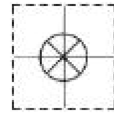


User's Guide

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

OMB-WAVEBOOK-512

High Speed Portable Digitizer

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

Servicing North America:

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

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

For immediate technical or application assistance:

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

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

Servicing Europe:

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

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

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

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

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

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

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

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

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

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

How To Use This Manual

This manual explains the setup and operation of the WaveBook data acquisition system including WBK option cards and modules. This manual is divided into a table of contents, 12 chapters, and an appendix. The chapters are briefly described as follows:

Chapter 1 - ***Getting Started*** gives an overview of the basic features of a WaveBook system. A "quick start" section outlines how to start up a simple system. Application programmers are advised on the use of various software drivers that come with the WaveBook.

Chapter 2 - ***Installation, Configuration, and Calibration*** explains in detail how to set up a WaveBook system including hardware, software, configuration, system testing and calibration.

Chapter 3 - ***WaveBook Expansion Options*** describes the various WBK devices that work with the WaveBook for expansion, signal conditioning, interfacing, and power supply. This chapter begins by explaining WBK power management and then proceeds to discuss the WBK cards and modules including block diagrams, setup procedures, operation, and software concerns.

Chapter 4 - ***Operation Guide*** explains the system's theory of operation. Basic concepts of data acquisition with a WaveBook are explained, including: managing the acquisition process, initialization and configuration, specifying the scan, processing analog signals and digital I/O, trigger types and capabilities, and data transfer. **Note**: many of the operations are explained using software commands from the standard Application Programming Interface (API).

Chapter 5 - ***Using WaveView*** explains the ready-to-use software that is included with your system. This chapter shows you how to install and configure the software to meet your system requirements. Numerous figures show screen shots related to all aspects of system operation.

Programmer s Note: Chapters 6 to 12 are for use by programmers; if you are using WaveView or another ready-to-use program, you can skip these chapters. The comparative features of the standard and enhanced APIs are described in the last section of chapter 1. If using the standard API, read the appropriate language chapter (6 to 9) and chapter 10. If using the enhanced API, read chapters 11 and 12. Chapter 4 explains system operation in terms of standard API commands.

Chapter 6 - ***Using WaveBook/512 with C*** explains how to program basic data acquisition tasks with sample programs in C using the standard API.

Chapter 7 - ***Using WaveBook/512 with QuickBASIC*** explains how to program basic data acquisition tasks with sample programs in QuickBASIC using the standard API.

Chapter 8 - ***Using WaveBook/512 with Turbo Pascal*** explains how to program basic data acquisition tasks with sample programs in Turbo Pascal using the standard API.

Chapter 9 - ***Using WaveBook/512 with VB Subroutines Support*** explains how to program basic data acquisition tasks with sample programs in Visual Basic using the standard API.

Chapter 10 - ***Command Reference (Standard API)*** explains all **`wbk`**…commands and their related parameters.

Chapter 11 - ***Enhanced API Programming Models (WaveBook)*** describes the fundamental building blocks for data acquisition software using the enhanced API. 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 modification) be used in your code.

Chapter 12 - ***WaveBook Command Reference (Enhanced API)*** explains all the **`daq`**…commands and parameters that pertain to WaveBook.

Appendix: Accessories* and *Specifications includes a list of available accessories and the physical and performance specifications for the WaveBook/512 and related WBKs.

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.

Table of Contents

1 Getting Started

General Description 1-1
Front Panel of the WaveBook/512 1-1
Rear Panel of the WaveBook/512 1-1
System Features 1-1
Quick Start 1-2
PC Connection 1-2
Signal Connections 1-3
Power Connections 1-3
Software Installation 1-3
Testing the System 1-4
WaveView Overview 1-4
Driver Options (for programmers only) 1-5
Standard API 1-5
Enhanced API 1-6
Language Support 1-6

2 Installation, Configuration, and Calibration

Inspection 2-1
Hardware Installation 2-1
Connecting to Desktop PCs 2-1
Standard Parallel-Port Connection 2-1
High-Speed Parallel-Port (WBK21) Connection 2-1
Connecting to Laptop PCs 2-2
Standard Parallel-Port Connection 2-2
PCMCIA Card (WBK20) Connection 2-2
Analog-Signal Connections and Proper Grounding 2-2
Digital I/O Connection 2-3
Optional Printer Connection 2-3
WaveBook Software Installation 2-4
Making a Backup Copy 2-4
Adding WaveBook Reset to AUTOEXEC.BAT 2-4
Installation Under Windows 3.1 2-4
Using WBKTest 2-5
Installation Under Windows 95 and Windows NT 2-5
WaveBook Configuration Under Windows 95/NT 2-7
Resource Tests 2-8
Performance Tests 2-9
Connection Troubleshooting (Windows 95/NT) 2-9
Calibration 2-10
Manual Calibration 2-10
WaveCal 2-10
Using WaveCal 2-11
Performing System Calibration 2-12

3 Wavebook Expansion Options

Overview 3-1
Power Requirements 3-1
Power Connector Pinout 3-2
DBK30A Rechargeable Battery Module 3-3
Description 3-3
Hardware Setup 3-4
Configuration 3-4
Connection 3-4
Connections for the DBK31 Mode 3-4
Charging the Battery Module 3-5
Using the Battery Module While Charging 3-5
DBK30A - Specifications 3-5
WBK10 - Expansion Module 3-6
Description 3-6

WBK10 Front Panel.....3-6
WBK10 Rear Panel.....3-7
Hardware Setup.....3-7
Configuration.....3-7
WBK10 Cabling.....3-7
Software Setup in WaveView.....3-9
WBK10 - Specifications.....3-9
WBK11 - Simultaneous Sample & Hold Card.....3-10
Description.....3-10
WBK11 Hardware Setup.....3-10
WBK11 - Specifications.....3-11
WBK12/13 Programmable Low-Pass Filter Cards.....3-12
Description.....3-12
Hardware Setup.....3-13
Configuration.....3-13
Connection.....3-13
Software Setup in WaveView.....3-13
Software Function.....3-13
WBK12/13 - Specifications.....3-14
WBK14 Dynamic Signal Input Module.....3-15
Introduction.....3-15
Hardware Description.....3-15
Current Source.....3-15
High-Pass Filter (HPF).....3-15
Programmable Gain Amplifier (PGA).....3-16
Programmable Low-Pass Filter Phase Equalizer (PLPPE).....3-16
Programmable Low-Pass Anti-Aliasing Filter (PLPAF).....3-16
Simultaneous Sample and Hold.....3-16
Excitation Source.....3-16
Power.....3-16
Calibrating the WBK14.....3-16
Hardware Setup.....3-17
Configuration.....3-17
Connection.....3-17
Software Setup in WaveView.....3-17
Software Function.....3-18
Accelerometer Tutorial.....3-18
Accelerometer Specification Parameters.....3-19
Electrical Grounding.....3-20
Cable Driving.....3-20
WBK14 - Specifications.....3-21
WBK15 5B Isolated Signal-Conditioning Module.....3-22
Description.....3-22
Hardware Setup.....3-22
Safety Concerns.....3-23
Power.....3-23
Configuration.....3-23
5B Module Insertion/Removal.....3-24
Signal Connection.....3-25
Software Setup in WaveView.....3-26
Software Function.....3-26
WBK15 - Specifications.....3-26
WBK20 - PCMCIA/EPP Interface Card.....3-27
WBK20 - Specifications.....3-27
WBK21 - ISA/EPP Interface Card.....3-27
WBK21 - Specifications.....3-27
WBK61 & WBK62 High-Voltage Adapters.....3-28
Hardware Setup.....3-28
Software Setup in WaveView.....3-30
Software Function.....3-30
WBK61/62 - Specifications.....3-30

4 Operation Guide

System Overview 4-1
The Acquisition Process....................4-3
Initializing the WaveBook....................4-3
One-Step Acquisitions....................4-4
Configuring an Acquisition....................4-4
Channel Numbering....................4-4
Specifying the Scan....................4-4
Specifying the Trigger Source....................4-4
Specifying the Number of Scans....................4-4
Specifying the Scan Rate....................4-5
Starting the Acquisition....................4-5
Transferring Results....................4-5
Stopping the Acquisition....................4-6
Shutting Down the WaveBook....................4-6
Operation Details....................4-6
Analog Signal Processing....................4-6
Digital I/O Processing....................4-6
Acquisition Composition....................4-8
Acquisition Mode and the Number of Scans....................4-8
Scan Composition....................4-9
Scan Period....................4-9
Triggering Capabilities....................4-10
Low-latency (Hardware) Triggering....................4-10
DSP-Based, Multi-Channel Triggering....................4-11
Eight Trigger Types....................4-12
Trigger Latency and Jitter....................4-14
Data Packing....................4-15
Packed-Data Format....................4-15
Data Transfer....................4-16
Time-outs....................4-16
Buffer Size....................4-16
Overlapped Execution....................4-18
Foreground-Linear Transfers....................4-19
Foreground-Cycle Transfers....................4-19
Background-Linear Transfers....................4-20
Background-Cycle Transfers....................4-20
Direct-to-Disk Transfers....................4-21

5 Using WaveView

Application Startup....................5-1
Loading WaveView....................5-1
Starting WaveView....................5-1
Simulated WaveBook....................5-1
WaveBook Attached....................5-2
WaveView Main Components....................5-2
Sample Acquisition Using WaveView....................5-4
WaveView Configuration Menu Items & Buttons....................5-5
File....................5-5
Edit....................5-5
Window....................5-6
System....................5-6
WaveView Configuration Screen Components....................5-7
Input Channel Configuration....................5-7
Scan and Trigger Configuration....................5-9
Scan Count....................5-9
Scan Rate....................5-9
Trigger....................5-10
WaveView Scope Menu Items & Buttons....................5-12
File....................5-12
Acquire....................5-13
Charts....................5-13
Options....................5-14

Window 5-14
WaveView Scope Display 5-15
WaveView Direct-To-Disk Menu Items & Buttons 5-16
File 5-16
Acquire 5-16
Window 5-17
WaveView Direct-to-Disk Data Destination Box 5-17
Using PostView 5-18
PostView Timebase 5-18
PostView Menu Items 5-18
File 5-18
Number of Charts 5-18
Go To 5-18
Options 5-18
Help 5-19
PostView Display 5-19

6 Using WaveBook/512 With C

Borland C for DOS 6-1
Microsoft C for DOS 6-1
Borland C for Windows 6-1
Microsoft C for Windows 6-2
Initializing WaveBook Communications 6-2
Error Handling 6-2
One-Step Analog Input 6-3
Low-Level Analog Input 6-4
Accessing the High-Speed Digital Input Port 6-5
Background Processing of Analog Input 6-6
Complex Triggering 6-7
Pre- and Post-Trigger Acquisitions 6-8
Buffer Management 6-9
Direct-to-Disk 6-11
Sample Programs 6-13

7 Using WaveBook/512 With QuickBASIC

Using Multiple Quick Libraries with QuickBASIC 7-1
Simple Analog Input 7-1
Low-Level Analog Input 7-3
Accessing the High-Speed Digital Input Port 7-4
Background Processing of Analog Input 7-6
Complex Triggering 7-7
Pre- and Post-Trigger Acquisitions 7-9
Buffer Management 7-11
Sample Programs 7-13

8 Using WaveBook/512 With Turbo Pascal

Simple Analog Input 8-1
Low-Level Analog Input 8-2
Accessing the High-Speed Digital Input Port 8-4
Background Processing of Analog Input 8-6
Complex Triggering 8-7
Pre- and Post-Trigger Acquisitions 8-9
Buffer Management 8-11
Sample Programs 8-13

9 Using WaveBook/512 With VB Subroutines Support

Accessing WaveBook from a Windows Program 9-1
Accessing WaveBook from a Visual Basic Program 9-1
Simple Analog Input 9-1
Low-Level Analog Input 9-3
Accessing the High-Speed Digital Input Port 9-4

Background Processing of Analog Input 9-5
Complex Triggering 9-7
Pre- and Post-Trigger Acquisitions 9-8
Buffer Management 9-10
Direct-to-Disk 9-12
Sample Programs 9-14

10 Command Reference (Standard API)
Overview 10-1
Commands in Alphabetical Order 10-2
API Reference Tables 10-24

11 Enhanced API Programming Models (WaveBook)
Overview 11-1
Data Acquisition Environment 11-1
Application Programming Interface 11-1
Enhanced vs Standard API 11-1
Hardware Capabilities and Constraints 11-1
Signal Environment 11-2
Basic Models 11-2
Initialization and Error Handling 11-3
Foreground Acquisition with One-Step Commands 11-5
Counted Acquisitions Using Linear Buffers 11-7
Indefinite Acquisition, Direct-To-Disk Using Circular Buffers 11-9
Multiple Hardware Scans, Software Triggering 11-12
Background Acquisition 11-14
Complex Triggering 11-16
Data Packing and Rotating 11-18
Double Buffering 11-20
Direct-To-Disk Transfers 11-22
Transfers With Driver-Allocated Buffers 11-25
Summary Guide of Selected Enhanced API Functions 11-27

12 WaveBook Command Reference (Enhanced API)
Overview 12-1
Commands in Alphabetical Order 12-2
API Reference Tables 12-38

Appendix: Accessories and Specifications
Available Accessories A-1
Specifications A-1
WaveBook/512 - Specifications A-2
DBK30A - Specifications A-2
WBK10 - Specifications A-3
WBK11 - Specifications A-3
WBK12/13 - Specifications A-3
WBK14 - Specifications A-4
WBK15 - Specifications A-4
WBK20 - Specifications A-5
WBK21 - Specifications A-5
WBK61/62 - Specifications A-5

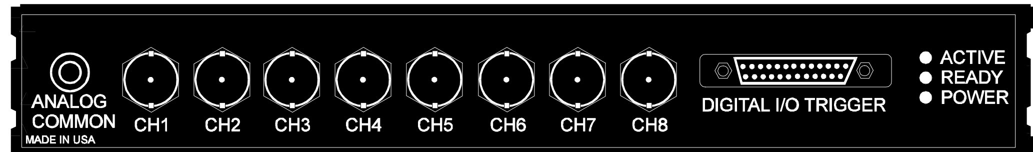
Getting Started 1

This short introductory chapter:

- Describes the WaveBook/512 controls and connectors
- Describes the system's basic features
- Offers guidelines for a "quick start" including setup and operation
- Explains software driver options for programmers

General Description

Front Panel of the WaveBook/512

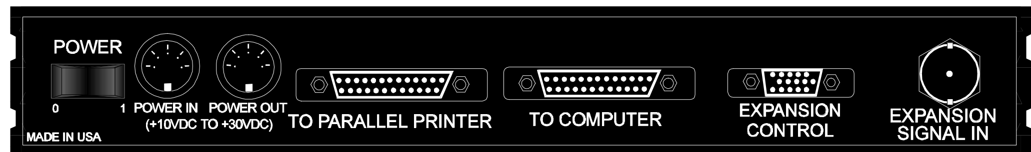


WaveBook/512 Front Panel Layout

The front panel of the WaveBook/512 has the following components (left to right):

- 1 Binding post for ANALOG COMMON reference
- 8 BNC connectors for analog inputs (analog channel 1 is also the low-latency analog trigger)
- 1 D-25F for digital I/O, including trigger input
- 3 Status LEDs (ACTIVE, READY, POWER)

Rear Panel of the WaveBook/512



WaveBook/512 Rear Panel Layout

The rear panel of the WaveBook/512 has the following components (left to right):

- POWER switch (0-off; 1-on)
- 2 circular 5-pin DIN connectors for POWER IN and POWER OUT (pass-through)
- 1 DB25F for TO PARALLEL PRINTER (pass-through)
- 1 DB25M for LPT/EPP host (TO COMPUTER) connection
- 1 HD-15F EXPANSION CONTROL output
- 1 BNC connector for analog input EXPANSION SIGNAL IN

System Features

The WaveBook/512 is a DSP-based, 12-bit, 1 million sample-per-second portable data acquisition system for notebook and desktop PCs. The 8-channel instrument is designed for applications that require high resolution and/or high-speed signal capture, such as engine strain testing, multi-channel acoustical testing, mechanical integrity testing, and vibration/shock/strain testing.

Specific features include the following:

- **Expansion Options**. Using 8-channel expansion chassis (the WBK10) allows system expansion up to 72 analog channels. Each 8-channel chassis may be equipped with the field-installable WBK11 simultaneous-sample-and-hold card which minimizes inter-channel skew and adds additional, higher-sensitivity gain ranges. Various WBK option cards for a variety of applications are described in chapter 3.
- **Power Options**. The WaveBook can be powered from the included AC adapter or a 10 to 30 VDC source such as a car battery or the DBK30A rechargeable-battery module. This flexibility for input power makes the WaveBook ideal for portable, field, and/or bench-top applications.

- **Easy PC Connection**. The WaveBook connects to a notebook PC via the enhanced parallel port (EPP) or an optional PCMCIA interface (WBK20). The WaveBook may also be connected to a desktop PC via the EPP port or an optional ISA plug-in card interface (WBK21). Standard non-EPP ports may also be used but at reduced throughput (also, a printer can be connected in pass-through mode).
- **Digital Signal Processing (DSP)**. The DSP-based design allows you to define an arbitrary scan sequence of channels and associated gains across all of its channels. The DSP also provides real-time digital calibration on a per-sample basis, eliminating the need for manual adjustments and allowing expansion of signal-capacity and installation of options without recalibration. The WaveBook permits a programmable trigger based on the signal levels of any combination of analog channels.
- **Programmable Scan Sequencing**. A 128-location scan sequencer allows you to program the analog channel scan sequence, the associated unipolar/bipolar A/D range, and the input amplifier gain. The unit performs 1 Msample/s scanning and gain switching over both its built-in and expansion channels (up to its limit of 72). The WaveBook/512 can also be switched from unipolar to bipolar operation on a per-channel basis at the full sampling rate (every 1 µs). The WaveBook/512 takes readings in scans of up to 128 individual readings. Within a scan, readings are taken at a fixed 1 µs rate. The time between scans is adjustable to provide varying sample rates. The start of each scan is delayed from the start of the previous scan by a programmable interval (settable in 0.05 µs (50 ns) increments, subject to the 1-µs minimum interval between samples and a 100-s maximum interval).
- **Multi-channel triggering**. Multi-channel triggering allows you to program any combination of analog input channels including expansion channels as the analog trigger. You can also program trigger-level slope and hysteresis for each trigger channel and then combine multiple trigger channels in a logical "and" or "or" function to determine whether an actual trigger condition has occurred. The DSP initiates sampling of the trigger channels, calibrates incoming data, compares readings to pre-programmed trigger states, and then determines whether the trigger conditions are satisfied. All these operations occur in 2 µs plus the time required to sample each trigger channel (1 µs/channel).
- **Single-channel triggering**. The WaveBook/512 is also capable of low-latency single-channel triggering using either analog input channel 1 or a TTL-trigger input. The analog trigger level is programmable with 12-bit resolution, and both trigger inputs may be set for either rising- or falling-edge detection.
- **Pre- and post-trigger readings**. The WaveBook/512 can acquire readings while waiting for a trigger and after the trigger has been detected. The pre-trigger readings may be sampled at a different rate than the post-trigger readings. At the end of a scan of readings or the end of a series of scans, the system will return to its original scan rate and may be re-armed for the next trigger event.
- **Digital Inputs**. The unit has the ability to sample 8 TTL-level digital inputs as part of the user-defined scan sequence and can acquire the state of all 8 digital input lines in 1 µs. This allows digital data to be time-correlated with the acquired analog data. The WaveBook can also write to the digital I/O port under program control. Digital inputs are readable at up to 1 Mbyte/s.
- **Analog Inputs**. BNC connectors provide convenient signal termination and ensure signal integrity. To prevent inadvertent ground loops, these connectors isolate the chassis from analog signals, commons, and shields.

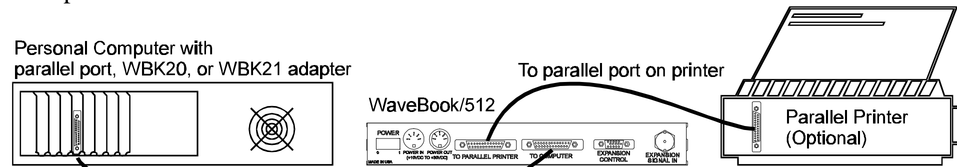
Quick Start

These Quick Start guidelines can be used for very simple systems or as a preview for more complex systems as described in the next chapter (*Installation and Configuration*). **Note**: Most users will need to refer to other chapters for specific details on setup and operation.

PC Connection

The WaveBook/512 communicates with a laptop or desktop computer through the parallel printer port or a port adapter (for the WBK20 or the WBK21, refer to their separately-supplied instructions).

Connect the supplied cable to the computer's parallel port or adapter and then to the WaveBook port marked "TO COMPUTER". Optionally (for printer pass-through capability), connect your printer's cable to the port marked "TO PARALLEL PRINTER".



PC-to-WaveBook/512 Connection

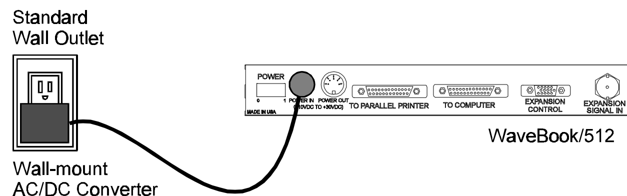
Signal Connections

The front panel has 8 BNC connectors for analog inputs, a binding post for analog common, and one D-25F connector for Digital I/O. Channel 1 is also used for low-latency analog triggering.

The center pin of each BNC connector is the high or positive input, and the outer shell is the low or negative input. The inputs are differential: the measured voltage is the difference between the high and low signal levels. For proper operation each analog input signal (high or low) must be within ±11 volts of the WaveBook's analog common level. An analog common connection is provided by the front panel binding post. If the analog signal source is floating (does not connect to a common ground) with respect to the WaveBook, then it may be necessary to connect the analog signal source to the analog ground binding post to keep the analog signals within the ±11 volt common-mode range. Analog common is at the same potential as the PC's digital ground. The analog channels are not isolated from the PC. In contrast, the WaveBook's power supply input is isolated from the rest of the WaveBook, including the analog ground and the PC’s digital ground. **Note**: grounding is explained more fully in chapter 2.

Power Connections

The WaveBook can be powered from the supplied wall-mount AC-to-DC converter or from the optional DBK30A battery module. The wall-mount converter (TR-27 or TR-27E for European applications) plugs into a standard wall outlet; its other end plugs into the circular DIN5 receptacle on the WaveBook's rear panel (the DIN5 pinout can be found under *Power Requirements* in chapter 3). If using the battery module, refer to the DBK30A section of chapter 3.



Power Connection

Software Installation

The WaveBook system includes a Windows program called WaveView that provides an easy way to collect data from the WaveBook/512. To install WaveView, insert software disk 1 into your floppy disk drive, and choose RUN from the Windows Program Manager FILE menu and type `A(or B):\SETUP.EXE`. Follow the on-screen prompts; setup is fairly intuitive whether using the 16-bit WaveView with Windows 3.1 or the 32-bit WaveView with Windows 95/NT (refer to chapter 5 if necessary).

The user should add the following line to the `autoexec.bat` file on the host PC: `c:\wavebook\wbkreset`. This command performs a reset of the WaveBook whenever the PC is restarted and is especially important if a printer is attached through the WaveBook/512.

Testing the System

After software installation, the hardware must be tested to verify communication between the PC and the WaveBook. The 16-bit software includes `WBKTest.exe` to test hardware performance (the 32-bit software has similar performance tests). This testing verifies the PC's parallel port (or WBK20/21 adapter) capabilities and then estimates the port's maximum performance, using both standard and enhanced protocols. Testing also verifies that the WaveBook is properly attached and ready to operate.

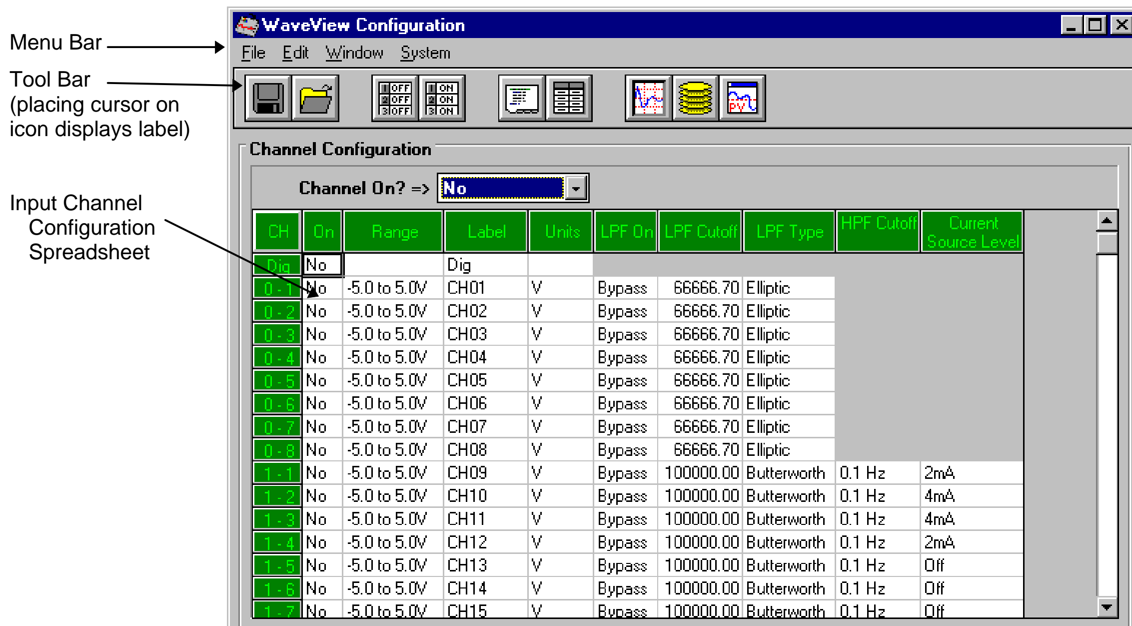
To run the program, move to the WaveBook directory and type `WBKTest`. The program will perform several tests on the PC and WaveBook and then print out the results. Once WBKTest successfully communicates with the WaveBook/512, WaveView may be used to collect readings.

WaveView Overview

Note: Chapter 5 explains WaveView in detail; the following is intended only as a "quick start".

WaveView is a Windows application for operating the WaveBook system. No programming is required to use WaveView. It allows users to acquire data for immediate viewing or for storage to the PC's hard disk.

Start WaveView by (double)clicking on its program icon to display the main window (see figure). WaveView interrogates the hardware after it starts up to see what options and expansion modules are actually connected to the WaveBook. The total number of channels displayed on the configuration menu corresponds to the number of channels connected.



Configure Channels

To begin acquiring data with WaveView, turn on only those channels to which you have signals connected. This can be done by clicking once in the On column next to the desired channel number and selecting "YES" or "NO" from the listbox above the spreadsheet. Turning a channel on will cause the channel to be sampled during an acquisition.

Select the appropriate parameters for each channel. Note that the spreadsheet entries can be changed for all channels by clicking once on the column label at the top of the spreadsheet to highlight the column and then making the appropriate entry within the selection box that appears as needed above the spreadsheet.

