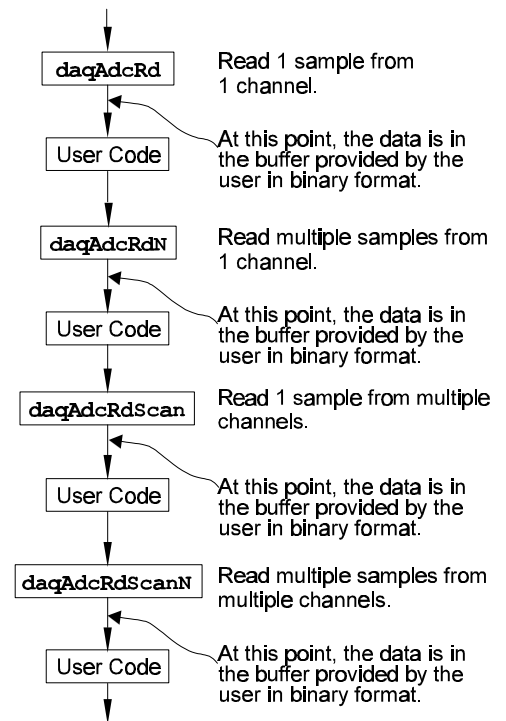
ret& = VBdaqAdcRd(handle& 1, sample%, gain&, flags&)
Print Format$" &####"; "Result of AdcRd:"; sample%(0)
    
```

The next line requests 10 readings from channel 1 at a gain of $\times 1$, using immediate triggering at 1 kHz.

```

ret& = VBdaqAdcRdN(handle&,1, buf%(), scans&, DatsImmediate&, rising&, 0!,
    freq!, gain&, flags&)
Print "Results of AdcRdN: ";
For x& = 0 To 9
    Print Format$ "#### "; buf%(x&);
Next x&
    
```

The program will then collect one sample of channels 1 through 8 using the `VBdaqAdcRdScan` function.



```

ret& = VBdaqAdcRdScan(handle&,1, channels&, buf%(), DgainX1&,
  DafAnalog&+DafUnipolar&)

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

```

Finally, the program will collect 9 scans from channels 1 through 7 with an immediate trigger, then display the results.

```

ret& = VBdaqAdcRdScanN& (handle&, 1, channels&, buf%(), scans&,
  DatsImmediate&, rising&, 0!, freq!, gain&, flags&)

For i& = 0 To channels&-1
  Print Format$" & # & ####"; "Channel:"; i&+1; "Data:";
  For j& = 0 To scans&-1
    print Tab(j&*7+17); InttoUint(buf%(j&*channels&+i&));
  next j
  print
next i&

```

Now to close the device when it's no longer needed:

```

ret& = VBdaqAdcClose(handle&)

```

Temperature Acquisition Using One-Step Commands

This example demonstrates the Enhanced API's high-level one-step routines to read thermocouple input from the TempBook/66 device. These functions combine scan sequencer setup, ADC data collection, and thermocouple linearization. Functions used include:

- VBdaqSetDefaultHandler&(handler&)
- VBdaqGetDeviceCount&(deviceCount&)
- VBdaqGetDeviceList&(deviceList&, deviceProps)
- VBdaqOpen&(daqName\$)
- VBdaqAdcSetDataFormat&(handle&, rawFormat&, postProcFormat&)
- VBdaqAdcRd&(handle&, chan&, sample%, gain&, flags&)
- VBdaqAdcRdScan&(handle&, startChan&, endChan&, buf%, gain&, flags&)
- VBdaqAdcRdScanN&(handle&, chan&, buf%, scanCount&, triggerSource&, rising&, level%, freq!, gain&, flags&)
- VBdaqAdcRdN&(handle&, chan&, startChan&, endChan&, buf%, scanCount&, triggerSource&, rising&, level%, freq!, gain&, flags&)
- VBdaqClose&(handle&, daqEvent&)

```
Const Scans& = 5000
Const Level% = 0
Const Rising& = 0
Const Start& = 0
Const End& = 0
Const Freq! = 6000.0
Const Gain& = TbkBiTypeJ&
Const Flags& = DafBipolar& + DafDifferential& + DafTcTypeJ&
Const Chans& = End& - Start&+1
```

```
Dim buf%(80), handle&, ret&, flags&
Dim I&, j&, deviceCount&, handle&, deviceIndex&
Dim temp%, temps%(Scans&*Chans&)
Dim sum!, totals!(Chans&)
Dim ret&
```

First, we need to open the TempBook/66 device. This example uses a device named "TempBook0". The device name must be a valid device name for a configured device.

```
handle& = VBdaqOpen&("TempBook0")
```

Set the raw data format to native and the post-processing data format to temperature in tenths of a degree C. The post-processing data format controls the format of the data returned by the one-step acquisition functions such as **daqAdcRd**. In this case, **daqAdcRd** returns temperatures rather than raw ADC readings.

```
ret& = VBdaqAdcSetDataFormat& (handle&, DardfNative&, DappdfTenthsDegC&)
```

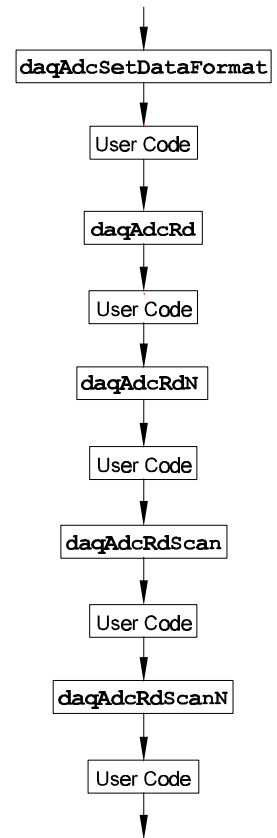
The following statement retrieves a single ADC sample from a type J thermocouple on channel **start&** and converts the reading to a temperature. The temperature is returned in the **sample%** parameter and is a 16-bit quantity which represents tenths of degrees C.

```
ret& = VBdaqAdcRd& (handle&, start&, temp%, TbkBiTypeJ, Flags&)
```

Now the results for the single-sample read are displayed:

```
Print "Results of daqAdcRd for channel ", start&, ": (single reading)"
Print "Temperature: ", temp/10.0, "C"
```

The next statement retrieves **scans&(5000)** number of samples from a type J thermocouple on channel **start&** and converts the readings to a single temperature using block averaging. The trigger source is set to **DatsImmediate&** so the scans will be acquired immediately. Again, the temperatures are returned as 16-bit words representing tenths of degrees C.



```
ret& = VBdaqAdcRdN(handle&, Start&, temps%(), Scans&, DatsImmediate&,
  Rising&, Level&, Freq!, TbkBiTypeJ&, Flags&)
```

The averaging of the temperature values into a single temperature value is performed through the following:

```
sum! = 0.0
For I& =0 To Scans&
    sum! = sum! + temps%(I&)
Next I&
sum! = sum!/Scans&
```

The averaged temperature can now be printed out.

```
Print "Results of daqAdcRdN: (\,Scans&," readings averaged)"
Print "Channel", Start&, " Temperature: ", sum!/10.0, "C"
```

Next, a single scan will be retrieved for a multiple channel scan configuration. The following statement configures the scan from the **Start& (0)** to the **End& (0)** and configures each channel as a type J thermocouple. The returned values for each channel will be placed in the **temps%** array and will be 16-bit words representing tenths of degrees C.

```
ret& = VBdaqAdcRdScan& (handle&, Start&, End&, temps%(), TbkBiTypeJ&,
  Flags&)
Print "Result of daqAdcRdScan (Single Readings) : "
For I& = Start& To End&
    Print "Channel ", I&," Temperature ", temps%(I&)/10.0, " C "
Next I&
```

Finally, we will retrieve multiple scans for a multiple-channel scan configuration. The scan will be defined using type J thermocouples for each channel in the scan group configuration. The scan group will start at **Start& (0)** and go to **End& (0)**; and all channels will be type J thermocouples. The returned values will be placed in the **temps%** array and will be 16-bit words representing tenths of degrees C.

```
ret& = VBdaqAdcRdScanN& (handle& , Start&, End& , temps%, Scans&,
  DatsImmediate&, Rising&, Level&, Freq!, Gain&, Flags&)
```

The following code will average all samples collected for each channel and display the results.