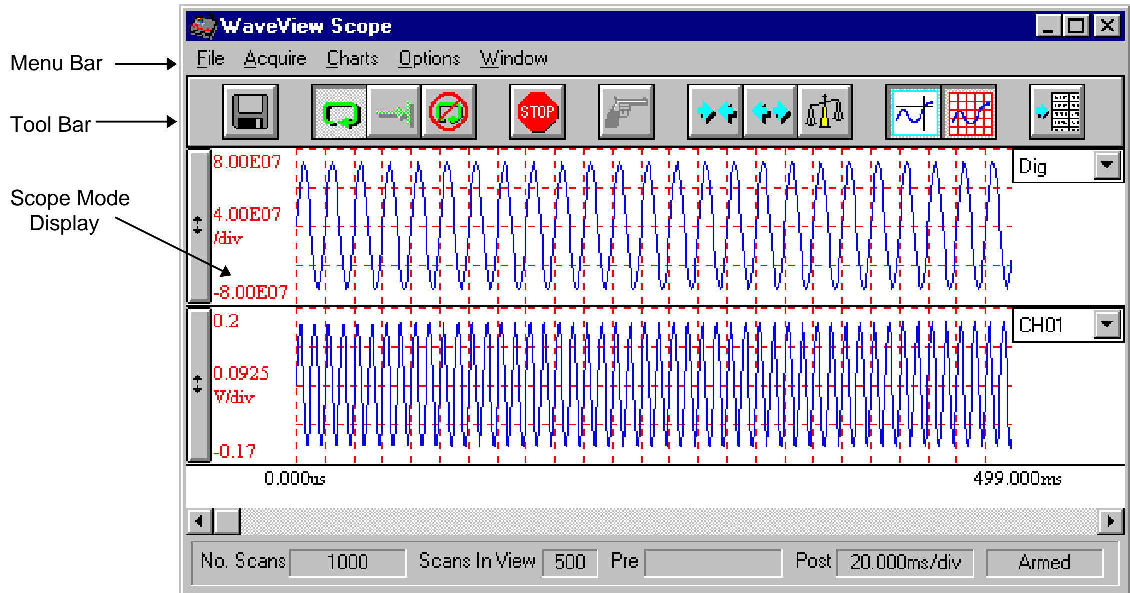
- Gain and offset are selected by choosing an entry from the **Range** entry box.
- A name may be assigned to each channel by editing the **Label** box for that channel.
- Volts or milliVolts for each channel may be selected within the **Units** column.

Configure Scan and Trigger

Next, move to the **Scan Count** selection box and enter values for Pre-trigger and/or Post-trigger, as desired. The timebase for the acquisition can be set to Frequency or Period within the **Timebase** selection box. The desired trigger source and parameters are selected in the **Trigger** selection box.

Collect Data

Now you are ready to read data from the WaveBook/512. Select the Scope option from the Window menu. To start performing acquisitions, click on the One Shot button, or the Continuous button, then click the Arm button. Additional channels may be viewed (up to 8) simultaneously by changing the entry in the Number of Charts menu. The next figure shows the WaveView Scope screen. Two channels are displayed in this example (up to 8 channels can be displayed at a time).



Store Data

Collected data may be stored to disk by clicking on the Disk button (first icon on left).

Driver Options (for programmers only)

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.

WaveBook applications can be written to either the Standard WaveBook API or to the Enhanced Daq* API. The Standard API has the same format (only written to 32-bit mode) as the Windows 3.1 version of the driver. Standard API functions have the **`wbk`**… prefix. The Enhanced API is a new format which can be generically used with the WaveBook, DaqBook, DaqBoard, Daq PC-Card and TempBook product lines. Enhanced API functions share the **`daq`**… prefix. Generally,

- If starting with an existing WaveBook 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 (wbk)

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

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

Enhanced API (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++ Borland C++ (v4.0 and greater)
BASIC Microsoft Visual Basic (v4.0 and greater) QuickBASIC	BASIC Microsoft Visual Basic (v4.0 and greater)
Pascal Turbo Pascal	Delphi Borland Delphi (v2.0)

Installation, Configuration, and Calibration 2

This chapter walks you through several important steps in properly setting up your WaveBook system. Although particular systems and environments may vary widely, following these guidelines will ensure a successful startup. The generic installation/setup procedure includes:

- Hardware installation—connecting all system devices and power
- Software installation—loading system software for your requirements and PC's operating system
- Configuration—setting up various software parameters to achieve the best performance for your requirements
- Testing and calibration—verifying performance of communication among connected devices and ensuring accuracy of the data.

Inspection

The WaveBook/512 system was carefully inspected prior to shipment. When you receive your 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.

Hardware Installation

PC connection. The WaveBook/512 connects to a laptop/notebook or desktop PC through the parallel printer port or through a WBK20 or WBK21 high-speed adapter. A parallel printer may be used with the WaveBook by connecting it to the WaveBook's printer pass-through connector.

Signal connection. Analog signals are connected with standard BNC connectors. Digital signals are connected to the DB25F connector.

Power connection. The WaveBook is powered by the supplied wall-mounted adapter. The unit can also be powered from a user-supplied 10 to 30 VDC source (such as a car battery) by using a standard 5-pin DIN connector (for connector pinout, see *Power Requirements* in chapter 3).

Connecting to Desktop PCs

The WaveBook may be connected to standard desktop PCs either through the PC's standard parallel port or through the optional WBK21 high-speed parallel port.

Standard Parallel-Port Connection

Follow these steps to connect to a desktop PC through the standard parallel port:

1. Power down the PC.
2. If a parallel printer connects to the PC, disconnect the printer cable from the parallel port.
3. Plug the WaveBook's cable (CA-140) into the computer's parallel port. The parallel port connector is a 25-pin socket connector.
4. Connect the other end of the cable to the WaveBook port marked "TO COMPUTER".

High-Speed Parallel-Port (WBK21) Connection

For maximum performance the high-speed WBK21 may be used.

1. Power down the PC.
2. Configure the WBK21 according to the instructions provided in its manual. Remember to record the WBK21 address and interrupt settings for future reference.
3. Open the PC and insert the WBK21 into an available 16-bit slot. The WBK21 will not operate correctly in an 8-bit slot.
4. Replace cover on the PC.
5. Plug the WaveBook's cable (CA-140) into the WBK21 parallel port. The parallel port connector is a 25-pin socket connector.
6. Connect the other end of the cable to the WaveBook port marked "TO COMPUTER".

Connecting to Laptop PCs

The WaveBook may be connected to laptop PCs either through the PC's standard parallel port or through the optional WBK20 PCMCIA high-speed parallel port.

Standard Parallel-Port Connection

Following these steps to connect to a laptop PC through the standard parallel port:

1. Power down the PC.
2. If a parallel printer is connected to the PC, disconnect the printer cable from the parallel port.
3. Plug the WaveBook's cable (CA-140) into the computer's parallel port. The parallel port connector is a 25-pin socket connector.
4. Connect the other end of the cable to the WaveBook port marked "TO COMPUTER".

PCMCIA Card (WBK20) Connection

For maximum performance the high-speed WBK20 PCMCIA card may be used.

1. Insert the WBK20 into a free Type II PCMCIA socket.
2. Insert the provided cable into the end of the PCMCIA card.
3. Connect the other end of the cable to the WaveBook port marked "TO COMPUTER".
4. Follow the instructions provided with the WBK20 to load the required software drivers. Remember to record the WBK20 address and interrupt settings for future reference.

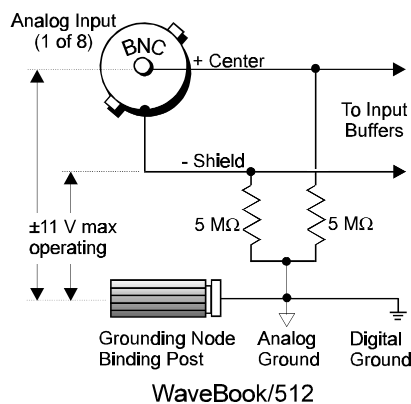
Analog-Signal Connections and Proper Grounding

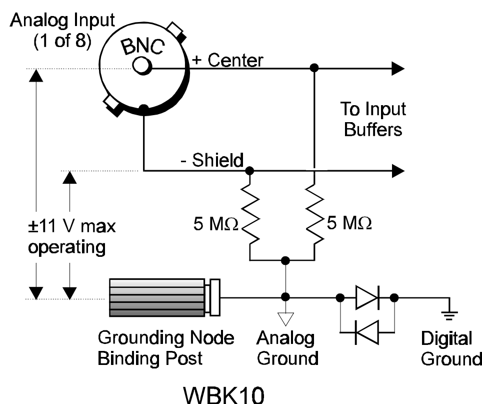
Both the WaveBook/512 and the WBK10 each have 9 analog input connections: 8 BNC connectors for analog input signals, and 1 binding post for analog common.

Each BNC analog-signal connection carries a differential signal, the measured analog voltage is the difference between the center (positive) voltage and the shield (negative) voltage. The center and shield are each connected to analog ground through a 5 MΩ resistor, giving a 10 MΩ differential input impedance. (Signals from many transducer types will need to be conditioned for voltage and impedance levels to achieve WaveBook's speed and accuracy; e.g., via WBK signal-conditioning cards and modules.)

The analog ground of the WaveBook/512 connects directly to the digital ground which connects to the computer's ground through the parallel-port cable. If the computer is a desktop computer, then this ground will be at the ground of the AC power line. If the computer is a laptop computer, its ground will be at the same potential as the WaveBook's analog ground but may have no relation to other grounds.

The analog ground of the WBK10 is not directly connected to digital ground. A pair of Shottkey diodes keeps the analog ground within a few tenths of a volt of the digital ground.





The WaveBook and WBK10 both have isolated power supplies. There is no connection between their power inputs and the analog ground.

In order for the WaveBook or WBK10 to correctly measure the input signals, each signal must be within ±11 volts of the analog signal level. Depending on your application, this may be achieved in one of several ways:

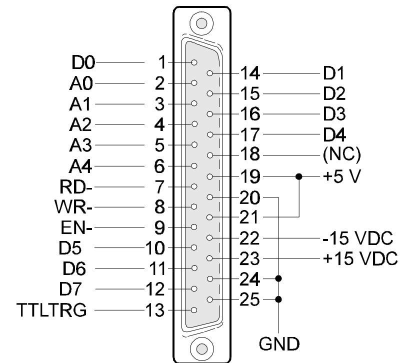
- **Common Grounds**: If your computer and signal source are grounded by the AC line, then your analog signals will already be referenced to analog ground and should be within the ±11 volt common-mode range. **Note**: laptop computers are usually not grounded by the AC line, even when plugged into their AC power adapters.
- **Floating Grounds**: If both the computer and the signal source are battery operated or otherwise isolated, then the internal 5 MΩ resistors may provide enough of a ground connection to keep the analog inputs within the common-mode range. If the computer or the analog signal source is AC line powered, then the parasitic capacitance will probably drive the signals out of the common-mode range and require a direct ground connection.
- **Direct Ground Connection**: Unless both the computer and the signal source are already connected to a common ground such as the AC power line, a direct ground connection is recommended between an appropriate reference point on the signal source and the analog ground binding post.
  - If the signal source is differential, then it may already have a reference ground that should be connected to the binding post. If it does not, then one of the differential signals should be connected to the binding post with a 100 KΩ or larger resistor.
  - If the signal source is single-ended, then connect its ground to the binding post.

When connecting several signal sources to the binding post, it is important to make sure that the grounds are all compatible. There should be no circuit paths or ground loops that would tend to force the ground connections to different potentials.

Digital I/O Connection

The following signals are present on the DB25F high-speed digital I/O connector (see pinout figure and table):

- 8 Digital I/O Lines
- 5 Address Lines
- Active-low Digital I/O Enable output
- Active-low Digital I/O Read and Write Strobes
- TTL Trigger Input
- +15 V, -15 V power, 50 mA max. (each supply)
- 2 +5 V power, 250 mA max. (total)
- 3 Digital Grounds



Digital I/O Connector Signals	
D7 - 0	Digital I/O data lines
A4 - 0	Digital I/O address lines
EN-	Active-low digital I/O enable
RD-	Active-low read strobe
WR-	Active-low write strobe
TTLTRIG	TTL trigger input
+5 VDC	250 mA maximum
+15,-15 VDC	50 mA maximum

To sample just 8 digital input signals, connect them directly to the digital I/O data lines. D7 is the most significant bit, and D0 is the least. The address lines, the read and write strobes, and enable signal may all be left disconnected.

To use the digital I/O address lines to select from up to 32 input bytes or to use the digital I/O port for output, see chapter 4.

Optional Printer Connection

WaveBook/512 allows for LPT pass-through for printer operation while the WaveBook is connected. When using a printer in the system configuration, attach the original printer cable (plug DB25) into the connector marked "TO PRINTER" on the WaveBook/512.

WaveBook Software Installation

The WaveBook/512 software installation consists of two steps: making a backup copy and installing the user-selected WaveBook/512 files. The software can be installed under Windows 3.1 or Windows 95/NT. A description of each method follows. **Note**: any previous installation of the WaveBook/512 drivers should be removed before installing a new version.

Making a Backup Copy

Make a backup copy of the release disks as follows:

1. Boot up the system according to the manufacturer's instructions.
2. Enter the DOS operating system (if in another operating system, follow diskcopy procedures).
3. Type the command CD\ to go back to your system's root directory.
4. Place driver disk #1 into drive A (or whatever floppy disk drive you are installing from).
5. Type DISKCOPY A: A: and follow the instructions given by the DISKCOPY command.
6. This process may require swapping the release and backup disks several times.
7. When the disk copy is complete, repeat the process for all disks in the installation set.

Adding WaveBook Reset to AUTOEXEC.BAT

If the host PC is reset during an acquisition, the WaveBook may not be automatically reset along with it. The WaveBook (and any attached printer) may then not operate correctly. To assure that the WaveBook is reset correctly, the WaveBook reset program (**`wbkReset`**), supplied with the WaveBook, should be executed as part of the AUTOEXEC.BAT file. Using some type of text editor, such as Windows "Notepad", add the following line to the AUTOEXEC.BAT file:

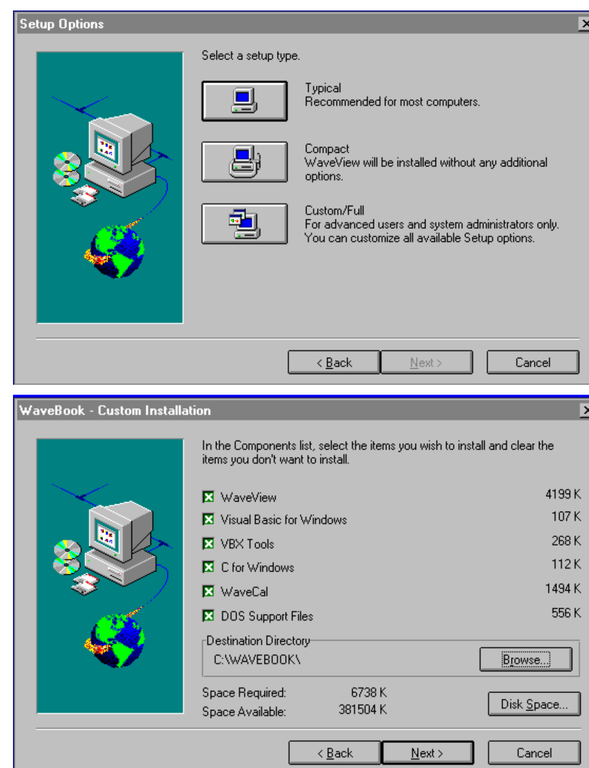
```
c:\wavebook\wbkreset
```

Note: The path may be different if the software was installed into a directory other than **`\wavebook`**.

Installation Under Windows 3.1

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

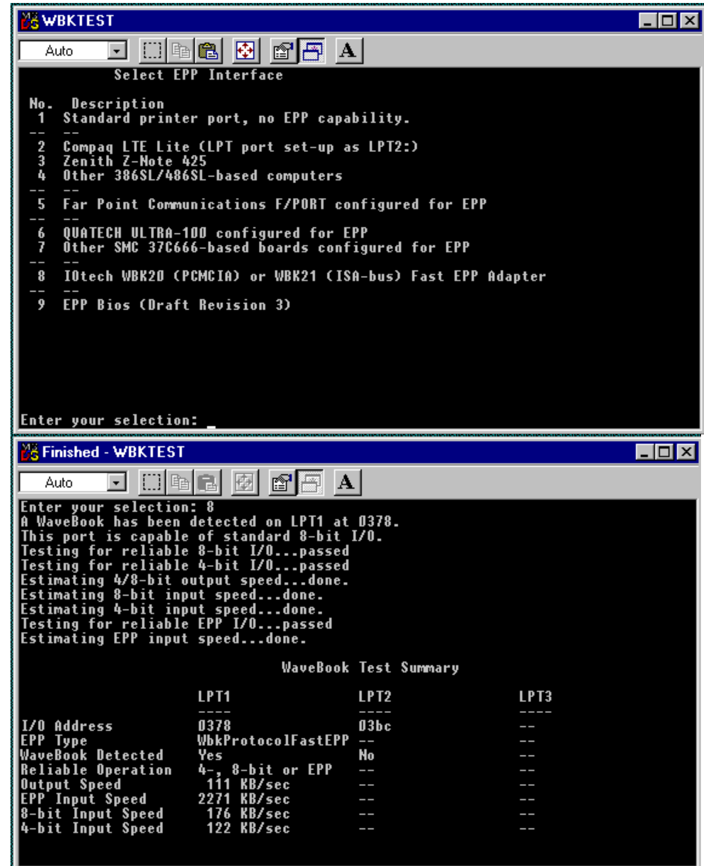
1. After exiting other programs, begin installation by putting disk #1 of the Windows 3.1 set into your floppy disk drive. From the program manager choose File/Run to run the **SETUP.EXE** file.
2. The Windows Setup dialog box asks you for a destination directory.
3. From the next dialog box, select 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. A final message will display when installation is complete.



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

Using WBKTest

In systems with Windows 3.1, WBKTest is a utility program that tests the communication channel from the computer to the WaveBook/512. WBKTest is easy to use. RUN the program, select your test options, and then view the results. Besides the opening screen, 2 other screens include the test selection screen and the test summary screen (see following figures). You should select the interface test that best matches your system configuration. Option 1 should work with printer ports from all computers.



Installation Under Windows 95 and Windows NT

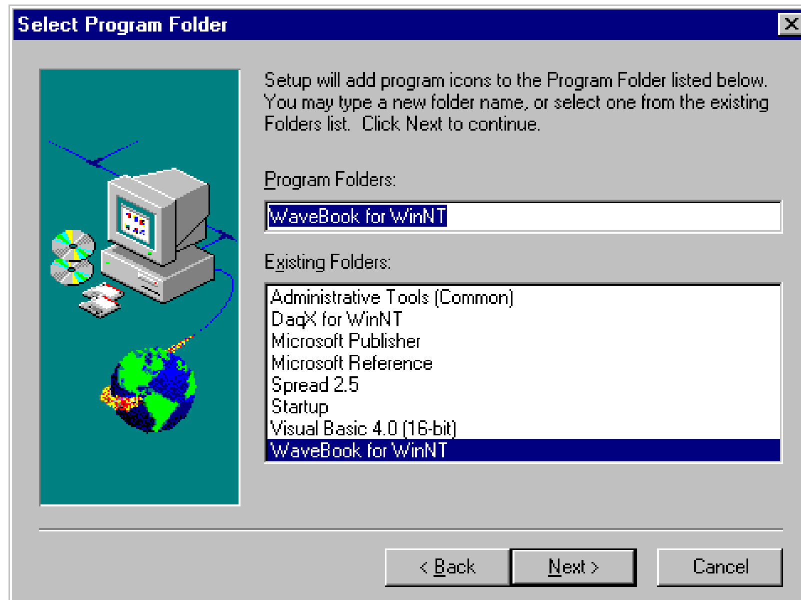
This section describes the installation of WaveBook software under the Windows 95 and Windows NT operating systems. Installation for both Windows 95 and Windows NT are operationally equivalent. In fact, they use the same installation disk set. The installation program automatically detects which operating system is running and then proceeds accordingly. **Note**: The following figures show screens for a Windows 95 installation; however, identical procedures apply for a Windows NT installation.

Two preliminary steps are to:

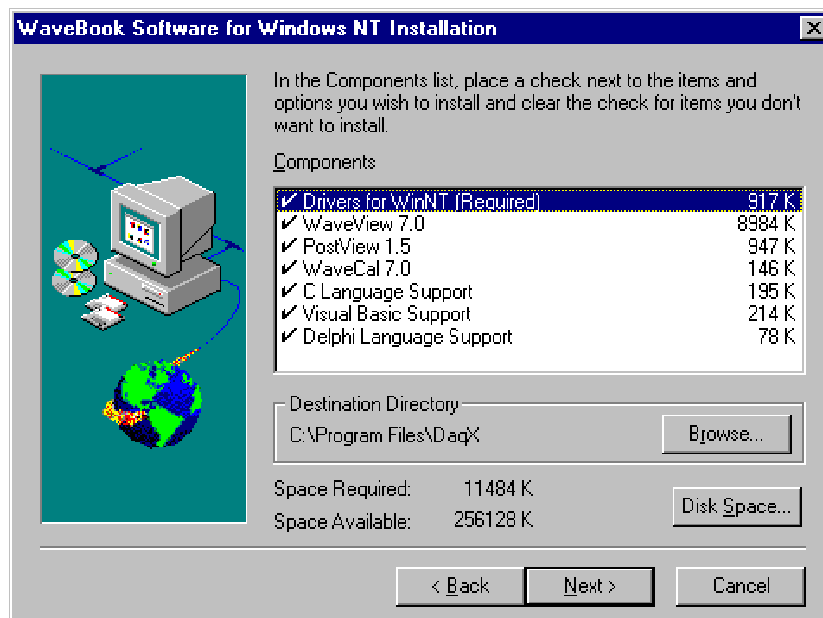
- Locate the disk set labeled *WaveBook Software for Windows95/NT* and make a backup copy of the disk set before proceeding.
- (If the WaveBook is to be used immediately) Attach the WaveBook to the desired LPT port and power-on the system before starting the software installation.

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 the WaveBook installation when ready.

The next screen (see next figure) allows you to specify the program folder in which to install the software. A list of current program folders can be used or you can specify a new one. Click Next> to continue with the installation. Click Cancel to exit without installing the software.



The next screen displays component installation options. All selected items (indicated by the check-mark) are installed. By default, all options are selected. To de-select an item, click the corresponding check box for that item. To select a de-selected item, click the corresponding check box for the item. The bottom of screen displays the amount of hard disk space required for installation and the amount of disk space remaining after installation. Make sure adequate memory is available before continuing. Click Next> to continue with the installation. Click Cancel to exit without installing the software.

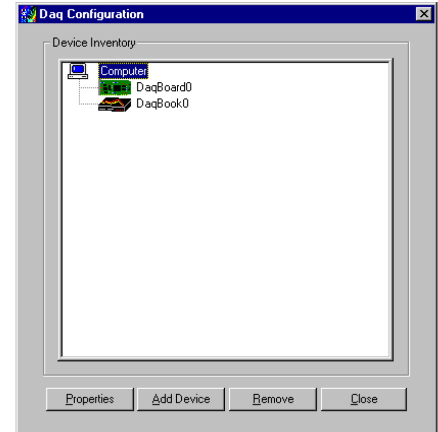


The next screen shows the progress of the installation of the software components. As the installation progresses, you may be asked to periodically insert the next disk into the floppy drive. When prompted, replace the current disk with the next disk. When the software component installation is complete,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

- Exit and perform device configuration (If the WaveBook 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.

WaveBook Configuration Under Windows 95/NT

This section describes the configuration of WaveBook 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:

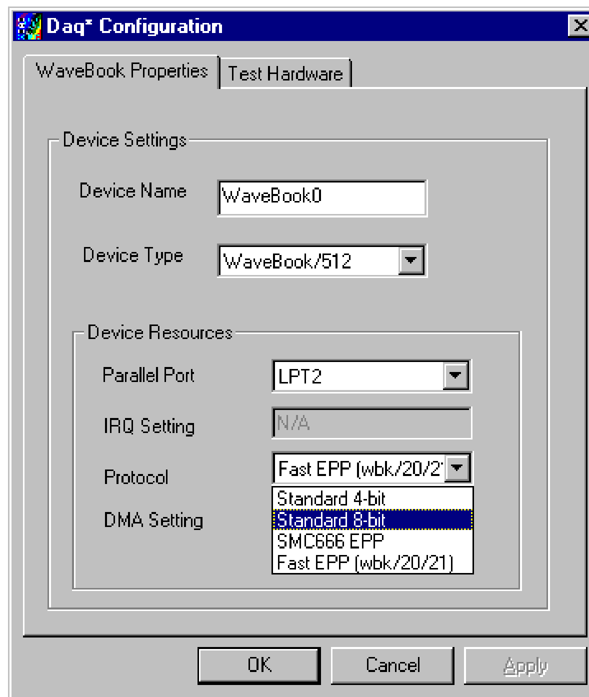
- **Properties**. Current configuration settings for a device can be changed by bringing up the corresponding properties screen. To do so, double-click the device icon or single-click the device and then double-click the Properties button.
- **Remove**. The Remove button is used to remove a device from 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 the following window (Note: screen shows WaveBook/512 currently selected):

Use the scroll bar to find the WaveBook 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 WaveBook 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. This name will also be displayed in the device lists for opening the device in the WaveView, WaveCal, and GageCal applications.
- 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 WaveBook. Depending on your system, not all protocols may be available.

Note: IRQ Setting and **DMA Setting** for the WaveBook 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 WaveBook, make sure the device has been properly installed and powered-on. Make sure the parallel port cable is firmly in place on both the WaveBook and the proper LPT port in the computer. **Note:** Testing the WaveBook 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 WaveBook configuration settings to those that work properly.

To test the current stored configuration for the WaveBook 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

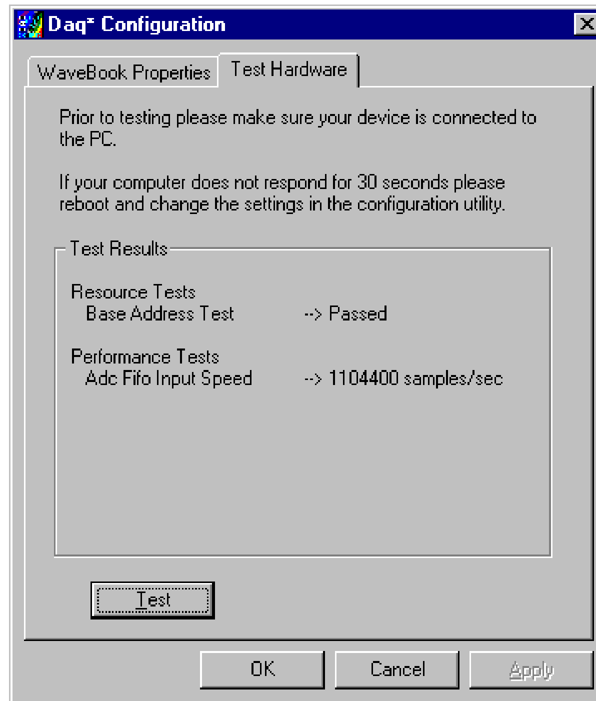
The resource tests are intended to test system capability for the current device configuration. These tests are pass/fail. Resource test failure may indicate a lack of availability of the resource or a possible resource conflict.

- ***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 WaveBook 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 WaveBook'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 WaveBook or if trying to connect causes the system to hang or crash, then you should:

- Check that WaveBook's power LED is ON. If not ON, verify power connection between the WaveBook and the power source.
- Make sure the LPT cable is firmly attached to the computer's proper LPT port and to the WaveBook 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 WAVEBK`. To unload the driver manually, use the following sequence: `NET STOP WAVEBK`.

Calibration

Calibration of the WaveBook/512 may be performed 2 different ways depending on the application.

- The first method uses factory-generated calibration constants—no manual calibration by the user is required. The stored constants are used by the WaveView program when the user selects the Factory Calibration Table option from the System Menu. The calibration constants are stored onboard the WaveBook/512, WBK10 Expansion Chassis, and WBK11 Simultaneous Sample and Hold card. When the components are interconnected, the software reads these constants and uses them to perform a calibration of the components working together as a system. This calibration method is ideal for applications that require the hardware to be continually reconfigured.
- The second method produces a slightly better calibration of the system as a whole but requires manual calibration. The procedure produces a calibration of the complete signal path from the input to the A/D converters. The results of the calibration procedure are stored in the User Calibration Table option of the WaveView System Menu. The user selects this option to use the manually derived calibration parameters. Recalibration is necessary whenever anything in the signal path is changed, such as adding a WBK11 to a WaveBook/512. For this reason, this calibration method may not be best suited to applications where the hardware configuration changes frequently.

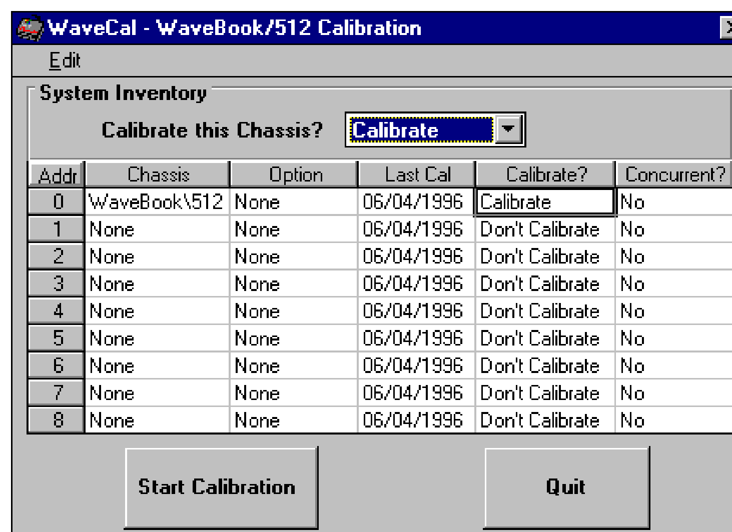
Manual Calibration

The WaveBook/512 supplies internal 0.000 V and 5.000 V (and 0.5000 V with WBK11) for calibration. You can calibrate all of the offsets and some of the gains using either the internal voltage references or user-supplied external voltage references. A user-calibration program (WaveCal) is provided to allow calibration using just the internal references, a DMM, and the factory-measured gain ratios, or a calibrator capable of supplying all of the required voltages.

Calibration is carried out by the host computer which sets up the calibration, analyzes the measurements, determines the correction factors, downloads them to the WaveBook/512 for use during acquisition, and optionally saves them in the WaveBook/512 EEPROMs. Calibration occurs only when commanded by host computer software. For best calibration, the software is able to adjust the calibration constants so that a user-supplied reference within a limited calibration range is measured accurately.

WaveCal

WaveCal is a Windows-based application for performing manual calibration of the WaveBook/512 system. The program is transferred to the user's PC when the WaveBook/512 installation software is loaded. A WaveCal icon is generated in the WaveBook/512 program group. Click on the icon to start the application (see figure of WaveCal main screen). WaveCal uses a 2-point linear approximation method to calculate calibration Gain and Offset Errors for a given gain and polarity of each channel.



Using WaveCal

Upon loading, WaveCal performs a system inventory of all equipment connected to the WaveBook/512 main chassis. For each chassis, the system inventory includes any option cards installed and the date that the chassis was last calibrated. The chassis, option cards and last known calibration date are displayed in spreadsheet form along with two other columns: "Calibrate?" and "Concurrent?". Each spreadsheet column is described below.

Chassis

This column displays the type of chassis found at the specified address. Currently, the valid entries for this column are WaveBook/512, WBK10 or None. This column is part of the general system inventory and cannot be changed by the user.

Option

This column displays the type of card (if any) installed on the corresponding chassis. Currently, valid values for this column include: WBK11, WBK12, WBK13, WBK14, WBK15, WBK61, WBK62, and None. This column is part of the general system inventory and cannot be changed by the user.

Last Cal

This column displays the date when the system was last calibrated. This date corresponds to the last date that calibration constants were written to the chassis' internal EEPROM. Therefore executing the WaveCal program without writing the changes will not change the value of this column for a particular chassis. This column is part of the general system inventory and cannot be changed by the user.

Calibrate?

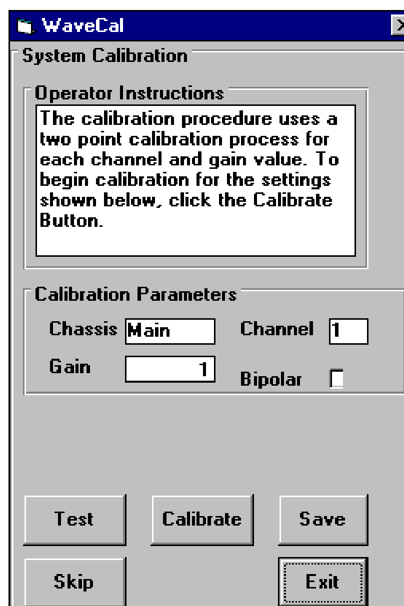
This column indicates whether the corresponding chassis should be calibrated. This column is user-setable by either double-clicking the entry of the desired chassis or by clicking the column header to select the entire column and then selecting "Calibrate" or "Don't Calibrate" under the "Calibrate This Chassis?" list box. Initially, all chassis are set to "Don't Calibrate". Only chassis which are present in the chassis (whose value is other than None) column may be changed to "Calibrate". When "Calibrate" is selected for a particular chassis, this indicates to WaveCal that the user wishes to calibrate at least one channel on that particular chassis.

Concurrent?

This column indicates whether all channels on the corresponding chassis should be calibrated concurrently. Normally, the two-point calibration process will calibrate each channel on a particular chassis individually. However, WaveCal has the ability to apply the same two-point calibration to all channels on the specified chassis concurrently. Concurrent calibration can be used to greatly reduce the calibration time for a particular system; however, concurrent calibration should not be used when the accuracy for a particular channel within the chassis is questionable subsequent to calibration. Concurrent calibration may be toggled for a particular chassis by double-clicking the entry corresponding to that chassis.

After WaveCal has taken the system inventory, check to make sure that all chassis and option cards connected to the system are properly displayed. You may then select which chassis to calibrate and whether their channels are to be individually or concurrently calibrated.

Performing System Calibration



When satisfied with the system inventory and calibration configuration, the calibration process can be initiated by clicking the Start Calibration button. A new screen, "System Calibration", will then be loaded (see figure). This screen controls the calibration process for each Chassis, Channel, Gain and Bipolar setting in the system. The fields under Calibration Parameters indicate the current setting for each of these parameters. Each of these is defined in detail below.

Chassis

Indicates which chassis is currently scheduled to be calibrated. Initially, the chassis will be set to "Main" to indicate that the first chassis to be calibrated is the WaveBook/512 main chassis.

Channel

Indicates which channel is currently scheduled to be calibrated. The value of this field will correspond to the current channel number regardless of the current chassis unless Concurrent calibration was selected for the current chassis. In this case, the value of field will be "All" indicating that all channels on the current chassis will be calibrated concurrently.

Gain

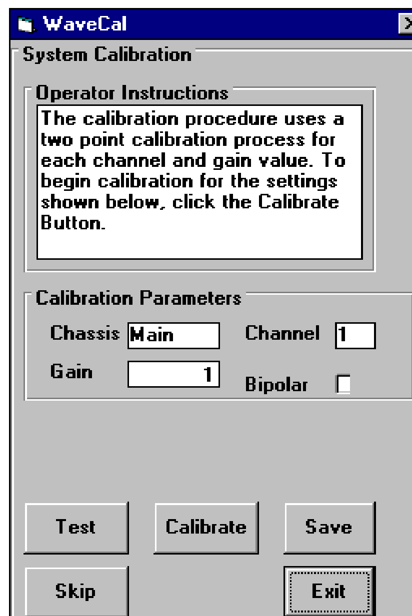
Indicates the current gain value scheduled to be calibrated. The value of this field initially starts at 1 for each channel being calibrated and will go up the maximum gain value for the current chassis. (10 for the WaveBook/512, 100 for the WBK10).

Bipolar

Indicates the current polarity scheduled to be calibrated. Both bipolar and unipolar polarity are calibrated for each gain value. This value initially is set to unipolar for each gain and will subsequently toggle to bipolar for the current gain value.

WaveCal automatically updates the current calibration parameters depending on their current settings and user events. The calibration procedure is controlled by the following command buttons:

Calibrate



This button initiates the 2-point calibration procedure for the current calibration settings displayed under calibration parameters. This may be performed as many times as the user desires for the current calibration parameters indicated. When the Calibrate button is clicked, the Two-Point Calibration screen will be displayed (see figure).

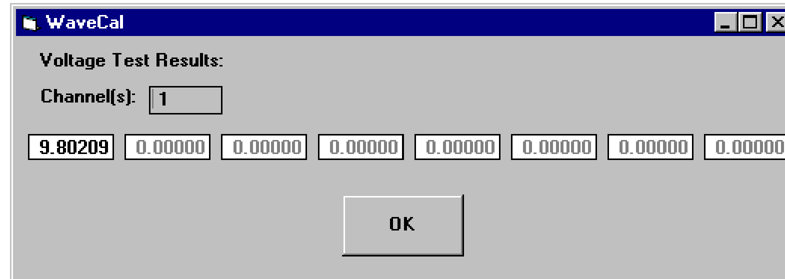
The calibration will be performed on the calibration parameters (Chassis, Channel, Gain and Bipolar) shown on the previous screen. The user will be prompted to apply a voltage source to the appropriate channel at the level indicated. The user may optionally change the voltage level slightly if it better suits the calibrator. After the appropriate voltage level has been applied, the Calibrate button may then be clicked to initiate the calibration of the first point. The user will then be prompted to apply a new voltage level to the appropriate channel for calibration of the second point. Again, the voltage level may be changed slightly if the indicated voltage is not ideal for the calibrator. The calibration of the second point is then again initiated by clicking the Calibrate button. The 2-point calibration for the current Calibration Parameters is now complete.

Save

This button saves the calibration constants for the current calibration parameters indicated. Before executing this command, the 2-point calibration for these settings must have first been performed. The save action will also automatically update the calibration parameters to reflect the next chassis, channel, gain and bipolar parameters scheduled to be calibrated. Once this command has been executed, however, you may not return to the previous calibration parameter settings. So, be sure that you are satisfied with the calibration before executing the Save command.

Test

This button may be used to test channel voltage levels both before and after calibration. This command will display the current voltage level for the current channel(s). If concurrent calibration has been selected for the chassis, then the voltage levels for all 8 channels for the current chassis will be displayed. Otherwise, only the voltage level for the current channel will be displayed. A sample test is shown below:



Skip

This button may be used to skip the current calibration parameters indicated and move the next scheduled chassis, channel, gain and bipolar setting. This button may be used if it is not desirable to change the current calibration for the settings indicated.

Exit

This button is used to exit the Chassis Calibration screen. This may be used following the successful calibration of the entire system or to exit the current calibration process.

Notes

WaveBook Expansion Options 3

This chapter describes the option cards and modules that can be used with the WaveBook/512. WaveBook expansion options are listed in the following table. Additional items, not reflected by the table, may be available by the time this document goes to print. Note that some expansion options, such as WBK20 and WBK21, are shipped with additional documentation.

Option Cards and Modules Used with the WaveBook			
Product	**Name/Description**	**Capacity**	**Page**
DBK30A	Rechargeable Battery/Excitation Module	12-14, 24-28 VDC 3.4 A-hr @ 14 V	3-2
DBK34	Vehicle UPS Module (Uninterruptable Power Supply)	12/24 VDC 5.0 A-hr @ 12 V	3-6
WBK10	Expansion Module	8 channels	3-8
WBK11	Simultaneous Sample & Hold Card	8 channels	3-11
WBK12	Programmable Low-Pass Filter Card	8 channels	3-13
WBK13	Programmable Low-Pass Filter Card With SSH	8 channels	3-13
WBK14	Dynamic Signal Conditioning Module	8 channels	3-16
WBK15	8-Slot 5B Signal Conditioning Module	8 channels	3-23
WBK16	Strain-Gage Module	8 channels	coming soon
WBK20	PCMCIA/EPP Interface Card	>2 Mbytes/s	3-28
WBK21	ISA/EPP Interface Card	>2.5 Mbytes/s	3-29
WBK61	High-Voltage Adapter with 200:1 Voltage Divider	1 channel	3-31
WBK62	High-Voltage Adapter with 20:1 Voltage Divider	1 channel	3-31

The WaveBook/512 is designed for easy expansion with the WBK family of option cards and modules. Internally, the WaveBook/512 has room for 1 signal conditioning card. For expansion, you can use 1 or more expansion modules (WBK10, WBK14, and/or WBK15).

In expanding systems, be sure to provide adequate power. Various configurations are possible including the use of a DBK30A Battery Module (also used in the Daq* family of products). Power supply options and setups are discussed in the next section and the DBK30A and WBK10 sections.

Power Management

Some WaveBook products use more power than others; it is important to compute your system's requirement so you can provide adequate and reliable power. Computing the power use is important when using batteries so you can determine a safe runtime before recharging. **Note**: using the AC adapter supplied with each module supplies sufficient power—you need not make these calculations unless you are daisy-chaining units or where battery runtime is critical.

To estimate your system's total power requirement, add up the amperage for all units in your system (see Worksheet table on next page). The table at right gives the amperage requirements for the WaveBook family of products using the DBK30A at both voltage settings and using the TR-40U. **Note**: higher voltages draw fewer Amps for the same power (current drawn with other sources such as a car battery can be estimated from this table).

Current Requirements (in Amps) of WaveBook Products			
Products and Product Combinations	**DBK30A 14 VDC**	**DBK30A 29 VDC**	**TR-40U 15 VDC**
WaveBook/512 (alone)	0.43	0.20	0.40
WBK10 (alone)	0.32	0.20	0.30
WBK11	0.27	0.10	0.22
WBK12	0.47	0.23	0.45
WBK13	0.57	0.28	0.20
WBK14 (alone)	0.90	0.50	0.85
WBK15 (alone)	0.13	0.08	0.12
WBK15 (typical)*	0.24	0.13	0.23
WBK15 (max)**	0.75	0.36	0.70

*Typical with 8 voltage modules.
** Maximum load with 8 strain-gage modules.
You may need to consult power specifications for individual 5B modules and for any excitation currents required.

You can compute your power requirements by filling in the following table and performing the indicated operations. Do not overload your power supply; with heavily-loaded systems (over 2.5 A when using the TR-40U), you may need to use more than one power source.

Worksheet for Power Requirements					
Unit	Qty		Amps		Totals
WaveBook/512		×		=	
WBK10		×		=	
WBK11		×		=	
WBK12		×		=	
WBK13		×		=	
WBK14		×		=	
WBK15		×		=	
			Maximum Amps		

Input voltage to the system modules (WaveBook/512, WBK10, WBK14, WBK15) must be 10 to 30 VDC and can come from an AC adapter or from a battery. System cards (WBK11,12,13) get power from their expansion module.

Available AC adapters include the TR-40U (supplied) and the TR-27 (optional).

- TR-40U has an input of 90-264 VAC and an output of 2.50 A @ 15 VDC.
- TR-27 has in input of 120 VAC (19.5 W) and an output of 800 mA @ 18 VDC.

Battery options include the DBK30A (see chapter 3) or other 10 to 30 VDC source such as a car battery. The DBK30A provides 14 VDC and when fully-charged has a storage capacity of 3.4 A·hr. (Car batteries have much higher capacities.) The basic formula for battery life is:

Runtime (hr) = Battery capacity (A·hr) / Current load (A)

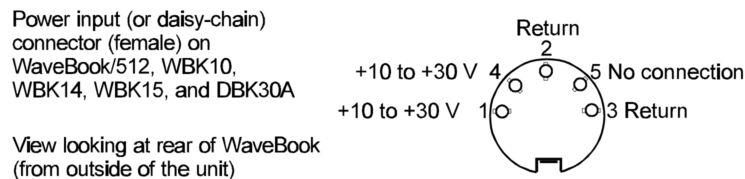
Note: Battery life and performance depend on various factors including battery type/condition, charge level, and ambient temperature—you may want to allow a corresponding tolerance factor where runtime is a critical factor.

Connection: multiple units running on a common power supply can be daisy-chained together with CA-115 power cables (the DC current source must be sufficient for all daisy-chained units). The optional CA-116 cable permits the system to be plugged into a car lighter socket.

Power Connector Pinout

Some users may wish to make their own cable for powering the WaveBook/512 and WBK option modules from a custom supply. The following diagram shows the pinout for the DIN5 power connector:

Note: CA-115 and 5-pin DIN are rated at 5 Amps maximum load.

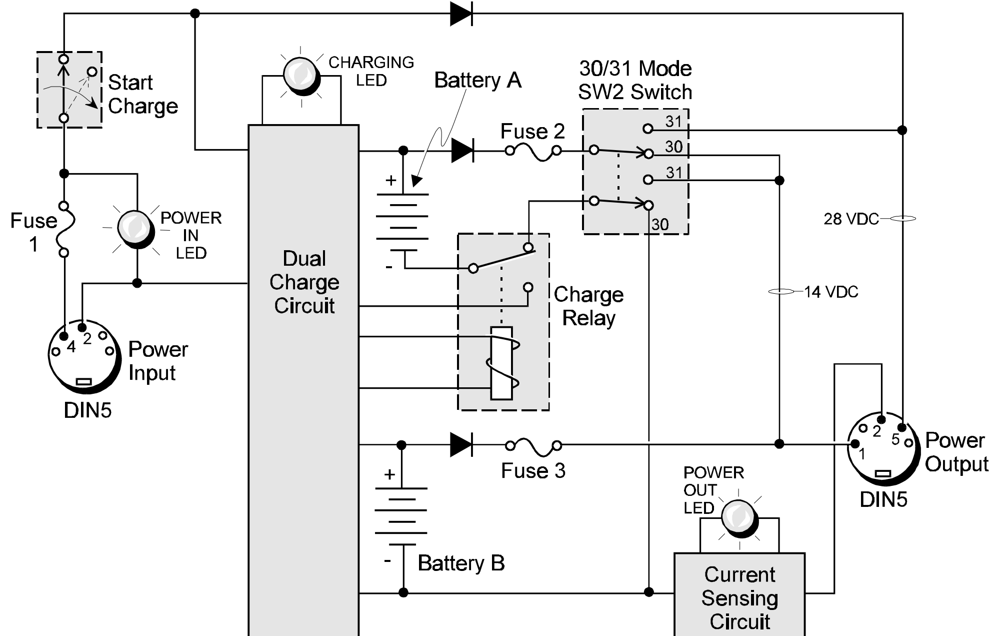


DIN5 Power Pinout

DBK30A Rechargeable Battery Module

Description

The DBK30A contains a rechargeable nickel-cadmium battery that can power portable applications of a WaveBook/512, expansion WBK modules, and transducers that require excitation. The unit's rugged metal package is the same modular size as WaveBook products for convenient stacking with the included fastener panels and Velcro tabs. **Note**: In some cases, it may be necessary to compute the power use by all connected equipment. Refer to power specifications for each component, and then verify that your power source is sufficient for your runtime requirements.



DBK30A Block Diagram

Power input comes from the included AC adapter that converts AC mains power into 24 VDC to charge the unit's 2 battery packs. Automatic charging circuits recharge the internal batteries quickly and safely when connected to the supplied AC adapters. For trouble-free operation, you must fully charge the batteries before use. The charged battery runtime will depend, of course, on the current load and mode of operation.

The 2 modes of operation are the DBK30 and DBK31 modes (the DBK30A replaces these models). An internal slide switch SW2 determines whether the DBK30A will act as a DBK30 (default) or DBK31.

- **The DBK30 mode provides 14 VDC** for 3.4 A-hr. The typical battery runtime is from 3 to 6 hours depending on the load. **Note**: This is the default mode and should normally be used for the WaveBook/512 and WBK modules unless 28 VDC is also required.
- **The DBK31 mode provides both 14 VDC and 28 VDC**. 14 VDC is used for unregulated bridge excitation for bridge-configured sensors (such as load cells) and power to WBK expansion products. 28 VDC is used for loop currents for 2-wire 4-20 mA transmitters (1.7 A-hr). The battery run-time ranges from 1 to 6 hours, depending on system configuration.

Note: Unless you need 28 V, leave the DBK30A in the default DBK30 mode. In the DBK31 mode, only 1 of the 2 battery packs provides 14 VDC (instead of both packs in parallel in the DBK30 mode); thus, runtime is reduced in the DBK31 mode.

Hardware Setup

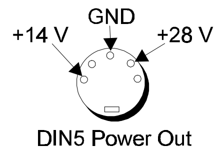
Configuration

The only configuration option is the choice of modes as discussed above. If you do not need to use 28 V, leave SW2 in the default position (for previously used units, you may need to verify the proper position). The SW2 slide switch is located inside the module on the printed circuit board near the front center of the unit. To configure the DBK30A's mode:

1. Remove the top cover by unscrewing one screw and sliding the cover forward until it separates from the module.
2. Locate SW2 and position it for the desired mode. Slide SW2 to the right for DBK30 operation providing 14 VDC only. Slide SW2 to the left for DBK31 operation providing both 14 and 28 VDC.
3. Replace top cover and secure with screw.

Connection

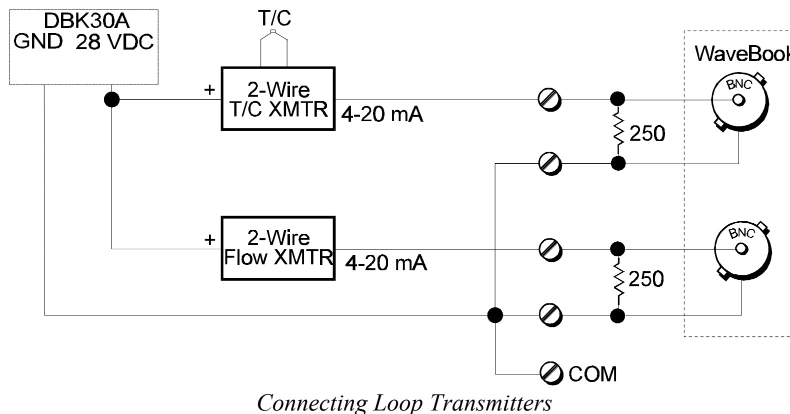
The figure shows the pinout for the POWER OUT DIN5 connector. The 28 V pin is only active in the DBK31 mode. The 14 V pin is always active.



The DBK30A package includes a short connecting cable (CA-115) to connect to the WaveBook/512 (and possibly daisy-chained to a WBK expansion module). This cable connects the POWER OUT connector on the DBK30A to the POWER IN connector on the WaveBook.

Connections for the DBK31 Mode

The primary purpose of the DBK31 mode is to power external user-supplied loop transmitters. The hookup is simple, as shown below.

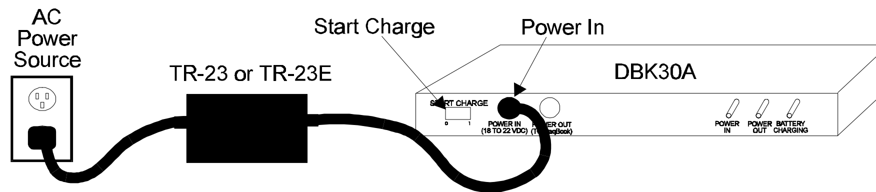


Connecting Loop Transmitters

Another use for the DBK31 mode is providing an excitation source for bridge-type sensors such as load cells (strain gages) and other devices that may be attached to 5B modules inside a WBK15. The excitation voltage is not regulated by the DBK30A; so, this voltage must be externally regulated to 10.00 V for most load cells.

Charging the Battery Module

The DBK30A package includes a charger for the intended line voltage (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 AC power. The charge cycle begins automatically whenever AC power is applied after an interruption. The charge cycle ends when the batteries are fully charged.



Connecting the Charger

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

Three LEDs on the DBK30A provide status information on the charging process or the external load.

LED	Meaning
POWER IN	Illuminates when the charger is connected to a source of AC power and to the battery module.
BATTERY CHARGING	Illuminates steadily while battery is in the high current (2 A) charge mode. Flashes briefly, one or two flashes at a time, when the internal batteries are fully charged.
POWER OUT	Illuminates steadily when an external WaveBook product is connected and drawing current from the battery modules.

CAUTION

Periodically, fully discharge the DBK30A to inhibit lazy chemistry (memory) in the cells. To manually discharge a battery pack, connect a WaveBook to the pack and leave it powered-on until the indicator lights go dark.

Using the Battery Module While Charging

Both operating modes are capable of powering the WaveBook products while being charged; however, the charging current is reduced, and charging time is increased. If AC power is interrupted, a new charge cycle will begin automatically when AC power returns.

CAUTION

Even with the AC adapter, the batteries will eventually discharge under a WaveBook operating load. Charging DOES NOT BEGIN AUTOMATICALLY (except on power-up). You must manually initiate the next charge cycle. Do not expect a WaveBook powered by a DBK30A to operate as an uninterruptable power supply.