```
` Zero the totals
For I& = Start& To End&+1
    totals!(I&) = 0.0
Next I&

` Average the temperatures
For I& = 0 To Scans&
    For j& = Start& To End&
        totals!(j&) = totals!(j&) + temps%(I&*Chans& + j&)
    Next j&
Next I&

` Divide the totals
For I& = 0 To Chans&
    totals!(I&) = totals!(I&)/Scans&
Next I&

` Print "Results of daqAdcRdScanN: "
For I& = 0 To Scans&
    Print "Channel", I&, " Temperature: ", totals!(I&)/10.0, "C"
Next I&

` Close the TempBook/66
ret& = VBdaqClose&(handle&)
```


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.

```
Const freq!=1000!
Const scans#=10
Dim buf%(BLOCK&*channels&), handle&, ret&, flags&
```

where

```
const block& = 6 and
const channels& = 8
```

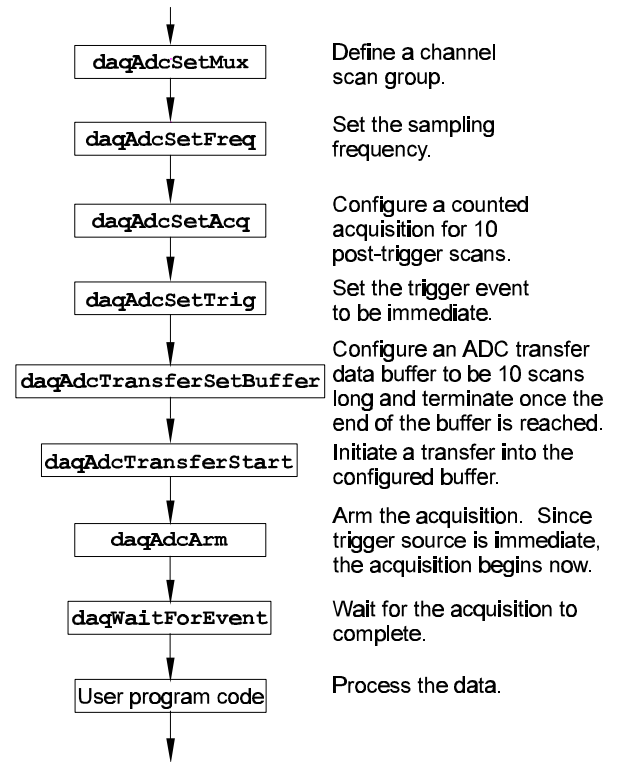
The acquisition mode must be configured as a fixed-length acquisition with no pre-trigger scan data and 10 scans of post-trigger scan data. The mode is set to **DaamNShot&** to configure a fixed-length acquisition that will terminate automatically upon the satisfaction of the post-trigger count of 10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

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

The following function defines the channel scan group. The function specifies a channel scan group from channel 1 through 8 with all channels being analog unipolar input channels with a gain of $\times 1$. 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&,1, channels&, DgainX1&,
DafAnalog&+DafUnipolar&)
```

Next, set the internal sample rate to 1 kHz.



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

The sample rate will not be *exactly* 1 kHz; the actual frequency can be checked if necessary by:

```
ret& = VBdaqAdcGetFreq(handle&, freq!)
```

The “actual” frequency set will be stored in `freq` after the function call returns.

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 software trigger source which will start the acquisition upon a call to `VBdaqAdcSoftTrig()`. The variable `DatsSoftware&` is a constant defined in `DaqX.bas`. Since the trigger source is configured as software, 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. 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 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 software trigger, the acquisition will begin immediately upon execution of the `daqAdcSoftTrig()` function.

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

After setting up and arming the acquisition, the data is ready to be collected. The following line initiates an A/D transfer from the `TempBook/Daq*` device to the defined user buffer which will begin after the trigger event is satisfied (upon the completion of the `daqAdcSoftTrig()` function call).

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

Now the trigger will start the transfer:

```
ret& = VBdaqAdcSoftTrig(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 j& = 0 To channels& - 1  
        Print Format$(IntToUInt&(buf%(j&)), "00000"); " ";  
    Next j&  
    Print  
Next i&  
Print "R"
```

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.

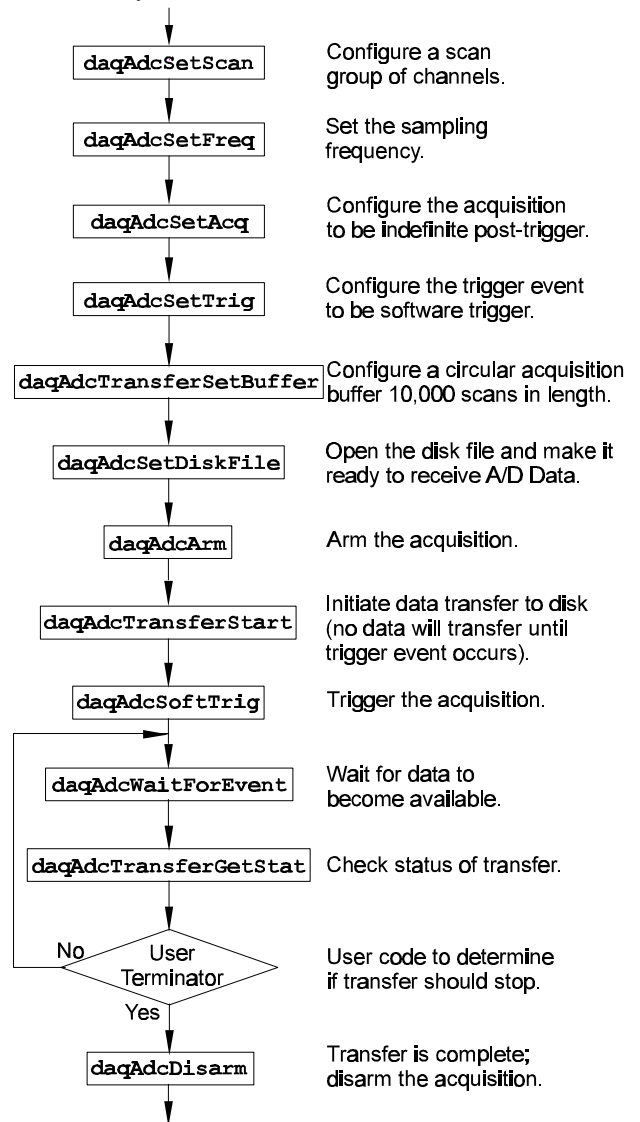
At this point, the Daq* device trigger is armed and A/D acquisition to disk will begin immediately upon trigger detection.

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.

Multiple Hardware Scans, Software Triggering

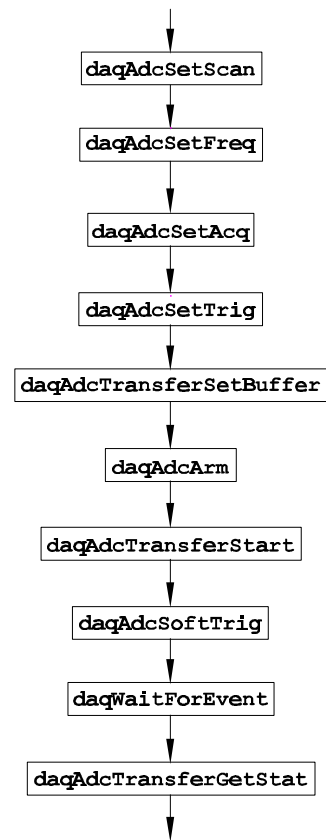
This model takes multiple scans from several channels. The functions used here are of a lower level than the one-step functions, and more control is allowed over the acquisition. This program exemplifies this flexibility by individually configuring the channels and by explicitly setting up the transfer buffer.

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 is set up and the acquisition is then armed by calling the `daqAdcArm` function.

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

This example will retrieve 10 scans for channels 0, 5, and 8, 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&)`



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

```

Const freq! = 3000
Const scans& = 10
Const channels& = 3
Dim buf%(scans& * channels&)
Dim chans&(channels&), gains&(channels&), flags&(channels&)
  
```

Now set up the desired channels and their individual gains and flags.

```

chans&(0) = 0      ' high speed digital channel
chans&(1) = 5      ' analog channel 5
chans&(2) = 8      ' analog channel 8
  ' Channel gains and flags setting
For i& = 0 To channels& - 1
    gains&(i&) = DgainX1& ' unity gain
    flags&(i&) = DafAnalog& + DafSingleEnded& + DafUnipolar&
Next i&
  
```

Open the device, and set up the error handler. For simplicity, the error handler is not defined explicitly. Refer to Example 1 for more information.

```

handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC3
  
```

Now set the scan configuration:

```
ret& = VBdaqAdcSetScan&(handle&,chans&(), gains&(), flags&(), channels&)
```

Next, set the internal sample rate to 3 kHz.

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

The acquisition mode needs to be configured to be a 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 having finite length and, as such, will be terminated when the post-trigger count has been satisfied. Once finished, the acquisition is automatically disarmed.