DBK30A - Specifications

Name/Function: Rechargeable Battery Module
Battery Type: Nickel-cadmium
Number of Battery Packs: 2
Battery Pack Configuration: 12 series-connected sub-C cells
Output Voltage: 14.4 V or 28.8 V (depending on the selected mode)
Output Fuses: 2 A
Battery Amp-Hours: 3.4 A-hr (1.7 A-hr/pack)
Charge Termination: Peak detection
Charge Time: 2 hours
Charging Voltage from Supplied AC Adapter: 22 to 26 VDC @ 2 A
AC Adapter Input: 95 to 265 VAC @ 47 to 63 Hz
Size: 221 mm × 285 mm × 35 mm (11" × 8-1/2" × 1-3/8")
Weight: 2.4 kg (6 lb)

DBK34 Vehicle UPS Module

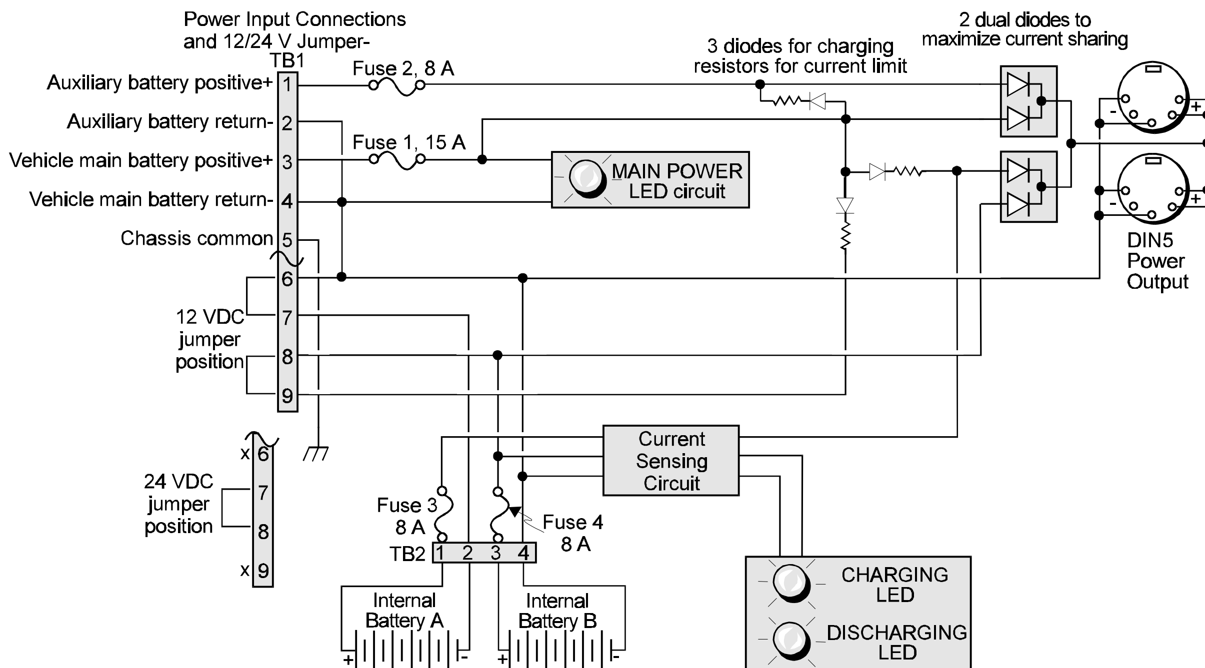
Overview

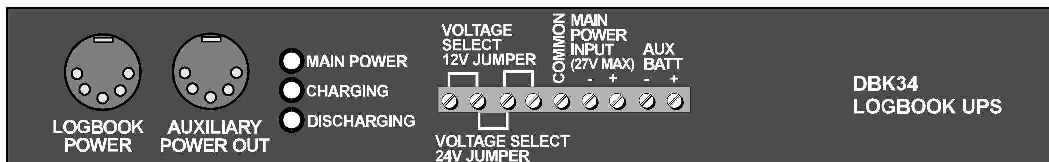
The DBK34 can power a data acquisition system in portable and in-vehicle applications (both 12 and 24 V systems). Power storage capacity is 5 A-hr @ 12 VDC or 2.5 A-hr @ 24 VDC. For reliable data acquisition in a vehicle, the DBK34 provides clean and consistent operating power:

- Prior to engine/generator start
- During engine start-up (battery sag due to the high-current demand of starter motor and solenoid)
- After engine turn off.

The DBK34 contains 2 sealed-lead rechargeable batteries and associated charging circuits and current indicators. Typically, these batteries can last more than 300 full cycles and up to 10 years standby lifetime at room temperature. Recharging is fast, and extreme temperature performance is good. The DBK34 can be used with the LogBook, DaqBook, WaveBook, and related DBKs and WBKs. The unit's rugged metal package has a compatible 8×11" footprint for convenient stacking with Velcro tabs and optional fastener panels and handles for carrying.

Main and auxiliary power input comes from 12 or 24 VDC via a terminal block on the unit's front panel (12/24 V modes are set by front-panel jumpers). Automatic charging circuits recharge the internal batteries quickly and safely. For trouble-free operation, you must fully charge the batteries before use. The charged battery runtime will depend on the load and mode of operation. **Note**: See *Power Management* at the beginning of this chapter for details how to compute power requirements.





DBK34 Block Diagram and Front Panel

Note: TB1 pin numbers read from right to left as viewed from the front panel.

Hardware Setup

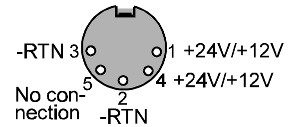
Configuration

The 12 or 24 V configurations are selected as follows:

- for 12 V operation, set 2 jumpers on terminals 6-7 and 8-9 of TB1
- for 24 V operation, set 1 jumper on terminals 6-7 of TB1

Connection

Power In - (vehicle main/auxiliary batteries, 12 or 24 VDC only) Connect main battery positive to terminal 3 of TB1 and main negative to terminal 4. If an auxiliary battery is used, connect its positive to terminal 1 and negative to terminal 2.



DBK34's DIN5 power output connectors (2 sockets)

Power Out - The figure shows the pinout for the POWER OUT DIN5 connectors. The DBK34 package includes a short connecting cable (CA-115) to connect to the powered device. This cable connects the POWER OUT connector on the DBK34 to the POWER IN connector on the LogBook, DaqBook, or DBK module.

DBK34 Operation

Indicators: 3 LEDs on the DBK34 provide status information on the power and charging process.

LED	Meaning
MAIN POWER	Lights when the DBK34 is connected to a live vehicle (main) battery.
CHARGING	Lights when internal batteries are being charged at a rate of 0.025 to 0.050 A or greater.
DISCHARGING	Lights when internal batteries are discharging at a rate of 0.025 to 0.050 A or greater.

Runtime: Approximate runtime under various loads can be computed from the storage capacity (5 A-hr in 12 V mode; 2.5 A-hr in 24 V mode) and the load (main unit and other DBKs). See section *Power Management* at the beginning of this chapter. Factory testing determined the following run-times:

Run-time	Load Conditions
240 minutes	DBK34 with 0 watt external load @ 23°C
190 minutes	DBK34 with 2.34 watt external load @ 23°C
120 minutes	DBK34 with 6.79 watt external load @ 23°C

Charging: In general, lead-acid batteries require charging at 120% of drain energy (e.g., the 5 A-hr DBK34 requires a charge equal to or greater than 6 A-hr). Charging times vary; but 4 to 5 hours at 14 V is typical for a totally empty battery.

CAUTION

Voltage applied to charge a DBK34 must not exceed 15 VDC in 12 V mode or 30 VDC in 24 V mode. If not charging from the vehicle, a generic automobile battery charger (3 A) in 12 V mode is recommended.

Environmental Concerns

CAUTION

The DBK34 s batteries contain toxic materials (Pb and H_2SO_4). After the battery s life cycle is over (up to 300 full cycles or 5-10 years of use), sealed-lead batteries must be recycled or properly disposed of.

DBK34 - Specifications

Name/Function: Vehicle UPS Module
Battery Type: Sealed-lead rechargeable
Number of Battery Packs: 2
Battery Pack Configuration: 6 series-connected D cells
Output Voltage: 12 V or 24 V (depending on jumpers)
Output Fuses: 8 A on each internal battery (2)

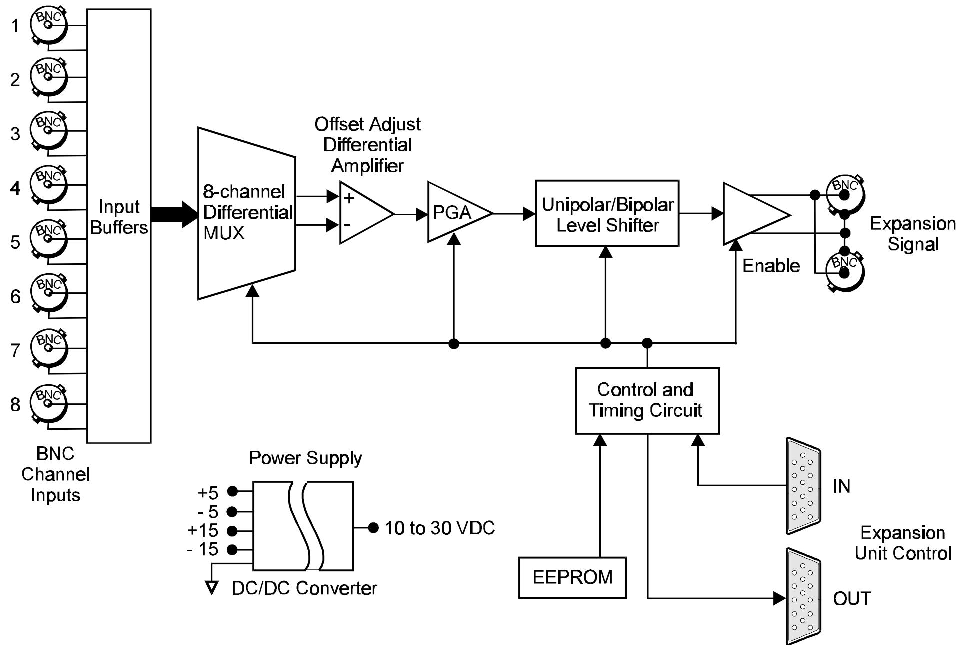
Battery Capacity (Amp-Hours):
5 A-hr in 12 V mode (parallel)
2.5 A-hr in 24 V mode (series)
Operating Temperature: -20°F to 122°F (-29°C to 50°C)
Size: 8½ × 11 × 1¾ in. (216 × 279 × 44 mm)
Weight: 7.2 lb (3.27 kg)

WBK10 - Expansion Module

Description

The WBK10 expansion module provides the WaveBook/512 with 8 additional differential analog inputs, each equipped with a programmable gain instrumentation amplifier (PGA). The WaveBook/512 and WBK10 have a built-in expansion bus. Up to eight WBK10s can be cascaded together, for a total system capacity of 72 differential channels. Each WBK10 is also capable of supporting a WBK11, WBK12, or WBK13 option card.

Physically, the WBK10 is the same size as the WaveBook/512 for convenient mounting. A fastener panel allows multiple units to be stacked vertically. Screw-on handles are available for portable applications.

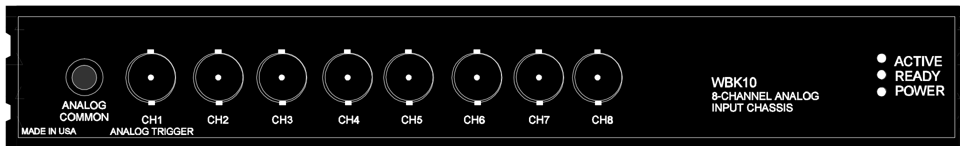


WBK10 Block Diagram

WBK10 Front Panel

The front panel of the WBK10 has the following connectors and indicators (see figure next page):

- 1 Analog Common binding post for reference
- 8 BNC connectors for analog inputs (channels are labeled 1 through 8 although additional WBK10s beyond the first will be identified by higher channel numbers as discussed in *Configuration*)
- 3 Status LEDs (Active, Ready, Power)



WBK10 Front Panel

WBK10 Rear Panel

The rear panel of the WBK10 has a power switch and the following connectors (see figure):

- 2 circular 5-pin DIN5 connectors for Power-in and Power Pass-through
- 1 HD-15M expansion control input
- 1 HD-15F expansion control output
- 2 BNC connectors for analog expansion in and out

WBK10 Rear Panel

Hardware Setup

Configuration

The analog input channel numbers are determined by the order of connection among the WaveBook/512 and attached WBK10s.

- Channel 0 is the WaveBook's 8-bit digital I/O port.
- Channels 1 through 8 are the WaveBook's main channels.
- Channels 9 through 16 are located on the first expansion unit (the one connected directly to the WaveBook).
- Additional channel numbers (in groups of 8) are added consecutively with added WBK10s (see table).

Unit	Channel #
WaveBook/512	0 (dig I/O)
WaveBook/512	1-8
1st WBK10	9-16
2nd WBK10	17-24
3rd WBK10	25-32
4th WBK10	33-40
5th WBK10	41-48
6th WBK10	49-56
7th WBK10	57-64
8th WBK10	65-72

CAUTION

If the following three conditions exist simultaneously:

- **operating WBK10s in a configuration of 4 or more modules**
- **ambient temperature ≥40°C**
- **WBK12 or WBK13 card installed,**

then you must mount the modules on their side (vertically) to facilitate air flow through the side plates. Failure to due so could result in thermal-related problems.

WBK10 Cabling

WBK expansion modules can be configured flexibly for various applications. A WBK10 connects to a WaveBook/512 or to another WBK10. A lone WBK10 in the system connects directly to the WaveBook/512. An add-on WBK10 connects to the previous WBK10 in a daisy-chain fashion. **Note**: other WBK expansion modules are connected in a similar way.

Connections must be made for signals, control messages, and power as shown in the full-page figure and described below.

Signals are carried by a CA-150-1 coax cable with BNC connectors. Each WBK10 drives a common parallel analog bus which carries the signals to the ADC in the WaveBook/512. Each WBK10 has input and output connectors for daisy-chaining multiple units.

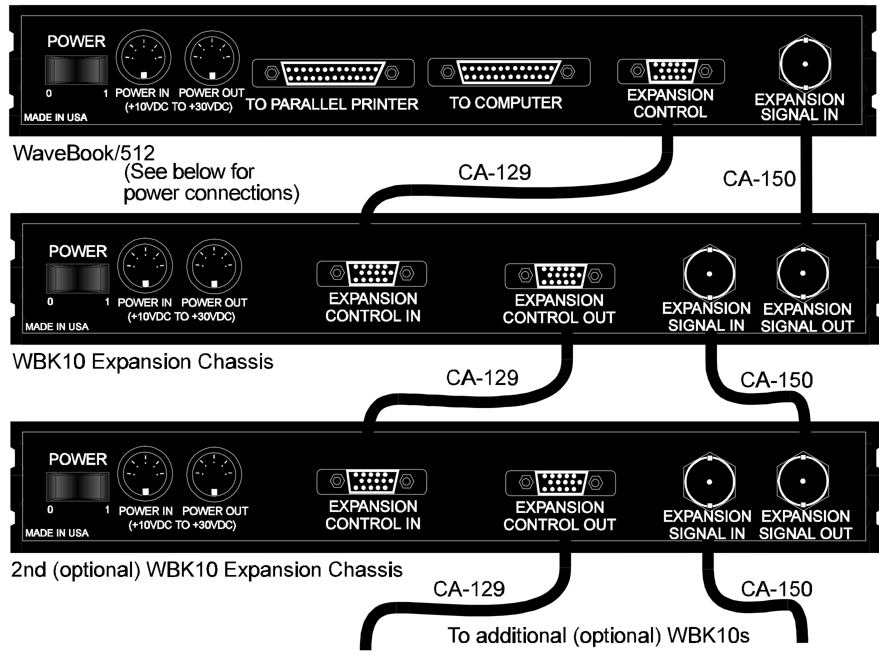
Control messages are carried by a CA-129 analog expansion control cable (HD-15, plug and socket connectors). The first expansion unit's control input is driven from the main unit's control output. Control inputs of additional WBK10s are driven from the preceding unit's control output.

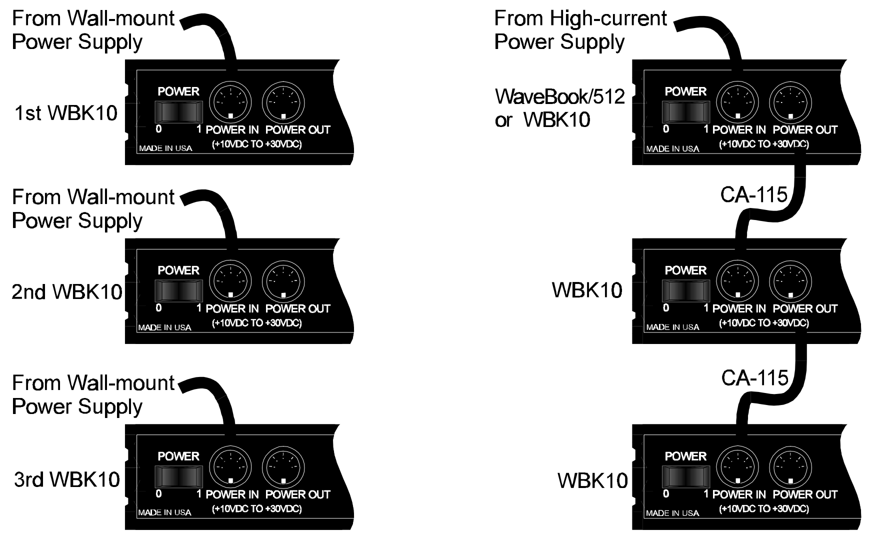
Power for the WBK10s can be supplied in 3 ways with several cable options that connect to the units via a DIN5 connector:

- Using the included power supply for one WBK10. A separate supply must be used for each WBK10 in the system. The next figure shows 3 WBK10s, each with a wall-mounted power supply.
- Providing a high-current supply for several WBK10s. A single high-current DC supply such as a TR-22 can power several WBK10s daisy-chained together with CA-115 power cables (6-inch length). (The number of WBK10s is limited to the amount of power available and the amount of power used by option cards.) See the next figure for connection details, and Appendix C for power requirements.

- The DBK30A can provide battery power for portable applications via the CA-115 power cable. Refer to preceding DBK30A section of this chapter.
- Other 10 to 30 VDC power source such as a car battery.

Note: For custom power cabling, refer to the DIN5 connector's pinout in Appendix E. For power requirements, refer to Appendix C.





WBK10 Cable Connections

Software Setup in WaveView

Depending on your application, you will need to set several software parameters so that WaveView will organize your system to your requirements. Two overview screens (shown later in this chapter and detailed in chapter 5) include:

- WaveView Configuration displays all channels in the system with their setup status.
- Module Configuration has a System Inventory box that displays all modules and attached option cards in the system.

WBK10 - Specifications

Name/Function: WBK10 8-Channel Analog Expansion Module
Number of Channels: 8 differential
Connector: BNC
Accuracy: ±0.025% FS
Offset: ±1 LSB max
Maximum Overvoltage: 30 VDC
Ranges: Unipolar/Bipolar operation is software selectable via sequencer
  Unipolar: 0 to +10 V, 0 to +5 V, 0 to +2 V, 0 to +1 V
  Bipolar: -5 to +5 V, -2.5 to +2.5 V, -1 to +1 V, -0.5 to +0.5 V
Input Current: 50 nA typ, 500 nA max
Input Impedance
  Single-ended: 5 MΩ in parallel 30 pF
  Differential: 10 MΩ in parallel 30 pF
Gain Temperature Coefficient: 5 ppm/°C typ
Offset Temperature Coefficient: 12 uV/°C max
Power: 0.6 A max @ 15 VDC
Dimensions: 220 mm wide × 285 mm long × 35 mm high (8.5” × 11” × 1.375”)
Weight: 1.3 kg (2.8 lb)

WBK11 - Simultaneous Sample & Hold Card

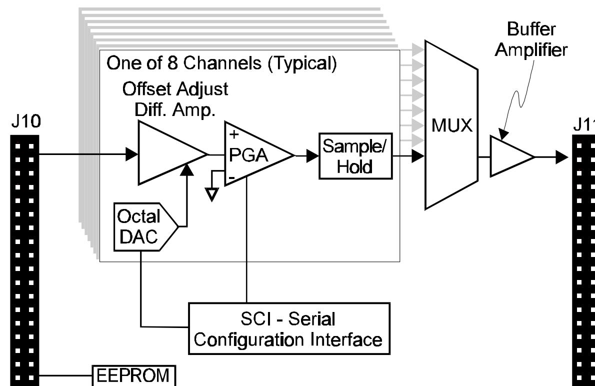
Description

The WBK11 is a field-installable sample-and-hold (SSH) card that can simultaneously sample 8 channels. The WBK11 installs internally into a WaveBook/512 or a WBK10 and is controlled by the WaveBook/512. The WBK11 allows concurrent (<150 ns) capture of multiple input channels and virtually eliminates channel-to-channel time skewing. The WBK11 also extends input voltage ranges to include ±0.05, ±0.1, ±0.25, and unipolar 0.1, 0.2, and 0.5 V. **Note**: With an SSH channel enabled, per-channel sample rates are reduced by the same amount as adding an additional channel. The per-channel rate with SSH is ***1 MHz / (n+1)***, where ***n*** is the number of active channels.

The WBK11 SSH card can accommodate higher gains than the main unit because its gains are fixed for each channel prior to the acquisition. Each channel may be set, in software for ranges shown in the table. All channels equipped with SSH circuitry are sampled simultaneously as a system.

WBK11 Voltage Ranges	
0-10.0 V	±5.0 V
0-5.0 V	±2.5 V
0-2.0 V	±1.0 V
0-1.0 V	±0.5 V
0-0.5 V	±0.25 V
0-0.2 V	±0.10 V
0-0.1 V	±0.05 V

The next figure shows a block diagram of the WBK11.



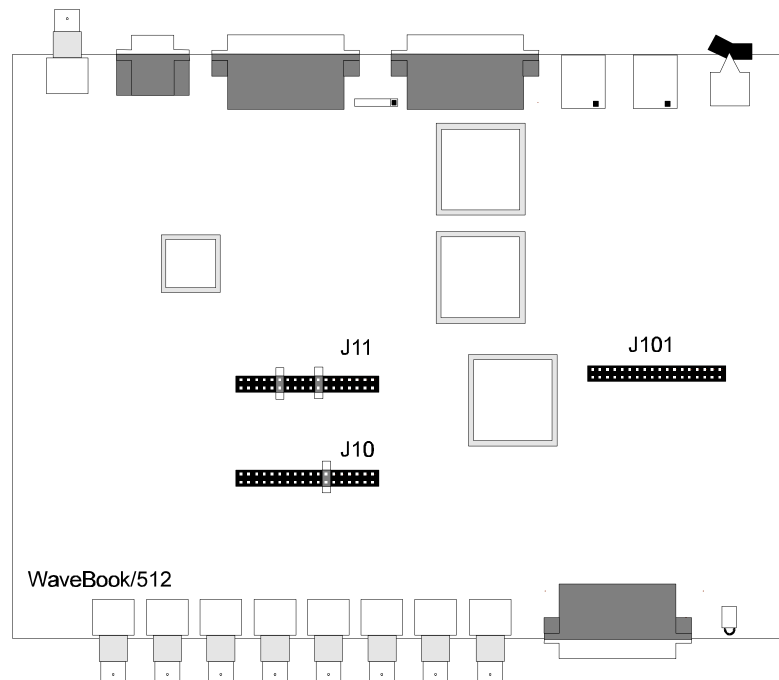
WBK11 Block Diagram

WBK11 Hardware Setup

The next figure shows the board layout for the WaveBook/512. The WBK10 layout is similar in those areas which connect to the WBK11.

There are 2 headers in the WBK10 (3 in the WaveBook/512):

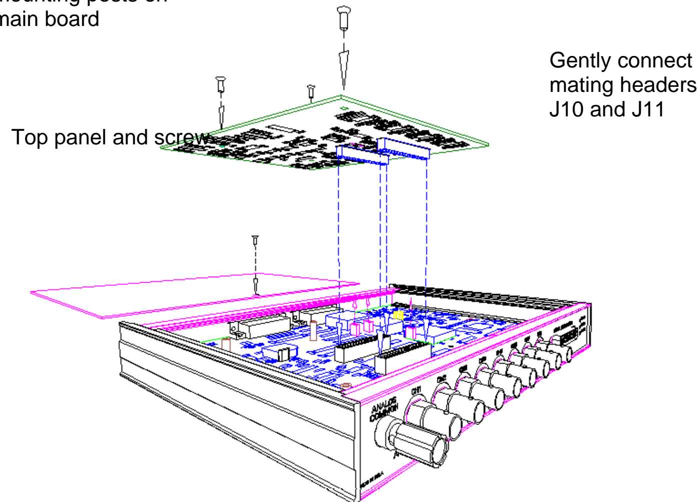
- J10 & J11 - Used for the connection of the WBK11 Simultaneous Sample & Hold option card. The jumpers located on J10/J11 provide signal pass-through when the WBK11 is not installed in the WaveBook.
- (WaveBook only - J101 reserved for future use)



Board Layout of WaveBook/512

The following steps detail the procedure to install the WBK11 card into the WaveBook/512 or WBK10 module.

1. Remove power from the unit.
2. Remove the screw holding down the top panel (cover).
3. Slide the panel out towards the back and remove.
4. Locate the two headers (J10 & J11) on the main board, and remove the jumpers (see previous figure). Save the jumpers in the event the SSH board needs to be removed.
5. The WBK11 fits into the host device only one way. Turn the WBK11 over so that the headers of both boards face each other. Position the board so that the headers of the WBK11 are lined up directly above the host board headers (see figure).



WBK11 Connection to WBK10 (or WaveBook\512)

6. Move the WBK11 board down until the connectors mate. Push down gently to seat them.
7. With screws, secure the WBK11 to the mounting posts on the main board.
8. Slide the top panel into the unit.
9. Reinstall the top panel screw.
10. Power up the unit and run WaveView to verify that the channels connected to the WBK11 now have the extra ranges available.

WBK11 - Specifications

Name/Function: WBK11 8-Channel Simultaneous Sample-and-Hold Card
Number of Channels: 8
Connectors: Internal to the WaveBook/512 (36-pin sockets mate with 36-pin connectors)
Accuracy: ±0.025% FS
Offset: ±1 LSB max
Aperture Uncertainty: 75 ps max
Voltage Droop: 0.1 mV/ms max
Maximum Signal Voltage: ±5.00 VDC (×1)
Input Voltage Ranges: Software programmable prior to a scan sequence; expands WaveBook/512 ranges to:
  Unipolar: 0 to +10 V, 0 to +5 V, 0 to +2 V, 0 to +1 V, 0 to +0.5 V
  Bipolar: -5 to +5 V, -2.5 to +2.5 V, -1 to +1 V, -0.5 to +0.5 V, -0.05 to +0.05 V
Programmable Gain Amplifier Gain Ranges: ×1, 2, 5, 10, 20, 50, 100
Weight: 0.14 kg (0.3 lb)

WBK12/13 Programmable Low-Pass Filter Cards

Description

The WBK12 and WBK13 are 8-channel programmable low-pass filter cards for use with the 1-MHz WaveBook/512 data acquisition system. These cards install directly into the WaveBook/512 or WBK10 expansion module and provide programmable low-pass filtering over all channels. Multiple WBK12 and WBK13 cards can be installed in one system for up to 72 channels. All of the cards' low-pass filters and cutoff frequencies are configured via software.

The WBK13 has the additional capability of simultaneous sampling all channels. If multiple WBK13 cards are installed within one system, all channels are sampled within 100 ns of each other.

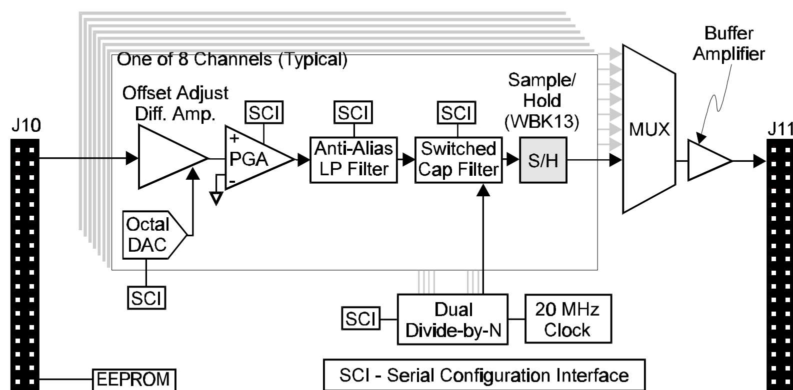
Features of the WBK12 and WBK13 include:

Low-Pass Filters. Each card provides 8 input channels, arranged in two 4-channel banks; the filter and cutoff frequency configurations are applied per bank. The cards' filters can be configured as either an 8-pole elliptic filter with cutoff frequencies of 400 Hz to 100 kHz, or an 8-pole linear-phase filter with 400 Hz to 50 kHz cutoff frequencies.