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

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 software, 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. Since a circular buffer will not be used, the buffer cycle mode should be turned off with **DatmCycleOff&**. The single update mode is specified with **DatmUpdateSingle&**. The buffer size is set to 10, the number of scans.

```
ret& = VBdaqAdcTransferSetBuffer&(handle&,buf%(), scans&,
DatmUpdateSingle&+DatmCycleOff&)
```

With all acquisition parameters and the transfer 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. 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.

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

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

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

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

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

Finally, display the results and close the device.

```
Print "Results of BufferTransfer:"
Print "          Digital_ch_0  Analog_ch_5  Analog_ch_8"
For i& = 0 To scans& - 1
  ' shift the upper (valid) 8 bits of the digital input to the lower 8
  bits
  buf%(i& * channels&) = ((buf%(i& * channels&) And &HFF00) \ 256) And
  &HFF
  Print "Scan"; i& + 1; "Data:";
  For j& = 0 To channels& - 1
    Print Tab(j& * 14 + 17); buf%(i& * channels& + j&);
  Next j&
  Print
Next i&
ret& = VBdaqClose(handle&)
```

Background Acquisition

This example reads scans from several channels into a user-allocated buffer in the background. Functions used include:

- VBdaqAdcArm&(handle&)
- VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)
- VBdaqAdcSetFreq&(handle&, freq#)
- VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, 1)
- VBdaqAdcSetTrig&(handle&, DatsSoftware&, 0,0,0,0)
- VBdaqAdcSoftTrig&(handle&)
- VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
- VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& + DatmUpdateSingle&)
- VBdaqAdcTransferStart(handle&)
- VBdaqClose(handle&)
- VBdaqOpen("TempBook0")
- VBdaqSetErrorHandler(handle&, 100)

The constants used are defined as follows:

```
Const channels& = 8
Const scans& = 9
Const freq# = 200
```

As usual, the device is opened and the error handler set up:

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC4
```

The acquisition is configured for 9 post-trigger scans and Nshot mode:

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

Set up the scan configuration for channels 1 to 9 with a gain of ×1:

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, 1)
```

Set the post-trigger scan rates:

```
ret& = VBdaqAdcSetFreq&(handle&, freq#)
```

Set the trigger source to a software trigger command; the other trigger parameters are not needed with a software trigger.

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

Arm the acquisition:

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

Now to set up the buffer for a background acquisition in update-single mode with cycle-mode off:

```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& + DatmUpdateSingle&)
```

Start the transfer, and trigger to begin transferring data:

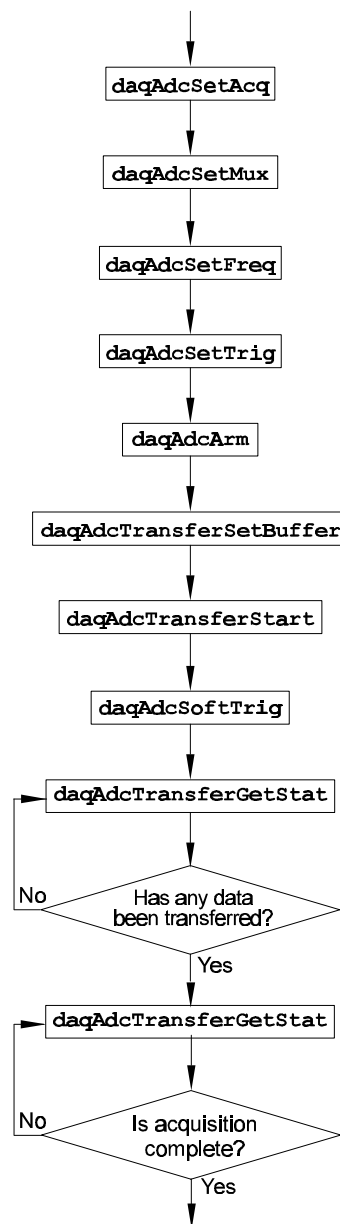
```
ret& = VBdaqAdcTransferStart(handle&)
ret& = VBdaqAdcSoftTrig&(handle&)
```

These next few lines wait for the first data to be received, by checking the **retCount** value after calling **daqAdcTransferGetStat()**:

```
retCount& = 0
While retCount& = 0
    ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
```

With the same function, wait for the acquisition to complete:

```
While active& <> 0
    ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
Print "Acquisition complete: "; retCount&; "scans acquired."
```



Now the data can be displayed or manipulated:

```
Print "Data acquired:"
  For i& = 0 To channels& - 1
    Print "Channel"; i& + 1; "Data:";
    For j& = 0 To scans& - 1
      Print Tab(j& * 7 + 17); buf%(j& * channels& + i&);
    Next j&
    Print
  Next i&
```

Finally, close the device:

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

Temperature Acquisition Using TC Conversion Functions

This example demonstrates the general-purpose data-transfer functions coupled with TC-specific conversion routines. This method first configures the channel scan group for thermocouple input, then acquires the raw A/D data from the thermocouple input, and finally, converts the raw A/D data to temperature units in degrees C.

- VBdaqOpen&(daqName\$)
- VBdaqAdcSetFreq& (handle&, freq!)
- VBdaqAdcSetAcq& (handle&, mode&, preTrigCount&, postTrigCount&)
- VBdaqAdcSetScan& (handle&, channels&(), gains&(), flags&(), chanCount&)
- VBdaqAdcSetClockSource&(handle&, clockSource&)
- VBdaqAdcSetTrig& (handle&, triggerSource&, rising&, level%, hysteresis%, channel&)
- VBdaqAdcTransferSetBuffer&(handle&, buf%(), scanCount&, transferMask&)
- VBdaqAdcTransferStart& (handle&)
- VBdaqAdcArm&(handle&)
- VBdaqWaitForEvent(handle&, event&)
- VBdaqAdcTransferGetStat& (handle&, active&, retCount&)
- VBdaqCvtTCSetupConvert(nscan&, cjcPosition&, ntc&, tcType&, bipolar&, avg&, counts%(), scans&, temp%(), ntemp&)
- VBdaqClose&(handle&)

The following list defines the necessary constants and variables for temperature acquisition and conversion.

```

Const Scans& = 10
Const Level% = 0
Const Rising& = 0
Const Start& = 0
Const End& = 7
Const NumTcChans& = End& - Start& + 1
Const TotalChans& = NumTcChans + 3
Const TcGain& = TbkBiTypeJ&
Const CjcGain& = TbkBiCJC&
Const TcType& = TbkTCTypeJ&
Const Freq! = 1000.0
Const AvgType& = 0
Const Gain& = TbkBiTypeJ&
Const Flag& = DafUnsigned& + DafDifferential& + DafBiPolar&
Const Chans& = End& - Start&+1

Dim      buf%(Scans& * TotalScans&)
Dim      temp%(NumTcChans&)
Dim handle&
Dim      I&,j&, active&, retCount&
Dim Gains&(TotalChans&)
Dim Chans&(TotalChans&)
Dim Flags&(TotalChans&)

```

First, we need to open the TempBook/66 device. This example uses a device named "TempBook0". The device name must be for a valid configured device.

```

` Open TempBook/66 device
handle& = VBdaqOpen&("TempBook0")

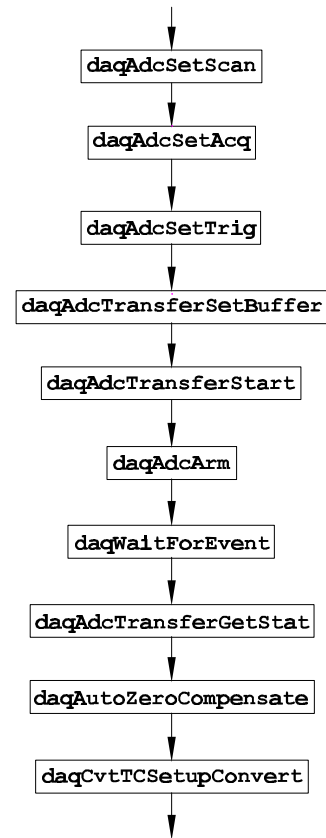
```

Next, build the channel scan group configuration. The following code sets up the configuration arrays that define the channel scan group. The channels are configured with the appropriate gain, mode and polarity definitions.

```

` Configure CJC and shorted channel

```



```

Chans[0] = 18    \ Shorted channel
Chans[1] = 18    \ Shorted channel
Chans[2] = 16    \ CJC channel

Gains[0] = TbkBiCJC&          \ Shorted channel at CJC gain
Gains[1] = TC_GAIN;          \ Shorted channel at thermocouple gain
Gains[2] = TbkBiCJC;          \ CJC channel reading at CJC gain

\ All channels bipolar
Flags&[0] = Flag&
Flags&[1] = Flag&
Flags&[2] = Flag&

\ Configure thermocouple channels after shorted and CJC channels in scan
For I& =0 To NumTcChans
  Chans&(I&+3) = Start& + I&;
  Gains&(I&+3) = TcGain&
  Flags&(i+3) = Flag&
Next I&

\ Configure the TempBook with the scan sequence
ret& = VBdaqAdcSetScan& (handle&, chans&(), gains&(), flags&(), TotalChans&)

```

Now the acquisition itself needs to be configured. This acquisition specifies a counted acquisition (**DaamNShot&**) with no pre-trigger and 10 post-trigger (**Scans&**) scans.

```

\ Set the acquisition mode and clock source
ret& = VBdaqAdcSetAcq& (handle&, DaamNShot&, 0, Scans&)
ret& = VBdaqAdcSetClockSource& (handle&, DacsAdcClock&)

```

The acquisition trigger is set to immediate triggering (**DatsImmediate&**); however, the acquisition will not be started until it is armed later on.

```

\ Configure to trigger immediately after armed
ret& = VBdaqAdcSetTrig& (handle&, DatsImmediate&, True, 0, 0, 0)

```

The scan frequency is now set to 1000 Hz (**Freq!**).

```

\ Set the scan frequency
ret& = VBdaqAdcSetFreq& (handle&, Freq!)

```

Now that the acquisition has been configured, the transfer buffer for the raw data must be defined. The following routines will configure the **buf%** array as the raw data array with a length of 10 (**Scans&**) scans. Also, the buffer is set to linear mode by specifying the **DatmCycleOff&** flag. The **DatmUpdateSingle&** flag indicates that the buffer should be updated while each sample is acquired. After the buffer has been defined, the raw data transfer will be started with the **daqAdcTransferStart** routine. However, data transfer into the raw data buffer will not really begin until the acquisition has been triggered.

```

\ Set up and start the transfer
ret& = VBdaqAdcTransferSetBuffer& (handle&, buf%(), Scans&, DatmCycleOff& +
DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart& (handle&)

```

The acquisition is now configured, and the raw data buffer is ready to receive transferred data. To initiate the transfer, the acquisition needs to be armed. Once armed, the data transfer will begin immediately since the trigger source was configured as **DatsImmediate&**.

```

\ Arm the acquisition
VBdaqAdcArm& (handle&)

```

The acquisition and transfer of the raw data is now active. The following statement can be used to wait for the termination of the transfer. Once the transfer is terminated, the raw data will then be present in the buffer.

```

\ Wait until the acquisition is complete
ret& = VBdaqWaitForEvent& (handle&, DteAdcDone&)
\ Get the number of scans acquired

```

```

ret& = VBdaqAdcTransferGetStat& (handle&, active&, retCount&)
Print retCount&, " Scans acquired. Converting data... "

```

The raw data is now available to be converted to temperature readings. The following code performs the conversion process on the `buf%()` raw data buffer. When complete, the temperature values will be available in the `temps%()` array. The `VBdaqCvtTCSetupConvert` function is used to perform the conversion. Here, the conversion is configured with the number of readings per scan (`TotalChans&=11`), CJC position of 2, the number of thermocouple channels (`NumTcChans&=8`), the thermocouple type (`TcType&= TbkTcTypeJ&`), bipolar raw data, no averaging (`AvgType&=0`), the raw data buffer (`buf%()`), the total number of scans (`Scans&=10`) and the target array for the converted temperature data in tenths of degrees C (`temp%()`).

```

` Configure the TC conversion functions to use zero correction
ret& = VBdaqAutoZeroCompensate& (1)

` Configure and Perform Thermocouple Linearization
ret& = VBdaqCvtTCSetupConvert& (TotalChans&, 2, NumTcChans&, TcType&,1,
    AvgType&,
    buf%(), Scans&, temps%(), NumTcChans&)

` Print a channel column labels
Print "Averaged temperature readings:"

For I& = 0 To NumTcChans&
    Print "Channel", I&, " Temperatue ", temp%(I&)/10.0," C"
Next I&

` Close and exit
ret& = VBdaqClose& (handle&)

```

Double Buffering

This example demonstrates using double buffering in the background mode, so that data can be read into one buffer while the another buffer can be processed in the foreground. Functions used include:

- `VBdaqAdcArm&(handle&)`
- `VBdaqAdcBufferTransfer(buf1%(0), BLOCK&, 0, 0, 0, tmpActive&, tmpRetCount&)`
- `VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)`
- `VBdaqAdcSetFreq&(handle&, freq!)`
- `VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, DafAnalog&+DafUnipolar&)`
- `VBdaqAdcSetTrig(handle&, DatsSoftware&, rising&, level%, HYSTERESIS%, 1)`
- `VBdaqAdcSoftTrig&(handle&)`
- `VBdaqAdcTransferGetStat(handle&, active&, retCount&)`
- `VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&, DatmCycleOff& + DatmUpdateSingle&)`
- `VBdaqAdcTransferStart(handle&)`
- `VBdaqClose(handle&)`
- `VBdaqOpen("TempBook0")`
- `VBdaqSetErrorHandler(handle&, 100)`

The following constants define the number of channels and other acquisition parameters:

```
Const channels& = 8
Const scans& = 20000
Const BLOCK& = 1000
Const freq! = 5000#
Const level% = 0
Const HYSTERESIS% = 0
Const rising& = 0
```

Dimension 2 buffers for double buffering:

```
Dim buf0%(channels& * BLOCK&)
Dim buf1%(channels& * BLOCK&)
```

Set error handler and initialize TempBook:

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC7
```

Set the acquisition to `NShot` on trigger and the post-trigger scan count:

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

Set the scan configuration for unity gain, from channels 1 to 8, in analog unipolar mode:

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, DafAnalog&+DafUnipolar&)
```

Set the post-trigger scan rate:

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

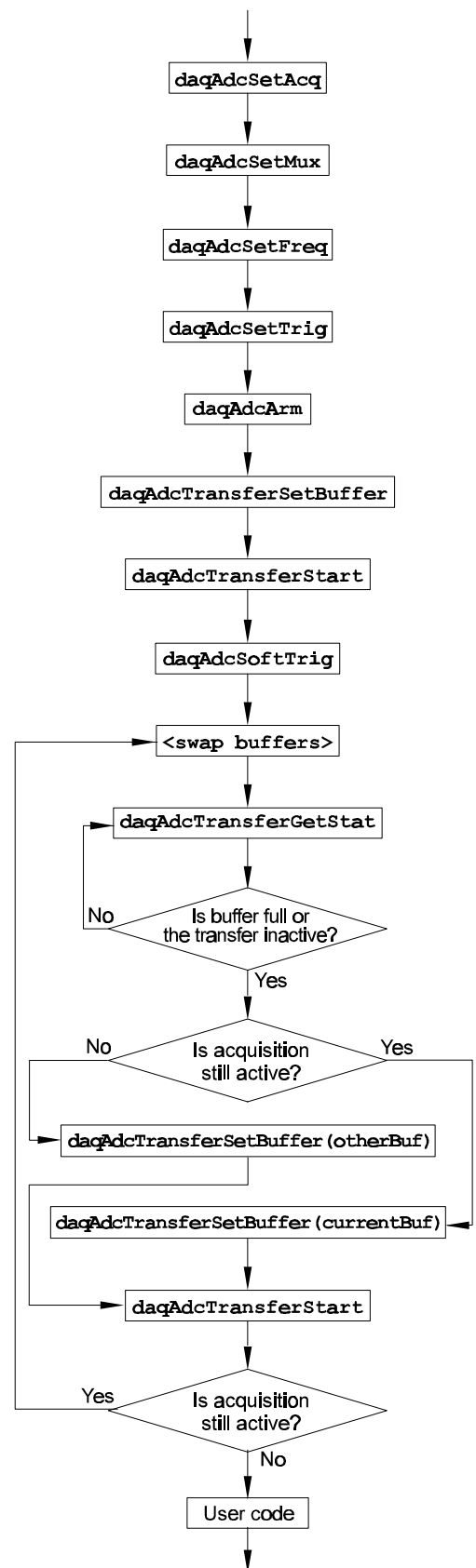
Set the trigger source to a software trigger command:

```
ret& = VBdaqAdcSetTrig(handle&, DatsSoftware&, rising&, level%, HYSTERESIS%, 1)
```

Arm the acquisition:

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

Set up the first buffer for `BLOCK` scans, with cycle mode off and update single on:



```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&, DatmCycleOff& +
DatmUpdateSingle&)
```