Cutoff Frequencies. The WBK12 and WBK13 provide 747 discrete cutoff frequencies that can be determined exactly by the formula *Fc = 300 kHz/N; where the integer N = 3 to 750*. Alternatively, you can configure any channel to bypass the programmable filter entirely, resulting in a 1-pole low-pass filter at about 500 kHz.

Programmable-Gain Amplifiers. The cards' programmable-gain instrumentation amplifiers can be software selected to gains of ×1, 2, 5, 10, 20, 50, and 100 on a per channel basis. The WBK12 and WBK13 provide 6 ranges with full-scale inputs from ±50 mV to ±5 V bipolar (+100 mV to +10 V unipolar). These gains are set prior to the beginning of an acquisition sequence and cannot be changed during an acquisition.

(WBK13 only) **Simultaneous Sample-and-Hold (SSH)**. In addition to the filtering capability of the WBK12, the WBK13 also provides per channel SSH. Simultaneous sampling of all channels occurs at the start of a scan sequence. **Note**: With an SSH channel enabled, per-channel sample rates are reduced by the same amount as adding an additional channel. The per-channel rate with SSH is ***1 MHz / (n+1)***, where ***n*** is the number of active channels. The figure shows a block diagram of the WBK12 and WBK13.



WBK12 and WBK13 Block Diagram

Hardware Setup

Configuration

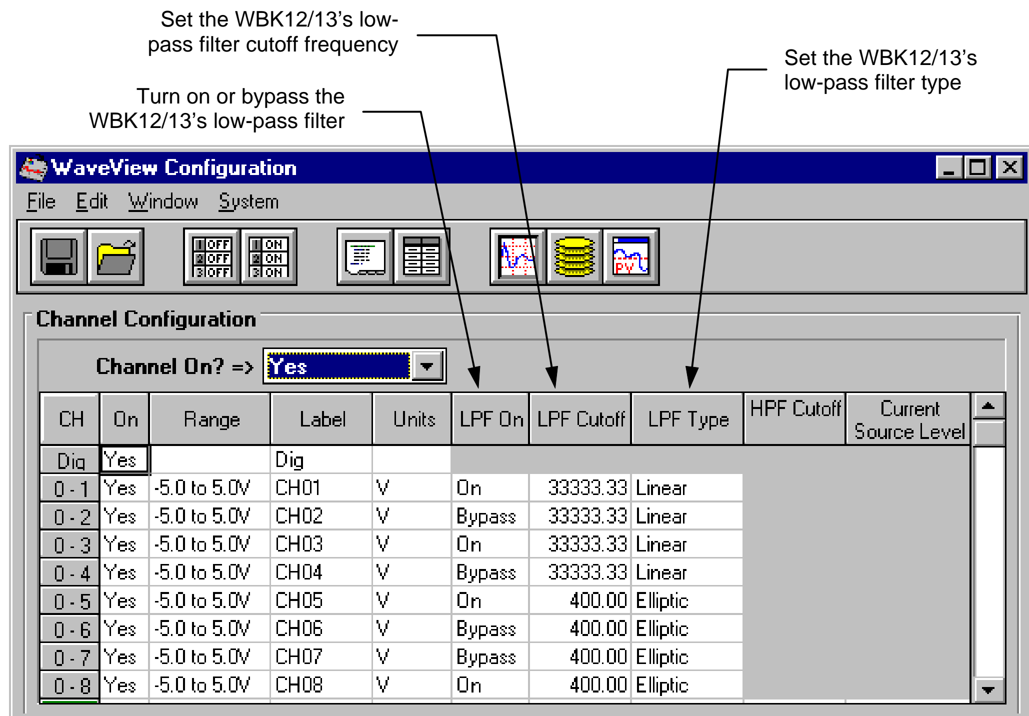
The WBK12 and WBK13 require no hardware setting (all configurations by software).

Connection

The WBK12 and WBK13 connect to a WaveBook/512 or WBK10 in much like the WBK11 (refer to previous section).

Software Setup in WaveView

Depending on your application, you will need to set several software parameters so that WaveView will organize data to your requirements. The next figure shows the WaveView Configuration screen and calls out several parameters of importance to the WBK12/13 (specifically the low-pass filter options). **Note**: if necessary, refer to chapter 5 *Using WaveView* for more information on using the columns to configure a channel.



Software Function

The following code (in C language with the standard API) demonstrates the function calls to configure a WBK12* located in the main WaveBook chassis. ***Note**: The WBK13 program is the same except the parameters use ...Wbk13... instead of ...Wbk12....

```
// Configure channel 1 for: low-pass filter on with a cutoff of 1000Hz, using a linear
         filter type and the software-chosen pre-filter.
wbkSetChanOption( 1, 1, WcotWbk12FilterCutOff, 1000.0);
wbkSetChanOption( 1, 1, WcotWbk12FilterMode,
   (double) WcovWbk12LowPassOn );
wbkSetChanOption( 1, 1, WcotWbk12FilterType,
   (double) WcovWbk12FilterLinearPhase);
wbkSetChanOption( 1, 1, WcotWbk12PreFilterMode,
   (double) WcovWbk12PreFilterDefault);
```

```
// Configure channel 5 for: low-pass filter on with a cutoff of 20000Hz, using an
        elliptic filter type and the software-chosen pre-filter.
wbkSetChanOption( 5, 1, WcotWbk12FilterCutOff, 20000.0);
wbkSetChanOption( 5, 1, WcotWbk12FilterMode,
  (double) WcovWbk12LowPassOn );
wbkSetChanOption( 5, 1, WcotWbk12FilterType,
  (double) WcovWbk12FilterElliptic);
wbkSetChanOption( 5, 1, WcotWbk12PreFilterMode,
  (double) WcovWbk12PreFilterDefault);

// Turn on all the low-pass filters on channels 6 through 8.  NOTE: These channels
        will use the same cutoff frequency and filter type as channel 5.
wbkSetChanOption( 6, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassOn);
wbkSetChanOption( 7, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassOn);
wbkSetChanOption( 8, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassOn);

// Turn off all the low pass filters on channels 2 through 4.
wbkSetChanOption( 2, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassBypass);
wbkSetChanOption( 3, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassBypass);
wbkSetChanOption( 4, 1, WcotWbk14LowPassMode,
  (double) WcovWbk12LowPassBypass);
```

WBK12/13 - Specifications

Name/Function: WBK12, Programmable Low-Pass Filter Card
WBK13, Programmable Low-Pass Filter Card With SSH
Number of Channels: 8
Connector: Internal to WaveBook/512 and WBK10 (two 36-pin sockets mate with 36-pin connectors)
Input Voltage Ranges: Software programmable prior to a scan sequence

Unipolar		Bipolar	
Voltage Range	**Gain**	**Voltage Range**	**Gain**
0 to +0.1 V	×100	±0.05 V	×100
0 to +0.2 V	×50	±0.1 V	×50
0 to +0.5 V	×20	±0.25 V	×20
0 to +1 V	×10	±0.5 V	×10
0 to +2 V	×5	±1 V	×5
0 to +5 V	×2	±2.5 V	×2
0 to +10 V	×1	±5 V	×1

Programmable Gain Amplifier Ranges: ×1, 2, 5, 10, 20, 50, and 100
Switched Capacitor Filter Cutoff Frequencies Range: 400 Hz to 100 kHz
Number of Cutoff Frequencies: 747
Filter Grouping: 4 channels each in 2 programmable banks
Low-Pass Filter: Software selectable, 8-pole elliptic filter
Low-Pass Filter Type: Software selectable, elliptic or linear phase
Low-Pass Filter Frequency Cutoff Range: 100 kHz, 75 kHz, 60 kHz...400 Hz,
bypass defined as Fc = 300 kHz/N where N = 3 to 750
Anti-Alias Frequencies: determined by software control
Accuracy: ±0.05% FS DC
Offset: ±1 LSB max
Aperture Uncertainty: 75 ps max
Voltage Droop: 1 mV/ms max (0.01 mV/ms typ)
Maximum Signal Voltage: ±5.00 VDC (×1)
THD: -65 dB (-70 dB typ)
Noise: 3 counts (RMS)
DC Offset: ±2.5 mV (2 LSB) max at any cutoff frequency
Number of Cutoff Frequencies Simultaneously Set: 2, one for each 4-channel bank of inputs
Weight: 0.14 kg (0.3 lb)

WBK14 Dynamic Signal Input Module

Introduction

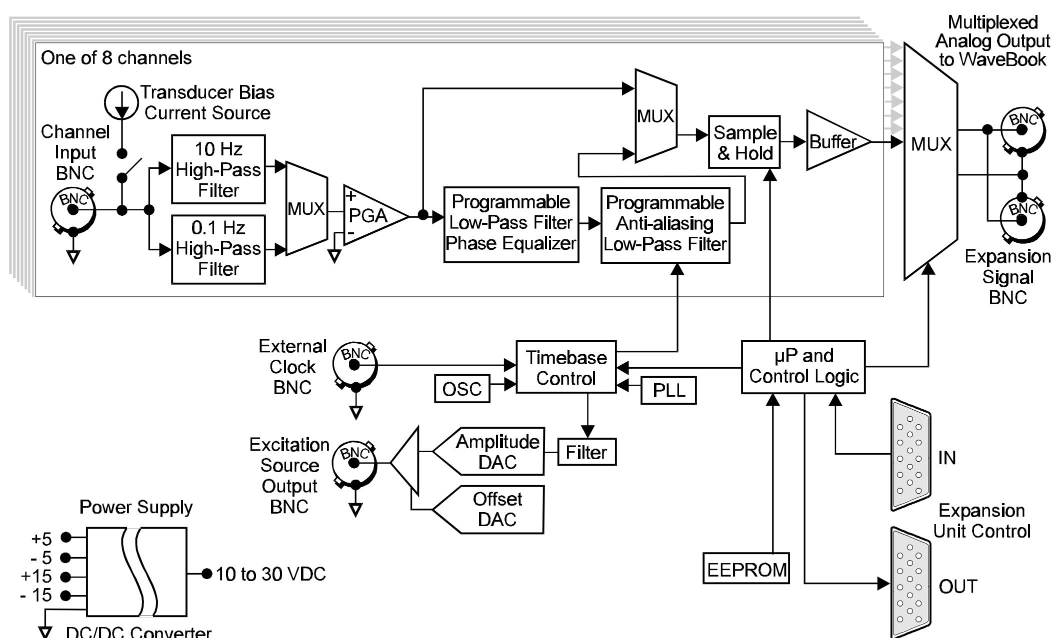
The WBK14 is a dynamic analog signal input module for the 1 MHz WaveBook/512 data acquisition system. The WBK14 provides a complete system to interface to piezoelectric transducers that include accelerometers, microphones, force/pressure transducers, and others. **Note**: A tutorial section near the end of this section (before the specifications) explains how to use accelerometers as well as some related theory of operation.

Each WBK14 channel has: a current source for transducer biasing, a high-pass filter, a programmable gain amplifier, an anti-aliasing low-pass filter, and sample-and-hold amplifiers. The gain, filter cut-off frequencies and current biasing levels are software programmable.

The WBK14 also includes a built-in programmable excitation source to provide stimulus to dynamic systems for transfer function measurements, as well as reference signals for calibration.

Hardware Description

The following text refers to the block diagram in the next figure.



WBK14 Block Diagram

Current Source

The WBK14 provides a constant current to bias ICP transducers. Two current levels (2 mA or 4 mA) with voltage compliance of 27 V can be selected via software. The bias current is sourced through the center conductor of a coaxial lead and returns to the WBK14 by the outer conductor. The output impedance is larger than 1 MΩ and presents virtually no loading effect on the transducer's output. For applications that do not require bias, the current source can be removed from the BNC input by opening a relay contact. The current sources are applied to (or removed from) the input in groups of two (i.e. channels 1-2, 3-4, 5-6, 7-8).

High-Pass Filter (HPF)

Each WBK14 channel has two independent HPFs with a 3 dB cut-off frequency (Fc) at 0.1 Hz and 10 Hz. The 0.1-Hz HPF is a single-pole RC filter, and is primarily used to couple vibration signals. The 10-Hz HPF is a two-pole Butterworth type and can be used to couple acoustic signals or attenuate setup-induced low-frequency signals that can reduce the dynamic range of the measurement (e.g. when using tape recorders as signal sources).

Programmable Gain Amplifier (PGA)

The HPF removes the DC voltage from the input signal. The AC voltage is amplified by a PGA with flat response up to 500 kHz. Each channel has a PGA with 8 programmable gains (1, 2, 5, 10, 20, 50, 100, 200) and a software-controlled DAC for offset nulling. The WBK14 measures only bipolar signals up to 5 V peak.

Programmable Low-Pass Filter Phase Equalizer (PLPPE)

The first filter stage is a programmable 2-pole continuous-time low-pass filter. The PLPPE provides more than 65 dB alias protection to the next filter stage. In addition, it fine-tunes the phase shift of the channel to optimize the phase-matching between channels. At calibration, the phase shift of each channel is measured and stored in a EEPROM, that is read at configuration.

Programmable Low-Pass Anti-Aliasing Filter (PLPAF)

Most of the signal alias rejection is performed by an 8-pole Butterworth filter. This filter is implemented with a switch capacitor network driven by a programmable clock. Each channel has an independent clock whose frequency determines the 3 dB cut-off frequency of the filter. The switch capacitor filter provides no attenuation at the clock frequency, (hence, the need for the continuous-time low-pass filter). **Note:** The PLPAF can be bypassed to process signals with a bandwidth higher than 100 kHz.

The EXT.CLK input provides a path to externally control the cut-off frequency of the PLPAF. The input waveform can be TTL or sinusoidal, with an amplitude peak of at least 500 mV. In this mode, the cut-off frequency is set to the input frequency divided by 50.

Simultaneous Sample and Hold

All WBK14 channels are sampled simultaneously, after which the WaveBook/512 measures each output at 1 µs/channel until all channels are digitized. The time-skew between sampling on all channels (up to72) is 150 ns, regardless of the number of WBK14s attached to the WaveBook/512. **Note**: With an SSH channel enabled, per-channel sample rates are reduced by the same amount as adding an additional channel. The per-channel rate with SSH is ***1 MHz / (n+1)***, where ***n*** is the number of active channels.

Excitation Source

The excitation source includes a sine/random waveform generator, a programmable gain amplifier, a DC level DAC, and a phase-lock loop (PLL). The PLL is used to synthesize the frequency of a fixed amplitude sine wave and control the bandwidth of the random signals. The PGA conditions the signal amplitude to a value between 0 V to 5 V peak. The DC level of the signal is varied independently of signal amplitude by a software-controlled DAC from -5 V to +5 V. The DC level of the excitation signal can be used to balance static loads, while the AC signal provides the dynamic excitation.

Power

Like the WaveBook/512, the WBK14 contains an internal power supply. The unit can be powered by an included AC power adapter or directly from any 10 to 30 VDC source, such as a 12 V car battery. For portable or field applications, the WBK14 and the WaveBook/512 can be powered by the DBK30A rechargeable battery module. **Note**:

- For custom power cabling, refer to the DIN5 connector's pinout in Appendix E.
- For power requirements, refer to Appendix C. You must compute power consumption for your entire system and (if necessary) use auxiliary or high-current power supplies.

Calibrating the WBK14

The WBK14 is calibrated digitally, eliminating the need for all potentiometers and manual adjustments. A Windows-based calibration application provided with the unit, simplifies the calibration process.

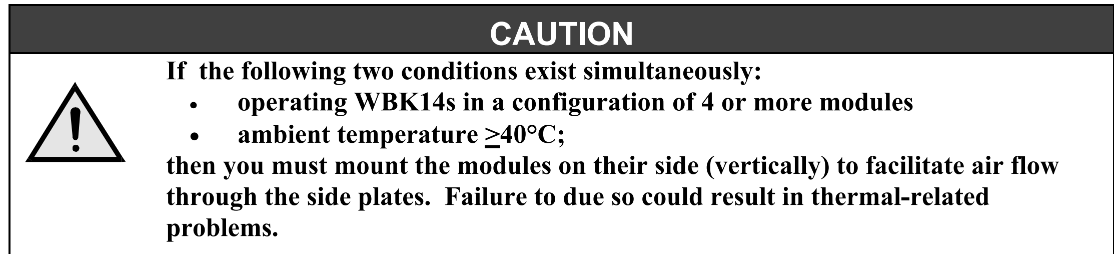
Hardware Setup

Configuration

The WBK14 requires no hardware setting. All configurations are controlled by software.

Connection

The WBK14 is connected to a WaveBook/512 or WBK10 in much the same way as the WBK10, as previously described.



CAUTION

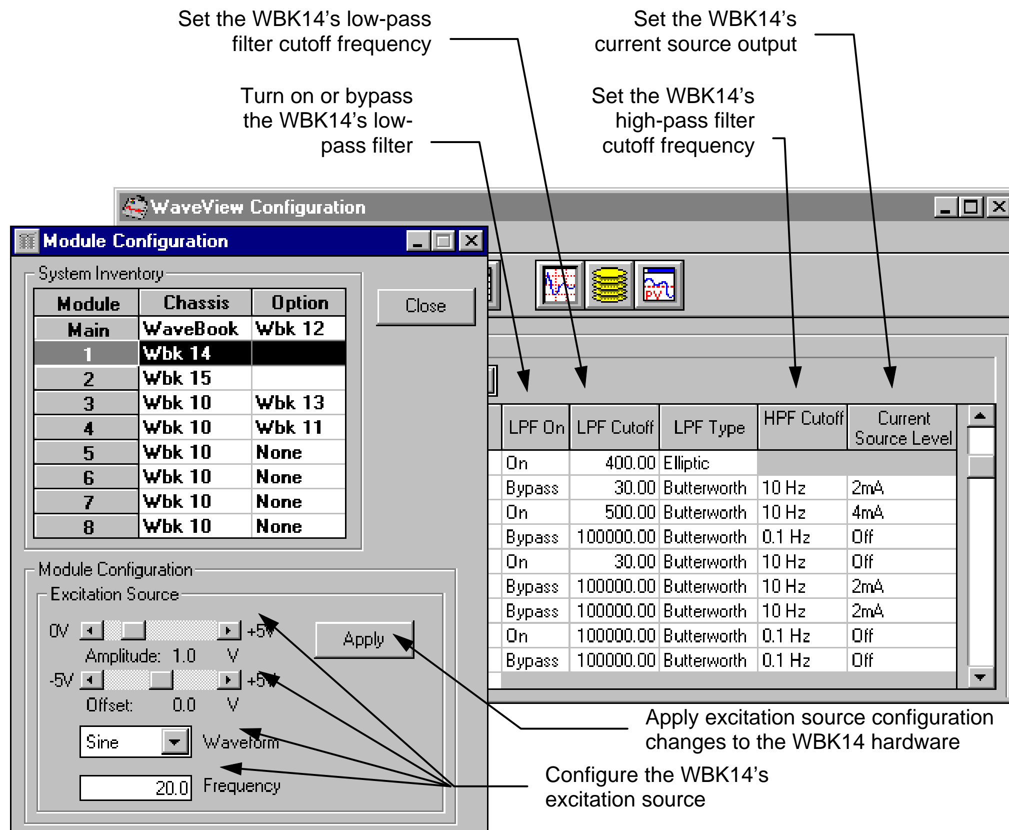
If the following two conditions exist simultaneously:

- **operating WBK14s in a configuration of 4 or more modules**
- **ambient temperature ≥40°C;**

then you must mount the modules on their side (vertically) to facilitate air flow through the side plates. Failure to due so could result in thermal-related problems.

Software Setup in WaveView

Depending on your application, you will need to set several software parameters so that WaveView will organize data to your requirements. The next figure shows the WaveView Configuration screen and calls out several parameters of importance to the WBK14 (specifically the low-pass and high-pass filter options). **Note**: if necessary, refer to chapter 5 *Using WaveView* for more information on using the columns to configure a channel.



Software Function

The following code (in C language) demonstrates the function calls to configure a WBK14.

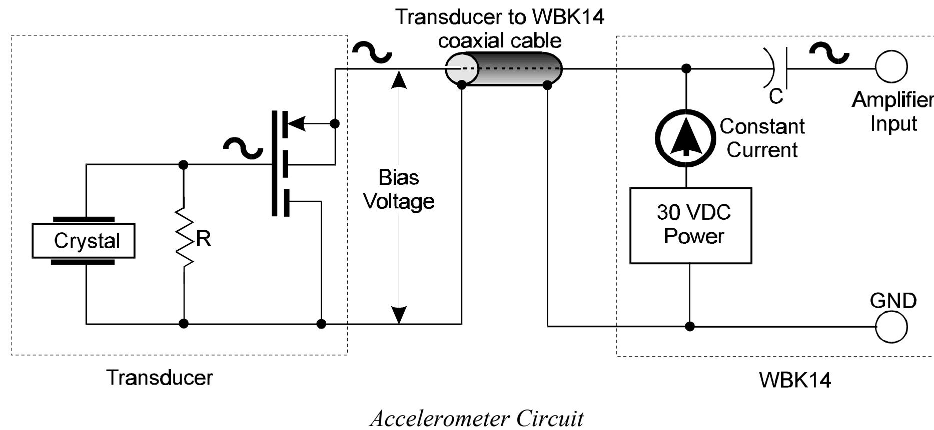
```
// Configure channel 9 for: low pass filter on with a cutoff of 500Hz, using the
       software chosen pre-filter, high pass filter cutoff at 10Hz, and current source
       set to 2mA.
wbkSetChanOption( 9, 0, WcotWbk14LowPassCutOff, 500.0);
wbkSetChanOption( 9, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassOn );
wbkSetChanOption( 9, 0, WcotWbk14HighPassCutOff,
   (double) WcovWbk14HighPass10Hz );
wbkSetChanOption( 9, 0, WcotWbk14CurrentSrc,
   (double) WcovWbk14CurrentSrc2mA );
wbkSetChanOption( 9, 0, WcotWbk14PreFilterMode,
   (double) WcovWbk14PreFilterDefault );

// Configure channel 10 for: low pass filter on with a cutoff of 20000Hz, using the
       software chosen  pre-filter, high pass filter cutoff at 0.1Hz, and current
       source set to 4mA.
wbkSetChanOption( 10, 0, WcotWbk14LowPassCutOff, 20000.0);
wbkSetChanOption( 10, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassOn );
wbkSetChanOption( 10, 0, WcotWbk14HighPassCutOff,
   (double) WcovWbk14HighPass0_1Hz );
wbkSetChanOption( 10, 0, WcotWbk14CurrentSrc,
   (double) WcovWbk14CurrentSrc4mA );
wbkSetChanOption( 10, 0, WcotWbk14PreFilterMode,
   (double) WcovWbk14PreFilterDefault );

// Turn all the low pass filters on the other channels (11-16) off
wbkSetChanOption( 11, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);
wbkSetChanOption( 12, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);
wbkSetChanOption( 13, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);
wbkSetChanOption( 14, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);
wbkSetChanOption( 15, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);
wbkSetChanOption( 16, 0, WcotWbk14LowPassMode,
   (double) WcovWbk14LowPassBypass);

// Setup the excitation source to output a sine wave at 100Hz, with an amplitude of
       2.5V, and an offset voltage of 0V.  NOTE: the channel number can be any channel
       on the Wbk14 module to be programmed.
wbkSetModuleOption( 9, 0, WmotWbk14ExcSrcWaveform,
    (double) WmovWbk14WaveformSine);
wbkSetModuleOption( 9, 0, WmotWbk14ExcSrcFreq, 100.0);
wbkSetModuleOption( 9, 0, WmotWbk14ExcSrcAmplitude, 2.5);
wbkSetModuleOption( 9, 0, WmotWbk14ExcSrcOffset, 0.0);
```

Accelerometer Tutorial

A low-impedance piezoelectric accelerometer consists of a piezoelectric crystal and an electronic amplifier. When stretched or compressed, the crystal develops a charge variation between its two surfaces that is related to the amount of stress, shock, or vibration on the crystal. The amplifier transforms the sensor's high impedance to the output impedance of a few hundred ohms. Low-impedance piezoelectric transducers are used to measure pressure and force as well as acceleration.

The accelerometer circuit requires only 2 wires (coax or twisted pair) to transmit both power and signal. At low impedance, the system is insensitive to externally induced or "triboelectric" cable noise. Sensitivity is not affected by cable length.

The figure shows a simplified accelerometer-WBK14 connection. The voltage developed across R is applied to the gate of the MOSFET. The MOSFET is powered from a constant current source of 2 or 4 mA and 27 volts.



Accelerometer Circuit

The MOSFET circuit will bias off at approximately 12 V in the quiescent state. As the system is excited, voltage is developed across the crystal and applied to the gate of the MOSFET. This voltage will cause linear variation in the impedance of the MOSFET, which will cause a proportional change in bias voltage. This voltage change will be coupled to the WBK14 input amplifier through the capacitor C. The low frequency corner is controlled by the value of R and the internal capacitance of the piezoelectric crystal. Units weighing only a few grams can provide high level outputs up to 1 V/g with response to frequencies below 1 Hz.

Accelerometer Specification Parameters

Noise in Accelerometers

The noise floor or resolution specifies lowest discernible amplitude (minimum “g”) that can be measured. There are two main sources of noise. **Noise from the crystal** and microcircuit inside the accelerometer. Some types of crystals, such as quartz, are inherently more noisy than others. A good accelerometer noise floor is 10 to 20 µV. **Noise from electrical activity on the mounting surface**. Since the signal from the accelerometer is a voltage, 60 Hz or other voltages (ground loop) will interfere with the signal. The best protection is to electrically isolate the accelerometer.

Sensitivity

The sensitivity of a low-impedance accelerometer is defined as its output voltage per unit input of motion. The unit of motion used is the “g”. One “g” is equal to the Earth’s gravitational acceleration which is 32.2 ft/(sec)(sec), 386.1 in/(sec)(sec), or 981 cm/(sec)(sec). The output is usually specified in millivolts per “g” (mV/g). Sensitivity is usually specified under defined conditions (frequency, testing levels, and temperature)—for example, 100 mV/g at a frequency of 100 Hz, level +1 g, at 72°F. While a given sensor model may have a “typical” sensitivity of 100 mV/g, its actual sensitivities may range from 95 to 105 mV/g when checked under stated conditions. Calibration values for individual sensors are typically provided by the manufacturer.

Transverse Sensitivity

Accelerometers are designed to have one major axis of sensitivity, usually perpendicular to their base and colinear with its major cylindrical axis. The output caused by the motion perpendicular to the sensing axis is called the transverse sensitivity. This value varies with angle and frequency and typically is less than 5% of the basic sensitivity.

Frequency Response

The frequency response of an accelerometer is defined as the ratio of the sensitivity of the accelerometer measured at frequency (f) to the basic sensitivity measured at 100 Hz. This response is usually obtained at a constant acceleration level, typically 1 g or 10 g. Convention defines the usable range of an accelerometer as the frequency band in which the sensitivity remains within 5% of the basic sensitivity. Measurements can be made outside these limits if corrections are applied. Care should be taken at higher frequencies because mounting conditions greatly affect the frequency range (see mounting effects).