Start the first transfer; the transfer will actually start upon trigger detection. In this case, the following software trigger will start the transfer:

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

Issue a software trigger command to the hardware to trigger the transfer:

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

The next **do** loop swaps the active buffer back and forth from **buf0** to **buf1** and waits for the acquisition to go inactive or the buffer to fill up. Swapping continues until the transfer goes inactive:

```
whichBuf& = 0
Do
```

The following line changes the current buffer:

```
If whichBuf& = 1 Then whichBuf& = 0 Else whichBuf& = 1
```

Wait for the acquisition to go inactive or the buffer to be filled:

```
Do
ret& = VBdaqAdcTransferGetStat(handle&, active&, retCount&)
Loop While ((active& <> 0) And (retCount& < BLOCK&))
```

If the previous acquisition is still active, start another transfer into the next buffer:

```
If (active& <> 0) Then
If whichBuf& = 0 Then
ret& = VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&,
DatmCycleOff& + DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart(handle&)
```

Otherwise, restart the transfer into the current buffer:

```
Else
'ret& = VBdaqAdcBufferTransfer(buf1%(0), BLOCK&, 0, 0, 0,
tmpActive&, tmpRetCount&)
ret& = VBdaqAdcTransferSetBuffer(handle&, buf1%(), BLOCK&,
DatmCycleOff& + DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart(handle&)
End If
End If
```

Send the data into the process buffer, **totals()**:

```
If (retCount& > 0) Then
```

Average the readings in the process buffer and print the results:

```
For j& = 0 To channels& - 1
totals&(j&) = 0
Next j&
For i& = 0 To retCount& - 1
For j& = 0 To channels& - 1
```

Decide which buffer to add the data from:

```
If whichBuf& = 0 Then
totals&(j&) = totals&(j&) + buf1%(i& * channels& + j&)
Else
totals&(j&) = totals&(j&) + buf0%(i& * channels& + j&)
End If
Next j&
Next i&
```

Display the averaged results:

```
Print "Averages:";
For j& = 0 To channels& - 1
Print Tab(j& * 7 + 17); Format$((5# / 32768#) * totals&(j&) /
retCount&, "#0.000");
Next j&
Print
End If
```

Continue the **do..while** loop until the acquisition goes inactive:

```
Loop While (active& <> 0)
```

Close the device before exiting:

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

Direct-to-Disk Transfers

This example takes multiple scans from multiple channels and writes them directly to disk in a packed-data format. Functions used are:

- VBdaqAdcArm&(handle&)
- VBdaqAdcSetAcq&(handle&, DaamNShot, 0, scans&)
- VBdaqAdcSetDiskFile&(handle&, "adcex8.bin", DaomAppendFile&, 0)
- VBdaqAdcSetFreq&(handle&, freq!)
- VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, DafUniPolar&+DafAnalog&)
- VBdaqAdcSetTrig&(handle&, DatsSoftware&, DatdRisingEdge&, 0, HYSTERESIS%, 1)
- VBdaqAdcSoftTrig&(handle&)
- VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
- VBdaqAdcTransferSetBuffer&(handle&, buf%(), BLOCK&, DatmCycleOn& + DatmUpdateBlock&)
- VBdaqAdcTransferStart&(handle&)
- VBdaqClose&(handle&)
- VBdaqCvtRawDataFormat&(buf%(), DacuUnpack, BLOCK&, channels&, scanCount&)
- VBdaqOpen&("TempBook0")
- VBdaqSetErrorHandler(handle&, 100)

File handling in MS-Windows requires calls to the windows API, so the following constants are defined for use in those calls. For further information, see `mapiwin.h`.

```
Const GENERIC_READ& = &H80000000
Const OPEN_EXISTING = 3
Const FILE_ATTRIBUTE_NORMAL& = &H80
Const OPEN_ALWAYS = 4
Const CREATE_ALWAYS = 2
```

Also define the usual constants defining scan parameters and some declarations for file manipulation:

```
Const channels& = 2
Const scans& = 800
Const freq! = 200#
Const BLOCK& = 200 ' CHANNELS& * BLOCK& must be
a multiple of 4
Const HYSTERESIS% = 0
Dim buf%(channels& * BLOCK&)
Dim fileHandle&
Dim byteCount&, wordCount&, sampleCount&,
scanCount&
Dim binFile$
```

First set the name of the file to be used for the acquisition:

```
binFile = "adcex8.bin"
```

Open the device, and set the error handler:

```
handle& = VBdaqOpen&("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC8
```

Set the acquisition to NShot on trigger and the post-trigger scan count:

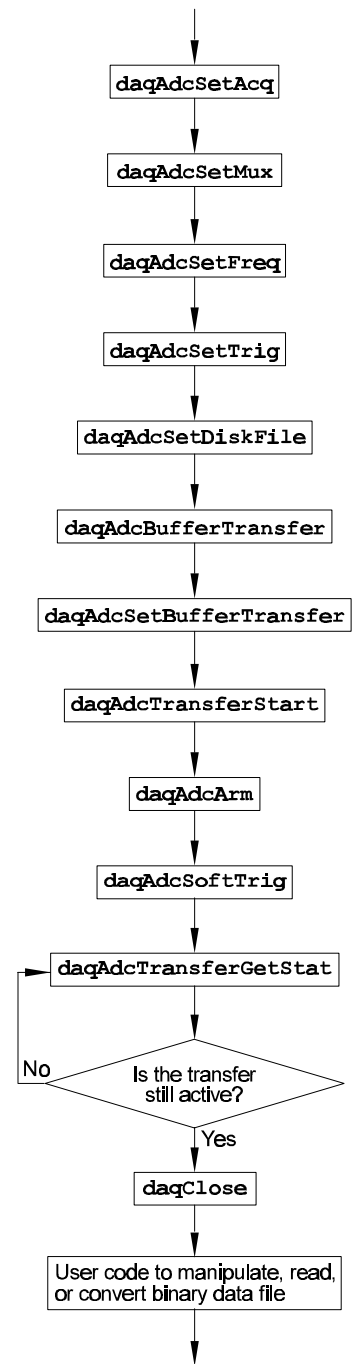
```
ret& = VBdaqAdcSetAcq&(handle&, DaamNShot, 0, scans&)
```

Set the scan configuration for channels 1 to 8 with a gain of ×1 in unipolar analog mode:

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&,
DafUniPolar&+DafAnalog&)
```

Set the post-trigger scan frequency:

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



Set the trigger source to a software trigger command; the rest of the parameters have no effect on a software trigger:

```
ret& = VBdaqAdcSetTrig&(handle&, DatsSoftware&, DatdRisingEdge&, 0,
HYSTERESIS%, 1)
```

Set the direct-to-disk filename with no pre-write, in append mode; also available is:

```
ret& = VBdaqAdcSetDiskFile&(handle&, "adcex8.bin", DaomAppendFile&, 0)
```

Start reading data in the background mode with cycle mode on and updateBlock:

```
ret& = VBdaqAdcTransferSetBuffer&(handle&, buf%(), BLOCK&, DatmCycleOn& +
DatmUpdateBlock&)
ret& = VBdaqAdcTransferStart&(handle&)
ret& = VBdaqAdcArm&(handle&)
ret& = VBdaqAdcSoftTrig&(handle&)
```

Monitor the progress of the transfer:

```
active& = -1
While active& <> 0
    ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
Print "Acquisition complete: "; retCount&; "scans acquired."
```

Close the device:

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

Now we convert the binary file to a text file. There is no simple way to do this, so it is necessary to open the file and manipulate it by hand.

First, open the binary file:

```
Open "adcex8.bin" For Input As 1
```

Next, get a handle for the file; this is one of the windows API calls, **CreateFile** (it doesn't actually create anything, however).

```
fileHandle& = CreateFile(binFile, GENERIC_READ, &H1, "", CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, "")
```

Now open the text output file where the converted data will be written:

```
Open "adcex8.txt" For Output As 2
```

Next, actually convert the binary data to text:

```
Do
```

Convert BLOCK unpacked scans to packed bytes:

```
scanCount& = BLOCK&
sampleCount& = scanCount& * channels&
wordCount& = sampleCount& * 3 / 4
byteCount& = 2 * wordCount&
```

Read the packed bytes from the input file, and get the number of bytes actually read. The **UBound()** and **Lbound()** functions just return the upper and lower bounds of the buffer. **Get #1** retrieves data from the file and stores it in the **buf()** array.