Bias Level

Under normal operation, a bias voltage appears from the output signal lead to ground. There are two basic MOSFET configurations commonly used. One exhibits a 7-8 V bias and the second a 9-12 V bias. Operation of the two circuits is identical except for the available signal swing. The low voltage version typically exhibits 5-10 µVrms versus 10-20 µVrms for the high voltage.

Dynamic Range

The dynamic measurement range is the ratio of the maximum signal (for a given distortion level) to the minimum detectable signal (for a given signal-to-noise ratio). The dynamic range is determined by several factors such as accelerometer sensitivity, bias voltage level, power supply voltage, and noise floor.

Thermal Shock - Temperature Transients

Piezoelectric accelerometers exhibit a transient output which is a function of “rate-of-change” temperature.

This phenomenon., called “thermal shock”, is usually expressed in g/°C, and is related to the non-uniform mechanical stresses set up in the accelerometer structure and the so-called pyroelectric effect in piezoelectric materials whereby an electrical charge is produced by the temperature gradient across the crystal. In practice, the effect is quasistatic, producing a low-frequency voltage input to the MOSFET amplifier. While usually occurring well below the low-frequency corner, the effect can momentarily reduce the peak clipping level and cause loss of data. The phenomenon does not affect the basic sensitivity of the accelerometer and does not affect the data unless the thermal shift in the operation bias level results in clipping. Where drastic thermal shifts are expected, the use of 12 V bias models is recommended. The effect's severity is related to the mass of the accelerometer. In 100 mV/g industrial units, the effect is usually negligible. The effect can be reduced significantly by using rubber thermal boots provided for that purpose

Base-Strain Sensitivity

Strain sensitivity in an accelerometer is defined as the output caused by deformation of the base due to bending in the structure on which it is mounted. In measurements made on large structures with low natural frequencies, significant bending may occur; units with low base strain sensitivity should be selected. Base strain effects can be substantially reduced by inserting a washer smaller than the accelerometer base diameter, under the accelerometer base to reduce the contact surface area. This technique lowers the usable upper frequency range.

Connector

This specifies the connector type and size (4-48, 6-40, 10-32 coaxial etc.) and the location on the sensor, i.e., top or side (usually on the hex base). In cases where there is no connector on the sensor, an integral cable is specified with the length and particular connector, i.e., integral 6 ft to 10-32.

Acoustic Sensitivity

High-level acoustic noise can induce outputs unrelated to vibrational input. In general, the effect diminishes as the accelerometer mass increases. This effect may be reduced by using a light, foam rubber boot.

Overload Recovery

Recovery from clipping due to over-ranging is typically less than one millisecond. Recovery from quasi-static overloads which generate high DC bias shifts are controlled by the accelerometer input RC time constant which is fixed during manufacture.

Power Supply Effects

The nominal power supply voltage recommended by most manufacturers is 15 to 24 V. Units may be used with voltages up to 28 volts. Sensitivity variations caused by voltage change is typically 0.05%/volt. Power supply ripple should be less than 1 mV rms.

Electrical Grounding

Case-Grounded Design

In case-grounded designs, the common lead on the internal impedance matching electronics is tied to the accelerometer case. The accelerometer base/stud assembly forms the signal common and electrically connects to the shell of the output connector. Case-grounded accelerometers are connected electrically to any conductive surface on which they are mounted. When these units are used, care must be exercised to avoid errors due to ground noise.

Isolated-Base Design

To prevent ground noise error many accelerometers have base-isolated design. In these models the outer case/base of the accelerometer is isolated electrically off ground by means of an isolation stud insert. The proprietary material used to form the isolation provides exceptional strength and mechanical stiffness to preserve high-frequency performance.

Cable Driving

Operation over long cables is a concern with all types of sensors. Concerns involve cost, frequency response, noise, ground loops, and distortion caused by insufficient current available to drive the cable capacitance.

The cost of long cables can be reduced by coupling a short (1 m) adapter cable from the accelerometer to a long low-cost cable like RG-58U or RG-62U with BNC connectors. Since cable failure tends to occur at the accelerometer connection where the vibration is the greatest, only the short adapter cable would need replacement.

Capacitive loading in long cables acts like a low-pass, second-order filter and can attenuate or amplify high-frequency signals depending on the output impedance of the accelerometer electronics. Generally this is not a problem with low-frequency vibration (10 Hz to 2000 Hz). For vibration and shock measurements above 2000 Hz and cables longer than 100 ft, the possibility of either high-frequency amplification or attenuation should be considered.

The WBK14 constant-current source provides 2 or 4 mA to integral electronics. Use the higher current setting for long cables, high peak voltages, and high signal frequencies.

The maximum frequency that can be transmitted over a given length of cable is a function of both the cable capacitance and the ratio of the maximum peak signal voltage to the current available from the constant current source:

$$f = \frac{K}{2\pi C\left(\frac{V}{Icc - Ib}\right)}$$

Drive Current (mA)	Cable Length @30 pF/ft (Ft)	Frequency Response to 5% of Maximum Output Signal Amplitude	
		± 1 V	± 5 V
2	10	185 kHz	37 kHz
2	100	18.5 kHz	3.7 kHz
2	1000	1.85 kHz	370 Hz
4	10	550 kHz	110 kHz
4	100	55 kHz	11 kHz
4	1000	5.5 kHz	1.1 kHz

Where:
f = Maximum frequency in Hz
C = Cable capacitance in picoFarads
V = Maximum peak measured voltage from sensor in volts
Icc = Constant current from current source in mA
Ib = Current required to bias the internal electronics, typically 1 mA
K = 3.45×10^9. K is the scale factor to convert Farads to picoFarads and Amperes to milliAmperes and a factor to allow cable capacitance to charge to 95% of the final charge.

WBK14 - Specifications

Name/Function: WBK14, 8-Channel Dynamic Signal Conditioning Module
Connectors: BNC connector, mates with expansion signal input on the WaveBook/512; two 15-pin connectors, mate with expansion signal control on the WaveBook/512; signals via 1 BNC per channel
Channels: 8
Gain Ranges: ×1, 2, 5, 10, 20, 50, 100, 200
Power Consumption: 15 Watts typical
Input Power Range: 10 to 30 VDC
Operating Temperature: 0°C to 50°C
Storage Temperature: 0°C to 70°C
Dimensions: 216 mm wide × 279 mm long × 35 mm high (8.5” × 11” × 1.375”)
Weight: 1.32 kg (2.9 lb)
ICP Current Source:
- **Output Impedance**: > 1.0 MΩ @ 20 kHz
- **Compliance**: 27 V
- **Current Levels**: 2 & 4 mA

Coupling: AC
- 10 Hz High-Pass Filter - Input Impedance: 590K
- 0.1 Hz High-Pass Filter - Input Impedance: 10 MΩ

Input Ranges:
±5.0 V, ±2.5 V, ±1.0 V, ±500 mV, ±250 mV, ±100 mV, ±50 mV, ±25 mV

Anti-Aliasing Low-Pass Filter:
- **Accuracy**: ±0.5 dB at the passband center
- **Frequency Span**: 30 Hz to 100 kHz
- **Frequency Settings**: 300 kHz / N; N = 3,4,...10000
- **Dynamic Range @ 1 kHz**: 69 Db
- **THD @ 1 kHz**: 70 db
- **Amplitude Matching**: ± 0.1 dB
- **Phase Matching**: ± 2°

Excitation Source:
- **Max. Output Voltage**: ± 10 V
- **Max. Output Current**: 10 mA
- **DC Output**: ± 5 V
- **Sine**:
  - **Frequency**: 20 Hz - 100 kHz
  - **Distortion**: < 0.1%
  - **Amplitude**: ± 5 V
  - **Steps**: 256
- **Random**:
  - **Spectral Distribution**: White, Band-limited
  - **Amplitude Distribution**: Gaussian
  - **Bandwidth**: 20 Hz - 100 kHz
  - **RMS level**: Adjustable in binary steps

External Clock:
- **Digital**: TTL levels
- **Sine**: > 500 mV peak

WBK15 5B Isolated Signal Conditioning Module

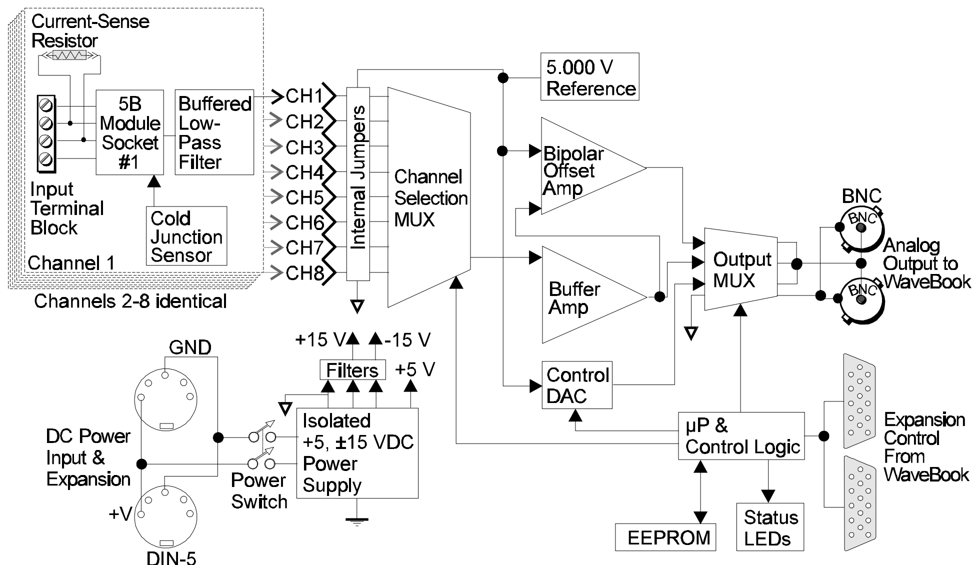
Description

The WBK15 module can accommodate eight 5B isolated-input signal conditioning modules for use with the WaveBook/512. The WaveBook/512 can accommodate 8 WBK15s for a maximum of 64 expansion channels. The WaveBook/512 scans the WBK15's channels at the same 1 µs/channel rate that it scans all WBK analog inputs, allowing it to measure all channels of a fully configured 72-channel system in 72 µs.

Other features of the WBK15 include:

- Built-in power supply that operates from 10 to 30 VDC and can power a full complement of 5B modules (even with bridge excitation)
- Removable, plug-in screw terminal blocks for convenient connection of 5B modules (each block has 4 terminals per channel for input and excitation-output features)
- On-board cold junction sensing for thermocouple 5B modules
- For each 5B module, 1500 V isolation from the system and from other channels

The following figure shows a block diagram of the WBK15.



WBK15 Block Diagram

Hardware Setup

Physically, the WBK10 is the same size as the WaveBook/512 for convenient mounting. A fastener panel allows multiple units to be stacked vertically. Screw-on handles are available for portable applications.

CAUTION

If the following two conditions exist simultaneously:

- **ambient temperature ≥40°C**
- **WBK15 is connected to 8 strain-gages and in a configuration of 4 or more modules;**

then you must mount the modules on their side (vertically) to facilitate air flow through the side plates. Failure to due so could result in thermal-related problems.

Safety Concerns

Voltages above 50 Vrms AC or 100 VDC are considered hazardous. Safety precautions are required when 5B modules are used in situations that require high-voltage isolation from the rest of the system.

The WBK15 is specified for 1500 VDC isolation in a normal environment free from conductive pollutants and condensation. The 1500 VDC rating requires a proper earth ground connection to the chassis and treatment of adjacent inputs as potentially hazardous. CE marked units used in the European community are rated at 600 VDC isolation. The 600 VDC CE isolation specification is based on a double insulation requirement, and no earth ground is required.

Input cables must be rated for the isolation potential in use. Line voltage ratings are much lower than the DC isolation values specified due to transients which occur on power lines. Never open the lid unless all inputs with potentially hazardous voltages are removed. The lid must be securely screwed on during use.

It is strongly recommended that you:

- Properly tighten all chassis screws before system use.
- Properly tighten the screw that retains the 5B module.
- Never plug in or unplug potentially hazardous connections with power applied to any connected equipment.
- Never attempt to change 5B modules or open the lid with power applied to the WBK15. You could short out internally exposed circuits and cause damage.
- Never leave a foreign object (conductive or non-conductive) inside the WBK15.

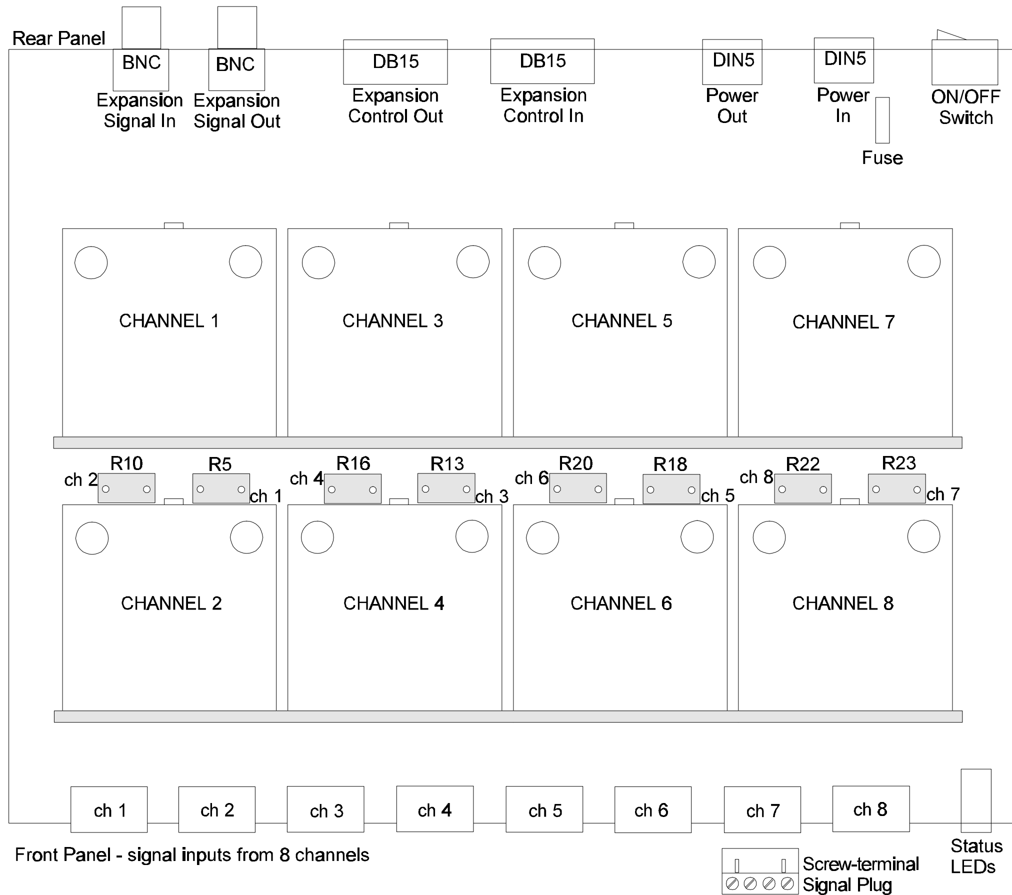
Power

Like the WaveBook/512, the WBK15 contains an internal power supply. The unit can be powered by an included AC power adapter or directly from any 10 to 30 VDC source, such as a 12 V car battery. For portable or field applications, the WBK15 and the WaveBook/512 can be powered by the DBK30A rechargeable battery module. The supply input is fully isolated from the measurement system. If the fuse requires replacement, it is a 2 A picofuse (Littlefuse #251002). **Note**:

- For custom power cabling, refer to the DIN5 connector’s pinout in Appendix E.
- For power requirements, refer to Appendix C. You must compute power consumption for your entire system and (if necessary) use auxiliary or high-current power supplies.

Configuration

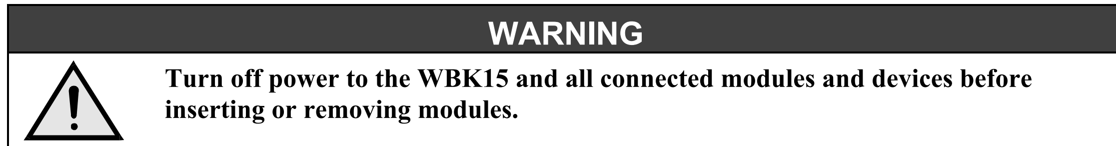
The figure shows the board layout within a WBK15. Note the channel number layout for the 5B modules and the location for plug-in current-sense resistors. **Note**: Only current-input type modules require the plug-in resistors. Such resistors must be removed for all other modules.



WBK15 Board Layout

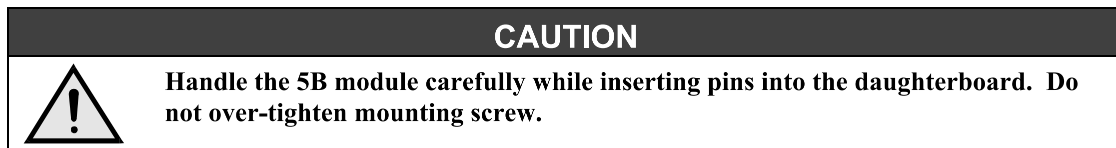
5B Module Insertion/Removal

The 5B modules plug into a daughterboard (×2) on the WBK15's motherboard. Rubber bumpers on one side and a tilted daughterboard allow the module to rest at a 5° angle to facilitate insertion and removal. The adjacent daughterboard has a cut-a-way to allow room for a screwdriver (see figure).



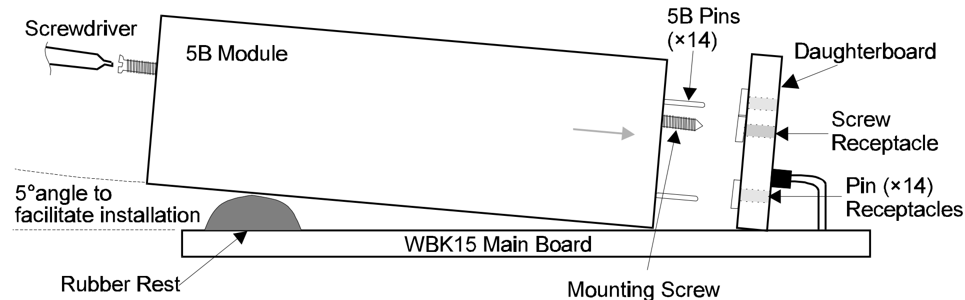
WARNING

Turn off power to the WBK15 and all connected modules and devices before inserting or removing modules.



CAUTION

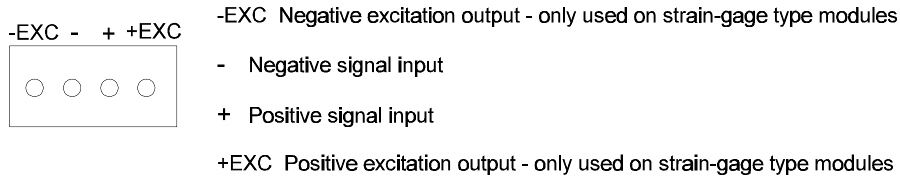
Handle the 5B module carefully while inserting pins into the daughterboard. Do not over-tighten mounting screw.



5B Module Insertion/Removal

Signal Connection

Signals are connected by screw-terminal signal plugs that plug into the 4-pin jacks on the WBK15's front panel (see figure).

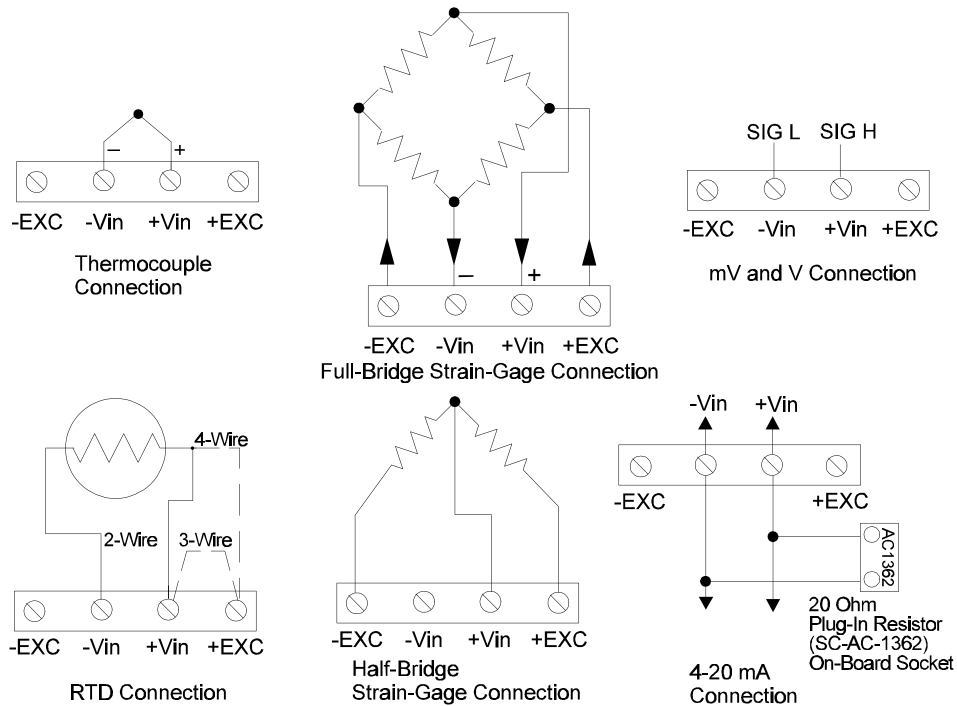


Signal Connection Jacks (per channel)

Input signals (and excitation leads) must be wired to the pluggable terminal blocks. Eight 4-terminal blocks accept up to 8 inputs.

WARNING
Shock Hazard. De-energize circuits connected to the WBK15 before changing the wiring or configuration.

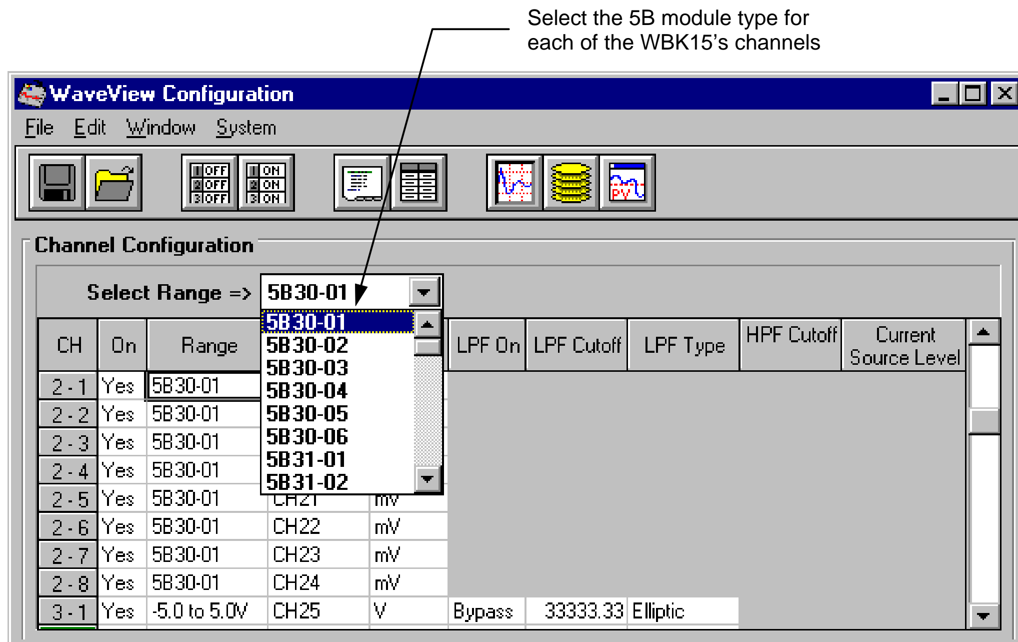
Terminal blocks are connected internally to their corresponding signal conditioning module. The terminal blocks accept up to 14-gage wire into quick-connect screw terminals. Each type of input signal or transducer (such as a thermocouple or strain gage) should be wired to its terminal block as shown in the figure below. Wiring is shown for RTDs, thermocouples, 20mA circuits, mV/V connections, and for full- and half-bridge strain gages.



Typical Signal Connections

Software Setup in WaveView

Depending on your application, you will need to set several software parameters so that WaveView will organize data to your requirements. The next figure shows the WaveView Configuration screen and calls out several parameters of importance to the WBK15. Ater the 5B module type is identified, WaveView figures out the *m* and *b* of *mx+b* for proper engineering units scaling. **Note**: if necessary, refer to chapter 5 *Using WaveView* for more information on using the columns to configure a channel.



Software Function

The Application Programming Interface does not contain functions specific to the WBK15. As needed, you can refer to related sections of chapters 6 through 12.

WBK15 - Specifications

Name/Function: WBK15 Multi-Purpose Isolated Signal Conditioning Module
Connector: 2 BNC connectors, mate with expansion signal input on the WaveBook/512; two 15-pin connectors, mate with expansion signal control on the WaveBook/512
Module Capacity: Eight 5B modules (optional)
Input Connections: Removable 4-terminal plugs
(Weidmuller type BL4, PN 12593.6 or type BLTOP4, PN 13360.6)
Power Requirements: 10 to 30 VDC or 120 VAC with included adapter
With 8 thermocouple-type modules: 12 VDC @ 0.25 A, 15 VDC @ 0.20 A, 18 VDC @ 0.2 A
With 8 strain-gage-type modules: 12 VDC @ 0.95 A, 15 VDC @ 0.75 A, 18 VDC @ 0.65 A
Cold-Junction Sensor: Standard per channel
Shunt-Resistor Socket: One per channel for current loop inputs
Isolation
Signal Inputs to System: 1500 VDC (600 VDC for CE compliance)
Input Channel-to-Channel: 1500 VDC (600 VDC for CE compliance)
Power Supply to System: 50 VDC
Dimensions: 221 mm × 285 mm × 36 mm (8.5” × 11” × 1.375”)
Weight: 1.8 kg (4 lb) with no modules installed

WBK20 - PCMCIA/EPP Interface Card

For full speed 1 M-sample/s portable applications, the optional WBK20 PCMCIA interface card cable can be used to link the WaveBook/512 to a notebook PC.

The WBK20 is simply inserted into a Type II PCMCIA socket (with the label side up) on the notebook PC. The cable provided is inserted into the end of the PCMCIA card, and the DB-25 socket connector is connected to the DB-25 plug connector on the WaveBook/512 (labeled "From PC Parallel Port"). No further hardware configuration is required; all configuration must be performed with the software which is provided with the WBK20.

Refer to the WBK20 installation manual for detailed information on installing this card.

WBK20 - Specifications

Name/Function: WBK20 PCMCIA/EPP Interface Card
Bus Interface: 8-bit PCMCIA Card Standard 2.1
Dimensions: 5 mm (PCMCIA Type II) card
Connector: DB25F
Transfer Rate: > 2 Mbytes/s
Cable: 2 ft (included)

WBK21 - ISA/EPP Interface Card

For full-speed (1 M-sample/s) desktop applications, the optional WBK21 ISA interface can be used to link the WaveBook/512 to a desktop PC.

The WBK21 installs into an IBM or compatible computer using any available 16-bit backplane slot. Prior to installing the card, check to make sure that the card is jumpered correctly for proper operation.

Refer to the WBK21 installation manual for detailed information on installing this card.

WBK21 - Specifications

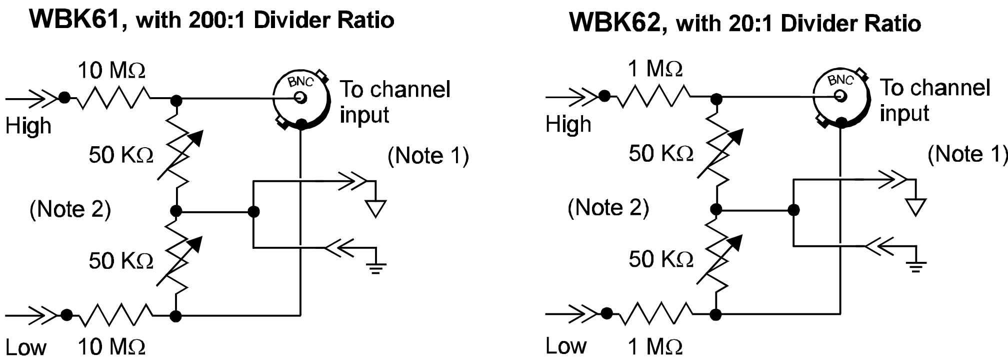
Name/Function: WBK21 ISA/EPP Interface Plug-in Board
Bus Interface: 16-bit ISA-bus interface
Transfer Rate: > 2.5 Mbytes/s
LPT Address: 378 or 278
LPT Interrupts: 5 or 7
Connector: DB25F
Serial-Port: high-speed 16C550 via DB9
Serial-Port Address: 3F8, 2F8, 3E8, or 2E8
Serial-Port Interrupt: 2, 3, 4, or 5
Connector: DB9M

WBK61 & WBK62 High-Voltage Adapters

The WBK61 and WBK62 are single-channel high-voltage adapters that can be used with the WaveBook/512 data acquisition system or WBK10 expansion module. In addition, the WBK61 and WBK62 can be used with the WBK11, WBK12, and WBK13 cards.

The WBK61 and WBK62 channel output connection is made through a BNC-to-BNC coupler. Each WBK61 and WBK62 high-voltage adapter has two signal input connections, one for low signal input, and the other for high signal input. The input channels are resistively isolated from ground, providing for safe connection of the test device. Series resistors, for both high and low channels, serve as attenuators, providing a maximum current limit of 100 µA.

The WBK61 and WBK62 include safety-style banana-jacks for the high and low inputs, and 60-inch (152 cm) cables with probe tips and alligator clips for easy input connection. The following figure shows simple diagrams for each high-voltage adapter.



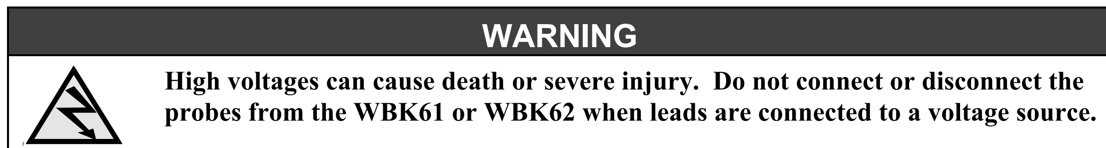
Note 1: Channel input connections are made (BNC-to-BNC) to the WaveBook/512 data acquisition system, or to the WBK10 analog expansion module. Refer to Hardware Setup in regard to analog ground and earth ground connections.

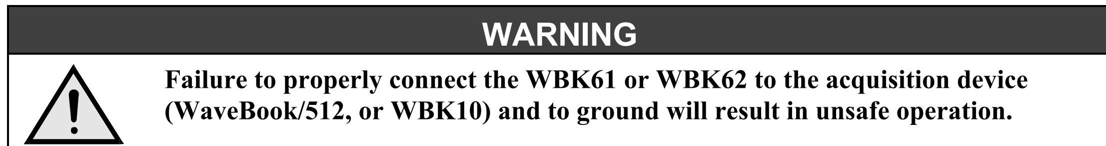
Note 2: The variable resistors are not user adjustable and are factory set at 50KΩ.

WBK61/62 Block Diagram

Hardware Setup

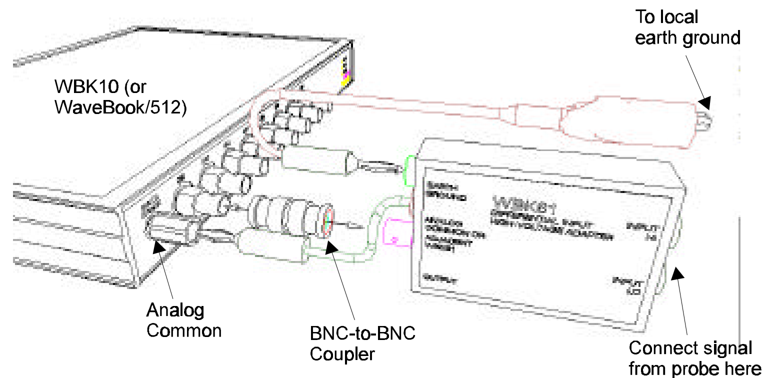
Refer to the following steps to connect the high voltage adapter. Refer to the figures on this and the following page as needed. Note that the term "**WBK6x**" refers to both the WBK61 and the WBK62 since the installation of these units is identical.

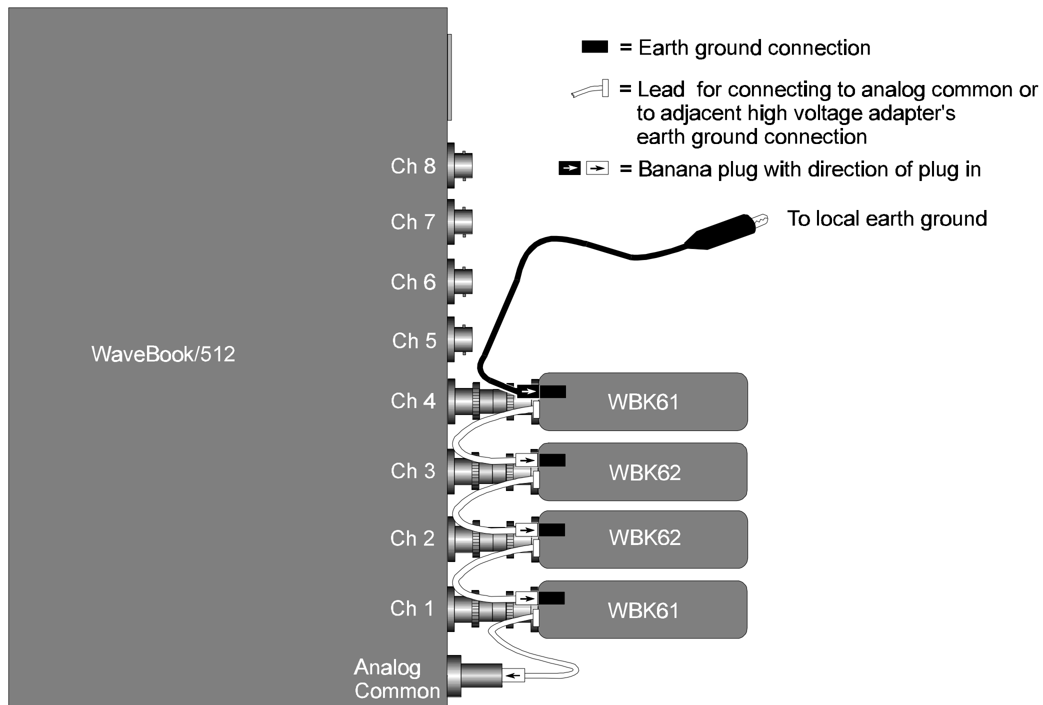




1. Connect the **WBK6x** (s) to any input channel(s) of the WaveBook/512 or WBK10 using the supplied BNC-to-BNC coupler (part # CN-110). Refer to the following two figures.
2. If connecting only one **WBK6x**, connect the green stacking banana plug to analog common (J12) on the WaveBook/512 or WBK10.

If connecting two or more **WBK6x**s, connect the green stacking plug of the first **WBK6x** to analog common (J12) on the WaveBook/512 or WBK10. Connect the other **WBK6x** stacking banana plugs to the adjacent **WBK6x** earth ground connections (see figure).





Example: Two WBK61 and two WBK62 High Voltage Adapters connected to a WaveBook/512

3. If connecting only one **WBK6x**, connect the green *banana plug/alligator clip* lead (part # CN-111) from the **WBK6x** earth ground connector to the local earth ground.
   If connecting two or more **WBK6x**s, connect the green *banana plug/alligator clip* lead (part # CN-111) from the *last* **WBK6x** earth ground connector to the local earth ground. Refer also to step 2 and the above figure.
4. Connect the input leads (part # CA-152) to the **WBK6x** Input HI and Input LO jacks.
5. Connect the test leads (part # CA-152) to the circuit under test. You may use alligator clips (part # CN-109) to connect test leads.

If desired, set the applicable WaveBook/512 channel(s) to the appropriate scale by setting the **mX+b** function in the WaveView program as discussed in the following section, *Software Setup*.

Software Setup in WaveView

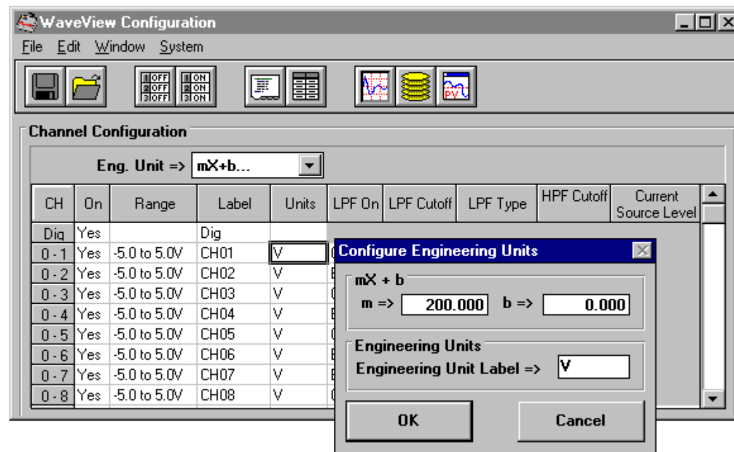
Note: Software which makes use of programs other than WaveView may require a similar setup.

Depending on your application, you may need to set software parameters to meet your requirements. The WaveView Configuration screen includes several important parameters including the "units." To configure the units, you must use the Configure Engineering Units pull-down box.

To set the **mX + b** equation so waveforms and data will be correctly scaled, enter the appropriate **m** value as follows:

- for WBK61: set m equal to 200.
- for WBK62: set m equal to 20.

Note: If necessary, refer to the *WaveBook/512 User s Manual* (chapter 5, *Using WaveView)* for more information on using the columns to configure a channel.



Software Function

The API (Application Programming Interface) does not contain functions specific to WBK61 or WBK62. Refer as needed to related sections of chapters 6 through 12.

WBK61/62 - Specifications

Name/Function:
WBK61, High-Voltage Adapter with Probes, 200:1 Voltage Divider
WBK62, High-Voltage Adapter with Probes, 20:1 Voltage Divider

Number of Channels: 1

Dimensions: 83 mm × 61 mm × 28 mm (3.25" × 2.375" × 1.1")

Cables: 60" leads with detachable probe tips and alligator clips

Output Connector: BNC socket

Voltage Divider:
WBK61: 200:1 fixed
WBK62: 20:1 fixed

Maximum Voltage
WBK61: 1000 Vpeak (on either input reference to earth ground)
WBK62: 100 Vpeak (on either input reference to earth ground)

Maximum Differential Voltage:
WBK61: 2000 Vpeak (if neither input exceeds 1000 Vp rating to earth ground)
WBK62: 200 Vpeak (if neither input exceeds 100 Vp rating to earth ground)

Frequency Characteristics: approximates a single-pole frequency response
-3 dB Bandwidth: 200 kHz minimum

Voltage Ranges: * Note: The asterisk indicates the range is obtained with the use of a WBK11, WBK12, or WBK13.

WBK61 Effective Ranges:
Unipolar: +1000V, 500V, 200V, 100V*, 40V*, 20V*
Bipolar: ±1000V, 500V, 200V, 100V, 50V*, 20V*, 10V*

WBK62 Effective Ranges: Note: For the WBK62 ranges which are followed by an asterisk, the WaveBook/512 (or WBK10) will exhibit superior performance with no WBK62 present.
Unipolar: +100V, 50V, 20V, 10V*, 4V*, 2V*
Bipolar: ±100V, 50V, 20V, 10V, 5V*, 2V*, 1V*

Measurement Errors:
The following values include total system error, i.e., they include errors from WaveBook/512, WBK10, WBK11, WBK12, and WBK13. The value for gain error does not include offset error.

Gain Error:
0.1% FS (unipolar)
0.2% FS (bipolar)

Offset Error:
0.1% FS (unipolar)
0.2% FS (bipolar)

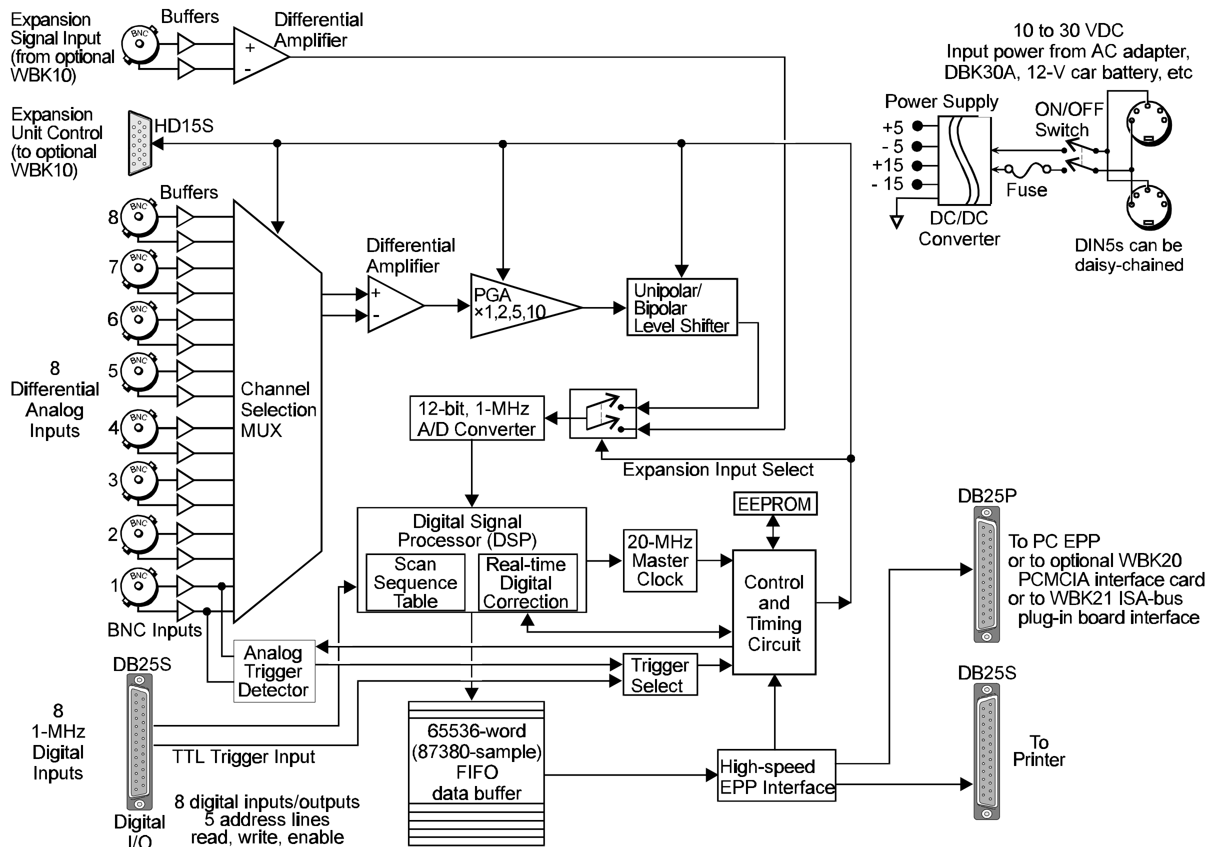
Operation Guide 4

This chapter describes WaveBook operation and related concepts of how data acquisition works. Understanding this material is important for effective use of the WaveBook whether using WaveView, other ready-to-run software, or custom-written software.

A block diagram of the WaveBook/512 base unit is explained. Later diagrams explain scan and triggering capabilities. If necessary, you may need to refer other chapters as follows:

- Chapter 1 outlines the system's basic features.
- Chapter 3 describes features of the WBK expansion options including block diagrams.
- Chapter 5 explains how to use WaveView.

System Overview



WaveBook/512 Block Diagram

Various features can be seen in the WaveBook/512's block diagram. The analog input signal path proceeds as follows:

- Each of the 8 pairs of differential signals (one per BNC connector) is buffered and then switched by the channel-selection multiplexer.
- The selected differential pair is then converted to a single-ended signal by the programmable gain amplifier (PGA), with a gain of ×1, ×2, ×5, or ×10, respectively corresponding to an input span of 10, 5, 2, or 1 volts.
- The amplified signal is then level-shifted to locate the desired range within the A/D converter's fixed input range. Two offset settings are available, unipolar and bipolar. The unipolar offset is used to sample signals that are always positive; the bipolar offset is used for signals which may be positive or negative. For example, when set for unipolar at a gain of ×5, the input span is 2 volts and the amplified signal is offset so that input voltages from 0 to +2 volts can be digitized. When set for bipolar operation, the offset is adjusted so that input voltages from -1.0 to +1.0 volts can be digitized.

- The signal is then switched into the A/D converter where it is digitized (to 12 bits) in 1 μs. The A/D converter's input can also be switched to the expansion signal input to read one of the 64 possible expansion channels, supplied by up to 8 WBK10 expansion chassis.
- The digitized value is then processed by the digital signal processor (DSP) which corrects the value for gain and offset errors. If this sample is to be delivered to the PC, then the DSP places the corrected result into the FIFO data buffer which holds the samples until the PC reads the data. If the sample is being used for triggering, then the DSP examines the result to determine if a valid trigger event has occurred.

The WaveBook also includes low-latency analog or TTL-level triggering. The low-latency analog trigger detector examines the WaveBook input channel 1 to determine if a trigger has occurred. The selected low-latency trigger is presented to the control and timing circuit that starts the acquisition after the trigger. The TTL trigger is taken directly from the digital I/O port.

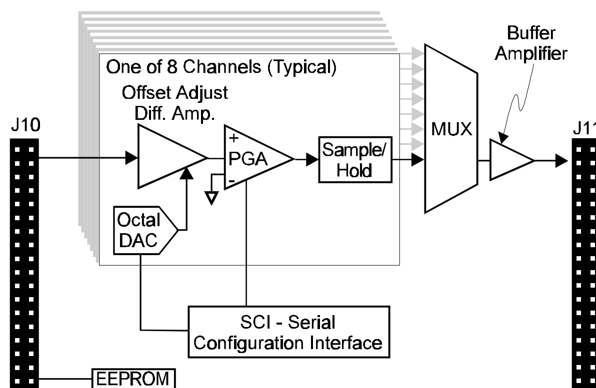
The control and timing circuit and the DSP together coordinate WaveBook activities. Every sample time, the DSP reads from the scan sequence table and accordingly programs the control and timing circuit for the next sample. The control and timing circuit waits precisely until the start of the next sample and then selects the input channel, the PGA gain, the level-shifter offset, and the A/D input source. It also conveys this information to any attached expansion units and precisely controls the A/D conversion timing.

The EEPROM holds the calibration information needed for the DSP-based real-time sample correction. Each WaveBook product (WaveBook/512, WBK10, and WBK11) includes such an EEPROM.

The digital I/O port is read and written by the DSP to transfer bytes of digital data. It may be used as a simple 8-bit input port or as a 32-address byte-wide I/O port.

The interface circuit connects the WaveBook and any attached printer to the PC. When the WaveBook is active, the interface holds the printer in a stable state; and when the WaveBook is inactive, the interface connects the PC to the printer.

Both the WaveBook/512 and the WBK10 can accept the WBK11 8-channel simultaneous-sample-and-hold card shown in the block diagram below. The SSH card includes an offset-adjusted differential amplifier, wide-range programmable gain amplifier (PGA), and sample/hold circuit for each of the 8 input channels. It also includes a buffered multiplexer that presents the sampled signal to the A/D and an EEPROM that holds the calibration constants. Unlike the WaveBook, the PGAs in the SSH cannot be dynamically configured during data acquisition. Their gain is fixed before samples are taken.

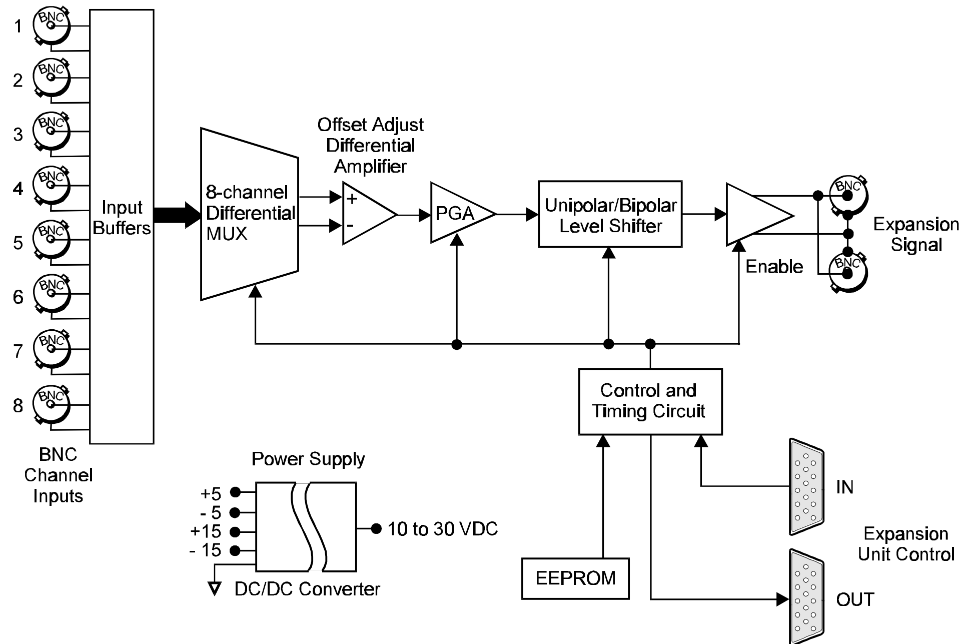


WBK11 Block Diagram

The PGAs in the SSH can each be set to one of seven gains: ×1, ×2, ×5, ×10, ×20, ×50, and ×100, respectively corresponding to input voltage spans of 10, 5, 2, 1, 0.5, 0.2, and 0.1 V. These input spans are offset for either bipolar or unipolar operation, just as in a WaveBook without an SSH.

Note that the level-shifter is still used with the SSH and the offset may still be adjusted on a sample-by-sample basis. Also, the analog trigger detector senses analog channel 1 after it has been amplified by the PGA. This improves the triggering resolution at high gains.

The WBK10 block diagram shown below is very similar to the analog portion of the WaveBook/512. Instead of digitizing the signals however, it sends them to the WaveBook through the expansion signal connectors. The expansion signal output can be disabled to allow multiple WBK10s to share a single expansion bus. The WBK10, like the WaveBook/512, can accommodate the installation of the WBK11.



WBK10 Block Diagram

The Acquisition Process

To take measurements with the WaveBook, several steps must be accomplished:

1. **Initializing the WaveBook** establishes communication between the PC, the WaveBook, and any attached options and automatically calibrates them according to the settings stored in their non-volatile EEPROM memory.
2. **Configuring an acquisition** is a multi-step process that includes specifying the signals to scan (and their voltage ranges), the trigger source, the number of scans to acquire, and the scan rate. These steps are described in more detail in the following sections.
3. **Starting the acquisition.** Once the acquisition has been configured, it must then be started. Samples will then be collected according to the configured trigger and acquisition modes. These samples must then be transferred into the PC for display, analysis, or storage.
4. **Stopping the acquisition.** Once all desired data has been collected, the acquisition must be stopped. It may be stopped automatically or may need to be stopped by a command.
5. **Shutting down the WaveBook.** Finally, when WaveBook operation is no longer needed, it may be shut down.

Programmer s note: The example programs supplied with the software use commands for a variety of acquisition types. These programs (explained in chapters 6 to 9), the theory of operation in this chapter, the enhanced API models chapter, and the command reference chapters are all useful for writing your own programs.

Initializing the WaveBook

The WaveBook is initialized with the `wbkInit` command which establishes communication with the WaveBook through a specified parallel port and interrupt. Before invoking `wbkInit`, it is often appropriate to use the `wbkSetErrHandler` and `wbkSetDefaultProtocol` commands. `wbkSetErrHandler` will catch errors that may occur during initialization. `wbkSetDefaultProtocol` will direct the software to use the best protocol, as identified by the `wbktest` program, when establishing communication.

One-Step Acquisitions

Instead of individually configuring and starting the acquisition, collecting the results, and stopping the acquisition, all of these steps can be performed with any of the one-step acquisition commands: `wbkRd`, `wbkRdScan`, `wbkRdN`, `wbkRdScanN`. These one-step commands greatly simplify the task of acquiring readings, but they lack the full flexibility of the individual, custom-acquisition commands.

Configuring an Acquisition

Channel Numbering

The analog input channels are identified by channel numbers which are determined by the connections among the WaveBook and the attached WBK expansion modules and are automatically assigned without switches or jumpers. The main unit's channels are always referred to as channels 1 through 8. The first expansion unit (connected directly to the WaveBook) has channels numbered 9 through 16. Subsequent expansion units add 8 more consecutive channels up to a total of 72 channels. **Note**: channel 0 always designates the main unit's 8-bit digital I/O port.

Specifying the Scan

Every acquisition is composed of one or more repetitions of a scan. A scan is a list of some or all of the input channels and their respective input ranges. Scans can vary in length from a single sample up to 128 samples. In general, the order in which channels are scanned is arbitrary—they need not be scanned in any particular order; channels may be duplicated within a scan, or they may be omitted. Each sample within the scan may be acquired at any range, specified by its gain and offset. Thus, the scan is configured by using a list of channels to be sampled and their gains and offsets. The `wbkSetScan` and `wbkSetMux` commands are used to configure the scan. **Note**: some minor limitations on the arbitrary arrangement of a scan are discussed in a later section.

Specifying the Trigger Source

The trigger is a recorded event to correlate the acquisition with other processes. The trigger may be used to start the acquisition or to serve as a marker within the acquisition. Every acquisition must be triggered. If there is no valid external trigger signal, the acquisition should be triggered by the PC.

The WaveBook supports several trigger sources: a software trigger from the PC, the channel 1 input level, the TTL-trigger input, or a combination of analog input samples. The `wbkSetTrigHardware` command is used to choose among the first three trigger sources. The `wbkSetTrigAnalog` and `wbkSetTrigComplex` are used to select the combination of analog input samples as the trigger source. This "multi-channel trigger" capability and others are described later in this chapter. The `wbkSoftTrig` command generates a software trigger. The software trigger takes effect regardless of what trigger source has been selected and so can be used to force a trigger if the desired trigger has not arrived.