```
Dim sz&
sz& = UBound(buf%) - LBound(buf%)
For i& = 0 To sz&
    Get #1, i&, buf(i&)
Next i&
byteCount& = sz
```

Write the scans read and unpacked to the text file

```
For i& = 0 To scanCount& - 1
    For j& = 0 To channels& - 1
```

Send a tab between channels and a newline after each scan:

```
    If (j& < channels& - 1) Then
        termChar$ = Chr$(9)
    Else
        termChar$ = Chr$(13) + Chr$(10)
    End If
```


Calculate and write out the voltage value:

```
        voltage! = buf%(i& * channels& + j&) * 5! / 32768!  
        Print #2, Format$(voltage!, ".000") + termChar$;  
    Next j&  
Next i&
```

Print something so the program does not appear to be locked:

```
    Print ".";  
Loop While (byteCount& > 0) ' A byteCount of 0 indicates end-of-file  
' Close the input and output files  
Close 1  
Close 2  
Print "complete."
```

After program execution: data has been collected directly to disk in a binary file format, the TempBook device closed, the binary file was then opened, the data unpacked, and then written to a text file.

Transfers With Driver-Allocated Buffers

This example demonstrates the use of the new `daqAdcTransferBufData()` function. The following program reads scans of multiple channels in the background mode and uses a software trigger to start the acquisition. Functions used include:

- `VBdaqAdcArm&(handle&)`
- `VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)`
- `VBdaqAdcSetFreq&(handle&, freq#)`
- `VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, 1)`
- `VBdaqAdcSetTrig&(handle&, DatsSoftware&, 0,0,0,0)`
- `VBdaqAdcSoftTrig&(handle&)`
- `VBdaqAdcTransferBufData(handle&, userBuf(0), 1, DatmWait , retVal)`
- `VBdaqAdcTransferGetStat(handle, active, retCount);`
- `VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& + DatmUpdateSingle&)`
- `VBdaqAdcTransferStart(handle&)`
- `VBdaqClose(handle&)`
- `VBdaqOpen("TempBook0")`
- `VBdaqSetErrorHandler(handle&, 100)`

The constants used are defined as follows:

```
Const channels& = 8
Const scans& = 9
Const freq# = 200
```

As usual, the device is opened and the error handler is set up:

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC4
```

The acquisition is configured for 9 post-trigger scans and `Nshot` mode:

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

Set up the scan configuration for channels 1 to 9 with a gain of $\times 1$:

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&,
DgainX1&, 1)
```

Set the post-trigger scan rates:

```
ret& = VBdaqAdcSetFreq&(handle&, freq#)
```

Set the trigger source to a software trigger command; the other trigger parameters are not needed with a software trigger.

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

Now to set up the buffer for a background acquisition, in update single mode with cycle mode off.

```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& +
DatmUpdateSingle&)
```

Start the transfer, and trigger to begin transferring data:

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

Arm the acquisition:

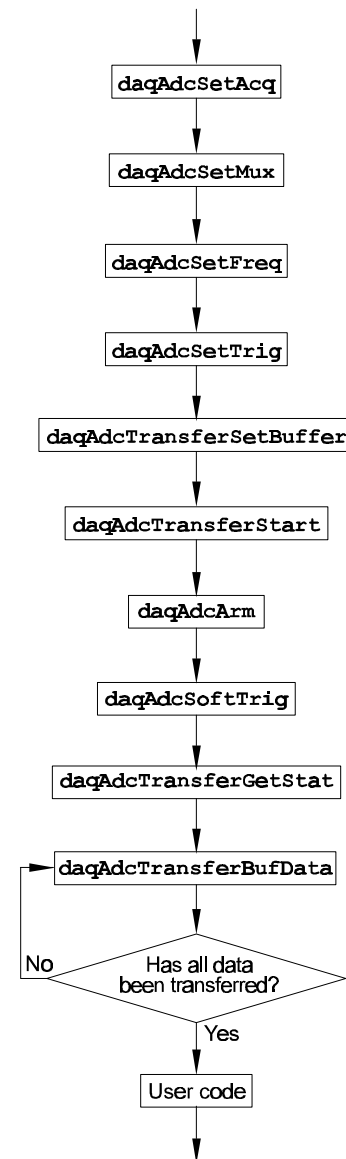
```
ret& = VBdaqAdcArm&(handle&)
```

Trigger the transfer:

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

Monitor the progress of the background transfer:

```
VBdaqAdcTransferGetStat(handle, active, retCount);
retCount=1;
while retCount<>0 do
    VBdaqAdcTransferBufData(handle&, userBuf(0), 1, DatmWait , retVal)
    print"Transfer in progress: ",retCount, "scans acquired."
    for i=0 to CHANS
```



```
        print userBuf(i)
        VBdaqAdcTransferGetStat(handle, active, retCount);
    next i
    print "Acquisition complete."
```

Now the data can be displayed or manipulated:

```
Print "Data acquired:"
  For i& = 0 To channels& - 1
    Print "Channel"; i& + 1; "Data:";
    For j& = 0 To scans& - 1
      Print Tab(j& * 7 + 17); buf%(j& * channels& + i&);
    Next j&
    Print
  Next i&
```

Finally, close the device:

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

Summary Guide of Selected Enhanced API Functions

The following table organizes the enhanced API functions by type and includes a brief description.

Simple One-Step Routines		
For single gain, consecutive channel, foreground transfers, use the following functions:		
Foreground Operation	Single Scan	Multiple Scans
Single Channel	<code>daqAdcRd</code>	<code>daqAdcRdN</code>
Consecutive Multiple Channels	<code>daqAdcRdScan</code>	<code>daqAdcRdScanN</code>
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.		
<code>daqAdcSetScan</code>	Set scan sequence using arrays of channel and gain values.	
<code>daqAdcSetMux</code>	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.		
<ul style="list-style-type: none"> • 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.		
<code>daqAdcSetTrig</code>	Configure the trigger event using source, level, rising and channel values.	
<code>daqAdcCalcTrig</code>	Using the selected trigger voltage, trigger direction, channel gain, and reference voltage, return the analog trigger source and value which can be used with <code>daqAdcSetTrig</code> .	
If a software trigger is selected, the start time of the scan depends on the application calling <code>daAdcSoftTrig</code> .		
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:		
<code>daqAdcSetRate</code>	Set/Get the specified frequency or period for the specified mode.	
<code>daqAdcSetFreq</code>	Set the pacer clock to the given frequency.	
A/D Acquisition		
A/D acquisition settings are not active until the acquisition is armed.		
<code>daqAdcArm</code>	Arm an A/D acquisition using the current configuration. If the trigger source was set to be immediate, the acquisition will be triggered immediately.	
<code>daqAdcDisarm</code>	Disarm the current acquisition if one is active. This command will disarm the current acquisition and terminate any current A/D transfers.	
<code>daqAdcSetAcq</code>	Define the mode of the acquisition and set the pre-trigger and post-trigger acquisition counts, 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:		
<code>daqAdcTransferSetBuffer</code>	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.	
<code>daqAdcTransferStart</code>	Start a transfer from the Daq* device to the buffer specified in the <code>daqAdcTransferSetBuffer</code> command	
<code>daqAdcTransferStop</code>	Stop a transfer from the Daq* device to the buffer specified in the <code>daqAdcTransferSetBuffer</code> command	
To find out whether a background A/D transfer is complete or to stop transfers, use the following function:		
<code>daqAdcTransferGetStat</code>	Return current A/D transfer status as well as a count representing the total number of transferred scans or the number of scans available.	

Overview

The first part of this chapter describes the TempBook 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. **Note:** The TempBook API is a subset of the Daq* API which also applies to other products; only TempBook-related commands are described here.

Beginning on page 11-35, several reference tables define parameters for: event-handling definitions, hardware definitions, ADC trigger-source and miscellaneous definitions, WBK card definitions, the API error codes, etc.

Function	Description	Page
Device Initialization Prototypes		
daqOpen	Open a session with the Daq* (including TempBook)	11-28
daqClose	End communication with the Daq* (including TempBook)	11-20
daqOnline	Check online status of the Daq* (including TempBook)	11-28
daqGetDeviceCount	Return the number of currently configured devices	11-26
daqGetDeviceList	Return the list of currently configured devices	11-26
daqGetDeviceProperties	Return the properties of specified device	11-27
Error Handler Function Prototypes		
daqSetDefaultErrorHandler	Set the default error handler	11-29
daqSetErrorHandler	Specify a user defined routine to call when an error occurs in any command	11-29
daqProcessError	Process a driver defined error condition	11-29
daqGetLastError	Return the last logged error condition	11-28
daqDefaultErrorHandler	Call the default error handler	11-25
daqFormatError	Return text string for specified error	11-26
Event Handling Function Prototypes		
daqSetTimeout	Set the time-out value for the Daq* operation (including TempBook)	11-30
daqWaitForEvent	Wait for specified Daq* device event (including TempBook)	11-32
daqWaitForEvents	Wait for multiple specified Daq* device events (including TempBook)	11-32
Utility Function Prototypes		
daqGetDriverVersion	Return the software version	11-27
daqGetHardwareInfo	Return the hardware version	11-27
Expansion Configuration Prototypes		
daqSetOption	Set options for a device's channel/signal path configuration	11-30
Custom ADC Acquisition Prototypes - Scan Sequence		
daqAdcSetMux	Configure a scan specifying start and end channels	11-13
daqAdcSetScan	Configure up to 256 channels making up an A/D or HS digital input scan	11-14
daqAdcGetScan	Read the current scan configuration	11-5
Custom ADC Acquisition Prototypes - Trigger		
daqAdcCalcTrig	Calculate the trigger level and trigger source for an analog trigger	11-4
daqAdcSetTrig	Configure an A/D trigger	11-15
daqAdcSoftTrig	Save a software trigger command to the DaqBook/DaqBoard	11-16
Custom ADC Acquisition Prototypes - Scan Rate and Source		
daqAdcSetRate	Configure the ADC scan rate with the mode parameter	11-13
daqAdcSetFreq	Configure the pacer clock frequency in Hz	11-12
daqAdcGetFreq	Read the current pacer clock frequency	11-5
Custom ADC Acquisition Prototypes - Scan Count, Rate and Source		
daqAdcSetAcq	Set acquisition configuration information	11-10
Custom ADC Acquisition Prototypes - Direct-to-Disk		
daqAdcSetDiskFile	Specify the disk file for direct-to-disk transfers	11-12
Custom ADC Acquisition Prototypes - Acquisition Control		
daqAdcArm	Arm an acquisition	11-2
daqAdcDisarm	Disarm an acquisition	11-4
Custom ADC Acquisition Prototypes - Data Transfer without Buffer Allocation		
daqAdcTransferBufData	Transfer scans from driver-allocated buffer to user-specified buffer	11-16
daqAdcTransferSetBuffer	Setup a destination buffer for an ADC transfer	11-18
daqAdcTransferStart	Start an ADC transfer	11-19
daqAdcTransferGetStat	Retrieve status of an ADC transfer	11-17

Function	Description	Page
daqAdcTransferStop	Stop an ADC transfer	11-19
Custom ADC Acquisition Prototypes - Buffer Manipulation		
daqAdcBufferRotate	Reorganize a circular buffer so that oldest data is oriented towards the front	11-3
One-Step ADC Acquisition Prototypes		
daqAdcRd	Configure an A/D acquisition and read one sample from a channel	11-6
daqAdcRdScan	Configure an A/D acquisition and read one scan	11-8
daqAdcRdN	Configure an A/D acquisition and read multiple scans from a channel	11-7
daqAdcRdScanN	Configure an A/D acquisition and read multiple scans	11-9
Data Format and Conversion Prototypes		
daqAdcSetDataForma	Set the raw and the post-acquisition data formats	11-11
daqCvtRawDataFormat	Convert raw data to a specified format	11-20
daqCvtSetAdcRange	Set the ADC Voltage Range for the conversion routines	11-21
Zero Offset Prototypes		
daqZeroSetup	Configure data for zero compensation	11-33
daqZeroConvert	Perform zero compensation on one or more scans	11-33
daqZeroSetupConvert	Perform both the setup and convert steps with one call	11-30
daqAutoZeroCompensate	Configure the thermocouple linearization functions to automatically perform zero compensation	11-19
Thermocouple Conversion Prototypes		
daqCvtTCConvert	Convert raw A/D readings from thermocouples to temperature readings	11-22
daqCvtTCSetup	Set up parameters for subsequent thermocouple temperature conversions	11-24
daqCvtTCSetupConvert	Set up and convert raw A/D readings from thermocouples into temperature readings	11-25
Test Prototypes		
daqTest	Perform a specified test on a Daq* device	11-31

Commands in Alphabetical Order

The following pages give details for each API command. Listed in alphabetical order, each section has 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 use a bold, mono-spaced **Courier** font to help distinguish characters that can be ambiguous in other fonts.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 $\times 8$ 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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

daqAdcSetDataFormat

DLL Function	<code>daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);</code>
C	<code>daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);</code>
Visual BASIC	<code>VBdaqAdcSetDataFormat (&(ByVal handle&, ByVal rawFormat&, ByVal postProcFormat&))</code>
Delphi	<code>daqAdcSetDataFormat(Handle:DaqHandleT; rawFormat:DaqAdcRawDataFormatT rawFormat; postProcFormat:DaqAdcPostProcDataFormatT);</code>
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	<code>DerrNoError</code> - No error (also, refer to <i>API Error Codes</i> on page 11-39)
See Also	<code>daqCvtRawDataFormat, daqCvtRawDataFormat</code>
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. The TempBook/66 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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
See Also	daqAdcTransferSetBuffer, daqAdcTransferGetStat
Program References	ADCEX9.C, ADCEX10.C
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
See Also	daqAdcTransferStart, daqAdcTransferStop, daqAdcTransferGetStat, daqAdcSetAcq, daqAdcTransferBufData
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi), ADCEX9.C, ADCEX10.C
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

daqAutoZeroCompensate

DLL Function	daqAutoZeroCompensate(BOOL zero);
C	daqAutoZeroCompensate(BOOL zero);
Visual BASIC	VBdaqAutoZeroCompensate&(ByVal zero&)
Delphi	daqAutoZeroCompensate(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 <i>API Error Codes</i> on page 11-39)
See Also	daqZeroSetup, daqZeroConvert, daqZeroSetupConvert, daqCvtTCSetup, daqCvtTCConvert, daqcvtTcSetupConvert
Program References	None
Used With	All devices

daqAutoZeroCompensate will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the TempBook. 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 18 using the TempBook CJC gain code (CJC zero)
- channel 18 using the gain code of the connected TC (TC zero)
- channel 16 using the TempBook 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.

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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
See Also	
Program References	None
Used With	All devices

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 device. 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 <i>API Error Codes</i> on page 11-39)
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 TempBook 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 TempBook:

- 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 TempBook. The auto-zero feature is enabled by default, but can be disabled using the **daqAutoZeroCompensate** 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 TempBook 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 TempBook, 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.

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

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

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	TbkUniCJC	TgainX100	TbkBiCJC	TgainX50
J	TbkUniTypeJ	TgainX200	TbkBiTypeJ	TgainX100
K	TbkUniTypeK	TgainX200	TbkBiTypeK	TgainX100
T	TbkUniTypeT	TgainX200	TbkBiTypeT	TgainX200
E	TbkUniTypeE	TgainX100	TbkBiTypeE	TgainX50
N28	TbkUniTypeN28	TgainX200	TbkBiTypeN28	TgainX100
N14	TbkUniTypeN14	TgainX200	TbkBiTypeN14	TgainX100
S	TbkUniTypeS	TgainX200	TbkBiTypeS	TgainX200
R	TbkUniTypeR	TgainX200	TbkBiTypeR	TgainX200
B	TbkUniTypeB	TgainX200	TbkBiTypeB	TgainX200

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 <i>Scan Setup</i> . 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, TbkTCTYPEJ , TbkTCTYPEK , TbkTCTYPET , TbkTCTYPEE , TbkTCTYPEN28 , TbkTCTYPEN14 , TbkTCTYPES , TbkTCTYPER or TbkTCTYPEB .
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 <i>API Error Codes</i> on page 11-39)
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: 16 18, if auto-zeroing is used with the TempBook.
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 <i>API Error Codes</i> on page 11-39)
See Also	DaqCvtTCSetup, daqCvtTCConvert
Program References	None
Used With	All devices

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

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 <i>API Error Codes</i> at end of this chapter).
Returns	Nothing (also, refer to <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 11-39.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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* chapter) 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 <i>API Error Codes</i> on page 11-39
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
See Also	daqSetDefaultErrorHandler, daqDefaultErrorHandler, daqGetLastError, daqProcessError
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS (Delphi)
Used With	All devices

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
See Also	daqWaitForEvents, daqSetTimeout
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 **daqSetTimeout** 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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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.

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 <i>API Error Codes</i> on page 11-39)
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 <i>API Error Codes</i> on page 11-39)
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
Daq Device Property Definitions - daqGetDeviceProperties	Identifies the format (DWORD, STRING, or FLOAT) for property parameters	11-36
Event-Handling Definitions	Transfer Event Definitions - DaqTransferEvent Transfer Event Wait Mode Definitions - DaqWaitMode	11-36
Hardware Information Definitions	Hardware Information Selector Definitions - DaqHardwareInfo Hardware Version Definitions - DaqHardwareVersion	11-36
ADC Trigger Source Definitions	DaqAdcTriggerSource DaqEnhTrigSensT	11-37
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	11-37
TempBook Definitions	Unipolar Thermocouple Gain Definitions Bipolar Thermocouple Gain Definitions Thermocouple Definitions Voltage Gain Definitions	11-38
General I/O Definitions	I/O Operation Code Definitions - DaqIOOperationCode	11-38
DaqTest Command Definitions	DaqTestCommand	11-38
Calibration Input Signal Sources	DaqCalInputT DaqCalTableTypeT	11-38
API Error Codes	Identifies API errors by number and description	11-38

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

Event-Handling Definitions

<i>Transfer Event Definitions - daqTransferEvent</i>		<i>Transfer Event Wait Mode Definitions - daqWaitMode</i>	
DteAdcData	0	DwmNoWait	0
DteAdcDone	1	DwmWaitForAny	1
DteDacData	2	DwmWaitForAll	2
DteDacDone	3		
DteIOData	4		
DteIODone	5		

Hardware Information Definitions

<i>Hardware Information Selector Definitions - daqHardwareInfo</i>		<i>Hardware Version Definitions - daqHardwareVersion</i>	
Definition	Value	Definition	Value
DhiHardwareVersion	0	DaqBook100	0
DhiProtocol	1	DaqBook112	1
DhiAdcBits	2	DaqBook120	2
DhiADmin	3	DaqBook200	3
DhiADmax	4	DaqBook216	4
		DaqBoard100	5
		DaqBoard112	6
		DaqBoard200	7
		DaqBoard216	8
		Daq112	9
		Daq216	10
		WaveBook512	11
		WaveBook516	12
		TempBook66	13

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

<i>ADC Flag Definitions - daqAdcFlag</i>					
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

<i>Frequency vs Period - daqAdcRateMode</i>		<i>ADC Acquisition Mode Definitions - daqAdcAcqMode</i>		<i>ADC Transfer Mask Definitions - daqAdcTransferMask</i>	
DarmPeriod	0	DaamNShot	0	DatmCycleOff	00h
DarmFrequency	1	DaamNShotRearm	1	DatmCycleOn	01h
		DaamInfinitePost	2	DatmUpdateBlock	00h
		DaamPrePost	3	DatmUpdateSingle	02h
				DatmWait	00h
				DatmReturn	04h
				DatmUserBuf	00h
				DatmDriverBuf	08h

<i>ADC Clock Source Definitions - daqAdcClockSource</i>		<i>ADC File Open Mode Definitions - daqAdcOpenMode</i>		<i>ADC Acquisition/Transfer Active Flag Definitions - daqAdcActiveFlag</i>	
DacsAdcClock	0	DaomAppendFile	0	DaafAcqActive	01h
DacsGatedAdcClock	1	DaomWriteFile	1	DaafAcqTriggered	02h
DacsTriggerSource	2	DaomCreateFile	2	DaafTransferActive	04h

<i>ADC Acquisition State - daqAdcAcqState</i>		<i>ADC Buffer Transfer Mask - daqAdcBufferXferMask</i>	
DaasPreTrig	0	DabtmOldest	1
DaasPostTrig	1	DabtmNewest	2
		DabtmWait	3
		DabtmReturn	4

TempBook Definitions

<i>Unipolar Thermocouple Gain Definitions</i>		<i>Bipolar Thermocouple Gain Definitions</i>	
TbkUniCJC	TgainX100	TbkBiCJC	TgainX50
TbkUniTypeJ	TgainX200	TbkBiTypeJ	TgainX100
TbkUniTypeK	TgainX200	TbkBiTypeK	TgainX100
TbkUniTypeT	TgainX200	TbkBiTypeT	TgainX200
TbkUniTypeE	TgainX100	TbkBiTypeE	TgainX50
TbkUniTypeN28	TgainX200	TbkBiTypeN28	TgainX100
TbkUniTypeN14	TgainX200	TbkBiTypeN14	TgainX100
TbkUniTypeS	TgainX200	TbkBiTypeS	TgainX200
TbkUniTypeR	TgainX200	TbkBiTypeR	TgainX200
TbkUniTypeB	TgainX200	TbkBiTypeB	TgainX200

<i>Thermocouple Definitions</i>		<i>Voltage Gain Definitions</i>	
TbkTCTypeJ	18	TgainX1	0
TbkTCTypeK	19	TgainX2	1
TbkTCTypeT	20	TgainX5	2
TbkTCTypeE	21	TgainX10	3
TbkTCTypeN28	22	TgainX20	5
TbkTCTypeN14	23	TgainX50	6
TbkTCTypeS	24	TgainX100	7
TbkTCTypeR	25	TgainX200	11
TbkTCTypeB	26		

General I/O Definitions

<i>I/O Operation Code Definitions - daqIOOperationCode</i>	
DiocReadByte	0
DiocWriteByte	1
DiocReadWord	2
DiocWriteWord	3
DiocReadDWord	4
DiocWriteDWord	5

daqTest Command Definitions

<i>DaqTestCommand</i>	
DtstBaseAddressValid	0
DtstInterruptLevelValid	1
DtstDmaChannelValid	2
DtstAdcFifoInputSpeed	3
DtstDacFifoOutputSpeed	4
DtstIOInputSpeed	5
DtstIOOutputSpeed	6

Calibration Input Signal Sources

<i>DaqCalInputT</i>		
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
<i>DaqCalTableTypeT</i>		
DcttFactory	0	Factory calibration constants
DcttUser	1	User-defined calibration constants

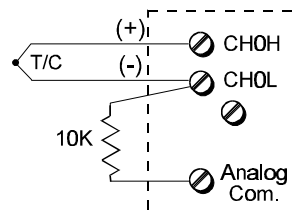
API Error Codes

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	<code>dacTCConvert</code> called before <code>dacTCSetup</code>
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	<code>dac...Convert</code> called before <code>dac...Setup</code>
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	<code>rtdValue</code> out-of-range
DerrRtdNoSetup	3Ch - 60	<code>rtdConvert</code> called before <code>rtdSetup</code>
DerrRtdArraySize	3Dh - 61	Temperature array not large enough
DerrRtdParam	3Eh - 62	Incorrect RTD parameter

Error Name	Code # hex - dec	Description
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

Floating Differential

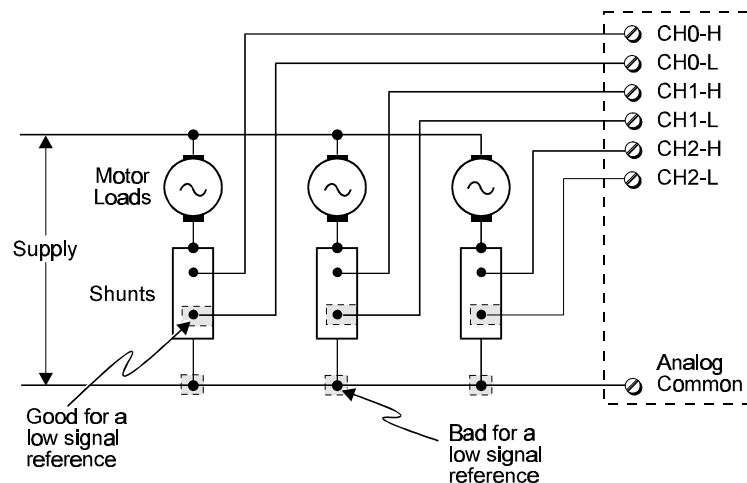
Floating differential measurements are generally made when low-level signals must be measured in the presence of relatively high levels of common mode noise. The most common example would be a non-grounded thermocouple. When the signal source has no direct connection to the system analog common, one must be provided. The easiest way to do this on the TempBook is to use the built-in biasing resistors by closing the corresponding input channel DIP switches (S1 and S2 on the termination card as described in chapter 2). As an alternative, a resistor can be connected between one of the two signal lines (usually the lower in potential) and common. A resistor of 10 K Ω to 100 K Ω is satisfactory (less noise with the lower values).



Floating Differential Configuration

Referenced Differential

Referenced differential measurement is used when measuring several voltages. However, the voltages cannot share the same common node as a group of single-ended measurements might. An example is to measure several currents via shunts which happen to share a common supply terminal along a bus. The weakness in a single-ended hookup is that current flow through the common bus can introduce substantial errors that can be indistinguishable from the real values measured. The need for the system common to be connected to the common supply terminal is shown in the figure.



Referenced Differential Configuration



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 1997 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