Specifying the Number of Scans

During an acquisition, several groups of scans are acquired, depending on the selected acquisition mode. In the N-shot acquisition mode, a single, fixed-length group of scans is collected after the first trigger. In the N-shot with re-arm mode, a single, fixed-length group of scans collected after each trigger until the acquisition is stopped. In the infinite post-trigger acquisition mode, a single indefinitely long group of scans is collected after the trigger until the acquisition is stopped. In the pre/post-trigger mode, three groups of scans are collected: a fixed-length pre-arm group, an indefinitely long pre-trigger group, and a fixed-length post-trigger group.

The `wbkSetAcq` command specifies the acquisition mode and the length of the fixed-length groups. In the N-shot mode, the length specifies the number of scans to acquire after the trigger. In the infinite mode, no length need be specified—the acquisition continues until explicitly stopped. And in the pre/post-trigger mode, both the pre-arm and post-trigger lengths must be specified. The acquisition modes are further described later in this chapter. Note that in the pre/post-trigger mode, the pre-arm group length (also called the pre-trigger length) is the guaranteed number of scans that will be collected before the trigger.

Specifying the Scan Rate

During a scan, the WaveBook always takes samples every microsecond (1 μs). Thus, the minimum scan period (the fastest scan rate) is 1 μs times the number of channels. The maximum scan period (the slowest scan rate) is 100 seconds per scan. In the pre/post-trigger acquisition mode, two different scan rates may be specified, one for before the trigger and one for after.

The `wbkSetPeriod` and `wbkSetFreq` commands are used to set the scan rate(s) either in nanoseconds (ns) per scan, or scans per second, respectively. The scan rate may be set within its limits to 50 ns precision.

The `wbkGetMinPeriod` and `wbkGetMaxFreq` commands retrieve the minimum possible scan period or maximum possible scan rate based on the scan composition and trigger source. When multi-channel analog triggering is used, it reduces the maximum pre-trigger scan rate.

The `wbkGetPeriod` and `wbkGetFreq` commands retrieve the current actual settings of the scan period and rate. These may be different from those specified with `wbkSetPeriod` and `wbkSetFreq` because they include the effects of the trigger source, scan composition, and a 50-ns timebase resolution.

Starting the Acquisition

Before beginning an acquisition, the WaveBook must be configured for which channels to sample, how often to scan, how many scans to acquire, and the trigger parameters. The `wbkArm` command starts the acquisition. If the acquisition mode is pre/post-trigger, then the WaveBook will immediately begin acquiring the pre-trigger scans. Otherwise, the WaveBook will wait for the trigger before sampling. After the WaveBook begins taking samples, samples are placed in an output buffer from which they must be read by the PC.

Transferring Results

When samples are read from the WaveBook, they are placed into a user-supplied data buffer whose size is limited only by the PC's memory. The `wbkBufferTransfer` command informs the WaveBook about the size and location of this buffer. It also specifies if the buffer is circular or linear, and if the command should wait for the transfer to complete. As data is transferred into the buffer, the `wbkGetBackStat` command may be used to retrieve the amount of data that has been placed into the buffer and the state of the acquisition: active or inactive.

Circular buffers are used when the data is processed as it is being received, or if only the last buffer-full of data is of interest. As data is transferred into a circular buffer, if it becomes full, the data transfer continues at the beginning of the buffer, overwriting old data. The `wbkBufferRotate` command may be used to arrange a circular buffer into chronological order. Circular buffers are discussed in more detail later.

Linear buffers are not overwritten. If they become full during transfer, then the transfer stops until a new buffer is provided.

The `wbkBufferTransfer` command can wait until the transfer is stopped or can immediately return to the user's program. In the "background" non-waiting mode, the user's program continues during the transfer but must periodically check the status of the transfer. In the simpler "foreground" waiting mode, the user's program cannot perform other operations during the data transfer. Foreground and background modes are discussed in greater depth near the end of this chapter.

The acquired data may also be automatically written to disk with the `wbkSetDiskFile` command which must be invoked before the acquisition is started.

For ease of interpretation, the 12-bit samples are normally transferred as 16-bit words; but for increased efficiency, the data may be packed to transfer four samples in three 16-bit words. The `wbkSetDataPacking` command controls this packing, and the `wbkBufferUnpack` command may be used to unpack the received data.

Stopping the Acquisition

The `wbkDisarm` command immediately terminates data acquisition. The N-shot and pre/post-trigger acquisition modes automatically stop when they are complete; however, both the N-shot with re-arm and the infinite post-trigger acquisition modes will continue to collect data until they are explicitly stopped with the `wbkDisarm` command. The `wbkDisarm` command may also be used to stop N-shot and pre/post-trigger acquisitions before they complete.

Shutting Down the WaveBook

After all the acquisitions are complete and the WaveBook is no longer being used, it should be shut down with the `wbkClose` command. This helps to guarantee the proper operation of any printer attached through the WaveBook and assures that the hardware and software are in a consistent, stable state.

Operation Details

Analog Signal Processing

Each of up to 72 analog inputs are amplified and level-shifted to provide 8 input ranges (or 14 with the WBK11 SSH). The amplified signals are digitized to at least 11.98 bit resolution (4050 counts) with a 12-bit A/D converter and then corrected to 11-bit accuracy (not counting the sample-to-sample noise) by the DSP. The total system noise is less than 1-bit RMS at a gain of ×10.

The analog data format is a signed 16-bit number with -32768 (8000h) representing the minimum input voltage, 0 representing midscale, and 32752 (7FF0h) representing the maximum input voltage. The four least significant bits are not zero, but instead include fractional bit information from the digital calibration process. These bits are discarded, after rounding, if the samples are packed for transfer to the PC. If data packing is not used, then these bits are preserved.

Digital I/O Processing

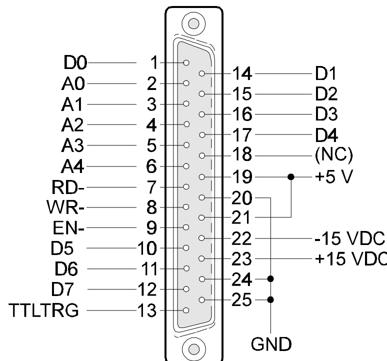
The WaveBook/512 digital I/O port writes and reads 8-bit bytes to and from 32 addresses. The pipelined, DSP-based design of the WaveBook restricts the digital I/O port to the following types of accesses:

- During Acquisition: Read a byte from the digital I/O port at address 0, as the first element in the scan.
- With Acquisition Stopped: Read and write bytes at any of the addresses under command of the host PC.

When read as the first element of a scan, the 8-bits that are read are left-justified within the 16-bit sample. The remaining 8 bits are undefined and may appear as any value. Thus a sample of the digital inputs is of the form 00XXh to FFXXh where the leftmost 8 bits are read from the port and the remaining (shown as XX) are undefined.

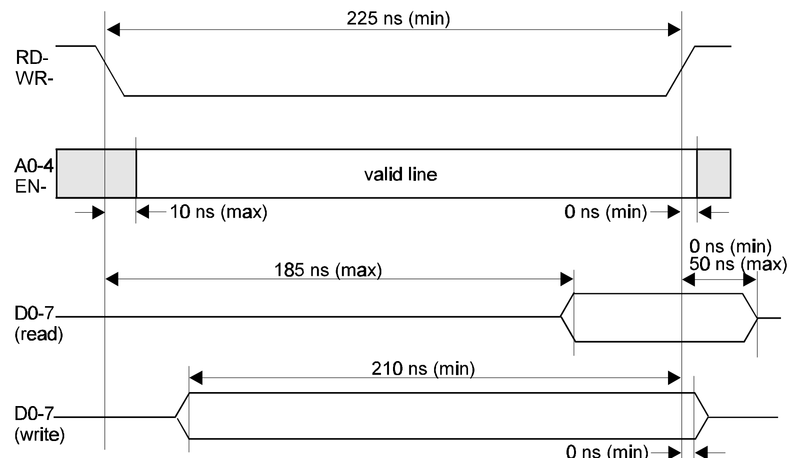
When the WaveBook is not acquiring data, then the `wbkDigRead` and `wbkDigWrite` commands may be used to read and write bytes at any one of the 32 addresses.

The pinout figure and table describe the digital I/O port signals.



Digital I/O Port Pin Use			
Signal Name	**Pin Number**	**Direction**	**Description**
D7 D6 D5 D4 D3 D2 D1 D0,	12 11 10 17 16 15 14 1	input/output	digital I/O port data lines
A4 A3 A2 A1 A0	6 5 4 3 2	output	digital I/O address lines
EN-	9	output	active-low digital I/O enable
RD-	7	output	active-low read strobe
WR-	8	output	active-low write strobe
TTLTRIG	13	input	TTL-trigger input
+5 volts	19, 21	power output	250 mA maximum total
+15 volts	23	power output	50 mA maximum
-15 volts	22	power output	50 mA maximum
Ground	20, 24, 25		

If the digital I/O port is only being used to sample the digital inputs during acquisitions, just connect the digital signals to the digital I/O port data lines. If the digital I/O port is to be used for addressed I/O, using the `wbkDigRead` and `wbkDigWrite` commands, then the digital signals must conform to the timing requirements shown in the digital I/O port-timing diagram shown below:



Digital I/O Port Timing

During a read from the digital I/O port, the RD- signal is asserted for at least 225 ns. Within 10 ns after RD- is asserted, the address and enable lines are valid. They remain valid until at least 0 ns after RD- is unasserted. The data to be read must be presented to the D0-7 data lines within 185 ns after RD- is asserted and must be held until after RD- is unasserted. If the digital I/O port is used for both input and output, then the data lines must not be driven for more than 50 ns after RD- is unasserted; but if the digital I/O port is only used for input, then D0-7 may be driven indefinitely.

During a write to the digital I/O port, the WR- signal is asserted for at least 225 ns. Within 10 ns after WR- is asserted, the address and enable lines are valid. They remain valid until at least 0 ns after WR- is unasserted. The data to be written is driven on the D0-7 data lines at least 210 ns before WR- is unasserted and remains on the data lines at least until 0 ns after WR- is unasserted.

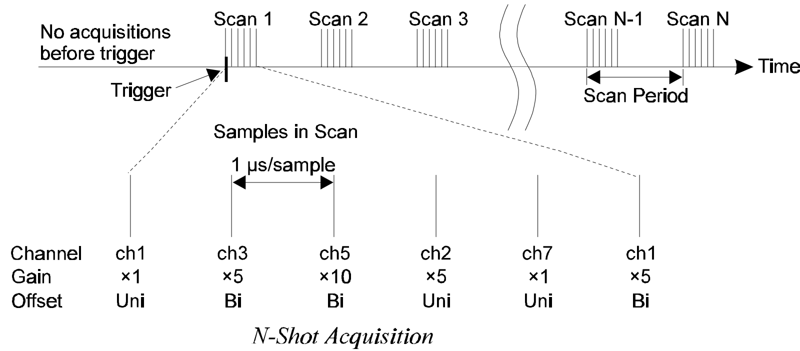
Acquisition Composition

Acquisition Mode and the Number of Scans

The acquisition mode determines the number of scans (and their relation to the trigger) that make up an acquisition. There are 4 acquisition modes: N-shot, Infinite post-trigger, N-shot with re-arm, and Pre/post-trigger.

N-Shot Acquisition Mode

The N-shot mode, shown below, is the simplest. After the trigger, it collects a fixed number of scans, from 1 to 100,000,000.

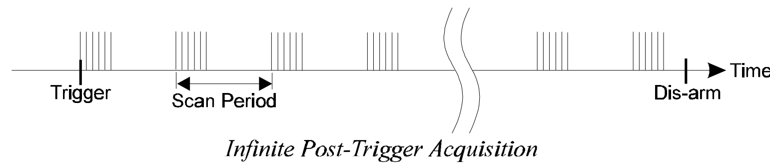


N-Shot Acquisition

Once the acquisition has been started, with the `wbkArm` command, the N-shot mode waits until a trigger occurs. It collects the post-trigger number of scans, as specified by the `wbkSetAcq` command, and then stops. No more data will be collected until another acquisition has been started. As shown in the diagram, each scan consists of one or more samples, taken 1 microsecond apart. The scans are repeated with the post-trigger scan period that was specified with the `wbkSetPeriod` or `wbkSetFreq` command. The number of scans collected shown as "N" in the diagram may be set from a minimum of one scan after the trigger to a maximum of 100,000,000 scans.

Infinite Post-Trigger Acquisition Mode

It is also possible to collect an unlimited number of scans after the trigger by using the infinite post-trigger acquisition mode as shown below:

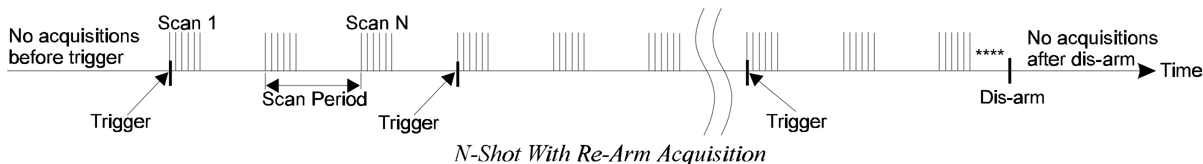


Infinite Post-Trigger Acquisition

This mode is identical to the N-shot mode except that the total number of scans is not limited. In the infinite post-trigger mode, scans are continually taken at the post-trigger scan rate. When the desired number of scans have been gathered, the `wbkDisarm` command must be used to terminate acquisition.

N-Shot With Re-Arm Acquisition Mode

The N-shot with re-arm acquisition mode, shown below, is used to collect scans after each trigger.

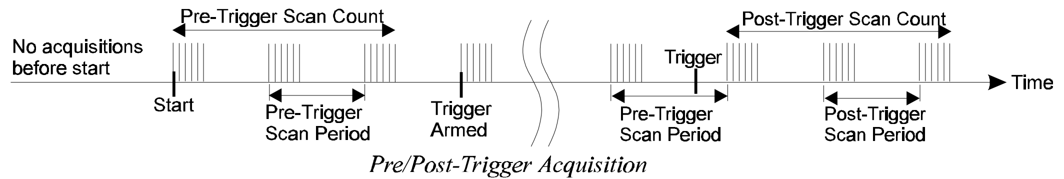


N-Shot With Re-Arm Acquisition

This mode is also similar to the N-shot mode. It differs in that after the "N" scans have been acquired, the trigger is re-armed so that another "N" scans will be taken for each subsequent trigger. As in the infinite post-trigger mode, the `wbkDisarm` command must be invoked to prevent further acquisitions.

Pre/Post-Trigger Acquisition Mode

The pre/post-trigger mode is substantially different from the preceding modes. This mode acquires samples both before and after the trigger (the other modes only take samples **after** the trigger).



Pre/Post-Trigger Acquisition

In this mode, as soon as the acquisition is started, the WaveBook immediately begins taking scans with the trigger disabled. The WaveBook guarantees that at least a certain number of scans are collected before the trigger is recognized. This number is referred to as the "pre-trigger scan-count". When the pre-trigger scan-count number of scans have been collected, the WaveBook then arms the trigger. The WaveBook continues scanning at the pre-trigger scan rate until the trigger occurs. When the trigger does occur, the WaveBook completes the current pre-trigger scan and then begins scanning at the post-trigger scan rate until the post-trigger number of scans has been collected.

The collected scans can be divided into three groups: the pre-arm scans, the post-arm scans, and the post-trigger scans. The pre-arm and post-trigger groups are each individually set to a fixed length from 1 to 100,000,000 scans. The post-arm group (the scans after the WaveBook is armed and before the trigger is detected) may contain any number of scans, depending on how long it takes before the trigger. Thus, the minimum number of scans collected in this mode is just the sum of the pre-trigger and post-trigger scan counts; but the maximum number is unlimited. Circular data buffers are often used to capture this type of data.

Scan Composition

As shown above, every acquisition collects one or more identical scans of the input signals. Each scan consists of from 1 to 128 samples collected at a rate of 1 microsecond per sample. Each sample of a scan specifies the channel to be acquired and the gain and offset to be applied to the signal. The `wbkSetMux` and `wbkSetScan` commands are used to specify the scan composition.

In general, the channel number, gain, and offset may be specified arbitrarily for each sample in the scan, but there are some restrictions:

- The digital input port (channel 0), if read, must be the first channel of the scan.
- The first channel of the scan may not be a channel equipped with a WBK11 (simultaneous sample/hold) option. If an SSH channel is specified as the first channel of the scan, then the software will automatically insert a dummy channel, whose result is discarded, as the actual first channel of the scan, thus reducing the maximum scan length to 127 samples.
- Every reading of a given SSH channel must be at the same gain because the WBK11 cannot change gain between multiple readings of the same channel. Different SSH channels may be sampled with different gains, and a single SSH channel may be read with both unipolar and bipolar offsets, but all samples of a single SSH channel must be made at the same gain.

Scan Period

The scan period is the time from the first channel of one scan until the first channel of the next scan. The WaveBook/512 supports two different scan periods: the pre-trigger scan period, and the post-trigger scan period. Because the WaveBook/512 samples at a rate of 1 MHz, the absolute minimum scan period is just 1 μs times the number of samples in the scan. However, the actual minimum period may be more:

- If the first channel in the scan is a SSH channel (with a WBK11 installed), then a "dummy" channel is added to the scan adding 1 μs to the minimum scan period.
- Multi-channel triggering increases the minimum **pre**-trigger scan period by 1 μs plus 1 μs for each trigger sample. However, multi-channel triggering does not affect the minimum **post**-trigger scan period.

The actual scan period may be any value from the minimum scan period to a maximum of 100 seconds, and can be set with 50 ns (0.05 μs) resolution. **Note**: the pre-trigger and post-trigger scan periods may each be set to any valid value from their respective minimum (which differs if multi-channel triggering is used) to their maximum of 100 seconds.

Triggering Capabilities

Triggering is the use of an external signal or signals to start or synchronize the data acquisition process. The WaveBook can use **four types of trigger signals**:

- **TTL-level** input (either rising- or falling-edge). TTL-level triggering is performed by digital logic connected to the digital expansion connector.
- **Channel 1 analog** signal level (rising or falling). Analog triggering is performed by comparator-based analog hardware connected directly to analog input channel 1.
- **Multi-channel** combination of measured channel values. Multi-channel triggering is performed by the WaveBook's Digital Signal Processor (DSP). The DSP samples the specified channels; and, if programmable conditions are met, a trigger is generated. Multi-channel triggering examines digitized data, and the trigger latencies are much greater.
- **Software** command from the PC. The software trigger allows the PC to generate a trigger without waiting for an external event. This may be used to immediately begin a data acquisition, or to force an acquisition to occur if the expected trigger did not occur.

Low-latency (Hardware) Triggering

The TTL-trigger input and the channel 1 analog signal comparator output are each connected directly to hardware trigger circuits to provide low-latency triggering. The WaveBook can respond to a TTL or analog trigger with a jitter (uncertainty in latency) of no more than 100 nanoseconds.

- Unless pre-trigger data is being collected, the WaveBook responds to the trigger with a latency of less than 200 ns for TTL and 300 ns for analog.
- If pre-trigger data is being collected, then triggers are not acted upon until the end of the current pre-trigger scan. This increases the trigger latency and jitter but preserves the specified scan rates.

When the analog channel 1 trigger is used, the channel 1 analog input signal is compared with a programmable voltage level to generate an internal TTL-level signal which is **true** if the analog input is greater than the programmable voltage level (and **false** if it is less). When the TTL trigger is used, then the TTL-level trigger signal from the digital I/O connector is used directly. The resulting TTL-level signal is then, under program control, examined for either a false-to-true (rising edge), or true-to-false (falling edge) transition which, when it occurs, is the trigger event.

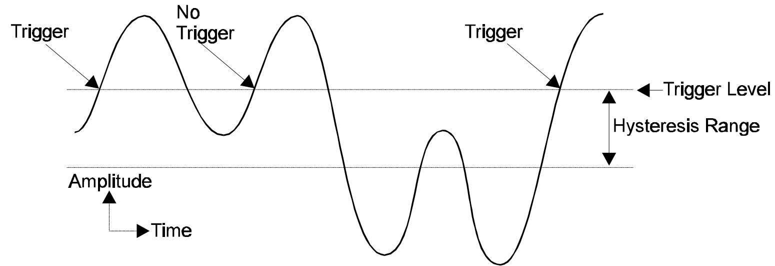
If the system is ready for a trigger, then the trigger event will be acted upon. If the system is not ready, because it is not completely configured, or because it is still finishing the previous trigger's action, the trigger will be ignored. No indication of such an ignored trigger is given.

The low-latency analog trigger compares the analog signal with a programmable voltage source. The effective range of this voltage source depends on whether or not the WBK11 SSH option is installed.

- Without SSH, the trigger threshold is settable from -5.0 to +9.996 volts with 12-bit (3.7 mV) resolution, regardless of any channel's gain settings. This gives better than 1% resolution at even the smallest input ranges (±0.5 or 0-1 volts).
- With SSH, the analog channel 1 signal is first amplified by the SSH programmable gain amplifier before being compared with the programmable voltage. This allows precise trigger-level adjustment, even at high gain levels. The analog-trigger comparator threshold-voltage range and resolution (with SSH) are shown in the following table.

SSH Input Range	Trigger Threshold Range	Resolution (mV)
0-10 or ±5	-5.0 to 9.996	3.66
0-5 or ±2.5	-2.5 to 4.998	1.83
0-2 or ±1	-1.0 to 1.999	0.73
0-1 or ±0.5	-0.5 to 0.9996	0.366
0-0.5 or ±0.25	-0.25 to 0.4998	0.183
0-0.2 or ±0.1	-0.10 to 0.1999	0.073
0-0.1 or ±0.05	-0.05 to 0.09996	0.0366

The analog trigger circuit has hysteresis which reduces the occurrence of retriggering due to input noise. The hysteresis without SSH is 25 mV; the hysteresis with SSH is 1/600 of the comparator range. The effect of this hysteresis (for rising-edge trigger) is shown in the next figure.



Hysteresis Effect on a Rising-Edge Trigger

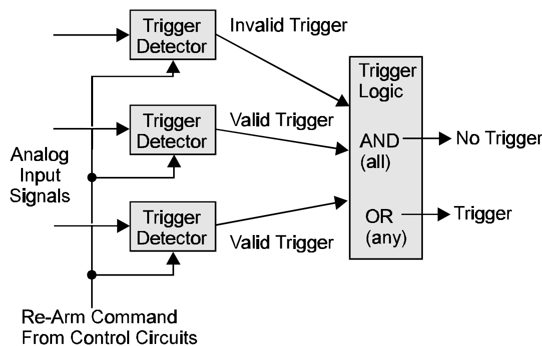
A trigger will occur when the analog input rises above the trigger level, but only after the input level has been below the hysteresis range. If the level momentarily drops just below the trigger level (perhaps due to noise) and then rises above it again, no extra triggers will be generated—the signal did not drop below the hysteresis range. After the level drops below hysteresis, it can then again produce a trigger by rising above the trigger level.

DSP-Based, Multi-Channel Triggering

When the small, hardware-limited latencies of the TTL and analog channel 1 triggers are not required, the DSP chip may be used to examine the samples from one or more channels and decide if they constitute a pre-defined trigger condition.

The DSP can sample up to 72 of the input channels and examine each one to determine if they meet programmed levels for a valid trigger. This multi-channel triggering is a two-step process:

1. The WaveBook DSP examines each of its specified input signals to determine if they are valid triggers.
2. After all of the channels have been examined, the DSP logically combines the individual triggers to generate the actual trigger. The DSP may be programmed to generate a trigger if **any** individual trigger is valid ("OR") or only if **all** triggers are valid ("AND")—see figure.



Multi-Channel Trigger Detection

Trigger validity in a multi-channel environment is determined by the logical relationship among 3 elements (slope, duration, and initialization) as discussed in the next section.

Eight Trigger Types

The first step in multi-channel triggering is to examine the input signals. The WaveBook can examine each input in 1 of 8 ways to determine trigger validity. Each of the 8 trigger types is a combination of 3 elements: slope, duration, and initialization.

Trigger Type	Slope	Duration	Initialization
Above-level	Rising	Instantaneous	Level
Below-level	Falling	Instantaneous	Level
Above-level-with-latch	Rising	Latched	Level
Below-level-with-latch	Falling	Latched	Level
Rising-edge	Rising	Instantaneous	Edge
Falling-edge	Falling	Instantaneous	Edge
Rising-edge-with-latch	Rising	Latched	Edge
Falling-edge-with-latch	Falling	Latched	Edge

Slope (above/rising or below/falling) sets whether the trigger is valid when the signal is:

- above the trigger level (**rising**)
- below the trigger level (**falling**)

Duration (instantaneous or latched) specifies the action to take if the signal level becomes invalid after it has been valid.

- **Instantaneous** triggers become invalid as soon as the signal does. Instantaneous triggers are used to trigger on the *coincidence of signals*: when two or more signals are *simultaneously valid*.
- **Latched** triggers remain valid until the acquisition is complete. Latched triggers are used to trigger on the *occurrence of signals*: when two or more signals *have already become valid*.

The trigger duration only makes a difference in multi-channel "AND" triggering. In multi-channel "OR" triggering, the WaveBook will be triggered as soon as any channel becomes valid; what happens when a channel becomes invalid does not matter. In contrast, "AND" triggering waits for all of the triggers to be valid; and so, latching can be important for rapidly changing signals.

Initialization (level or edge) specifies the sequence necessary for a signal to be a valid trigger.

- **Level** triggers become valid as soon as they reach or exceed the trigger level, even if they are already past the trigger level when the acquisition is started.
- **Edge** triggers first wait until the signal level is invalid. Then they wait for the signal to reach the trigger level before becoming valid.

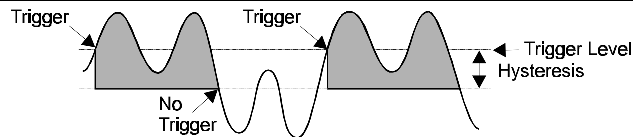
Thus, level triggers look for a signal level, whenever it occurs; and edge triggers look for a rising or falling transition that reaches the trigger level.

Examination of the input signals compares two specified signal levels: the trigger level and the hysteresis.

- The **trigger level** determines when the input channel is a valid trigger.
- The **hysteresis** is the amount by which the channel must differ from the trigger level for the channel to become invalid.

Above-Level Trigger

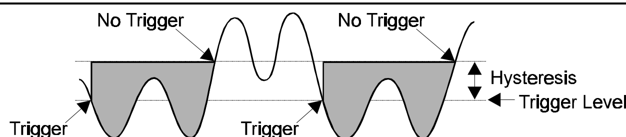
Rising slope
Instantaneous duration
Level initialization



This trigger is valid whenever the signal level is above the trigger level and stays valid until the signal level goes below the hysteresis range. In the figure, the channel trigger is valid during the 2 shaded intervals. Whether this triggers the WaveBook depends on the type of multi-channel triggering ("AND" or "OR") and on the state of the other trigger channels. If multi-channel triggering is configured for "OR", then the WaveBook will be triggered when the signal first rises above the trigger level; if the WaveBook is ready for a new trigger, then it will also be triggered the second time the signal rises above the trigger level. If multi-level triggering is configured for "AND", then the WaveBook will not be triggered until every specified trigger channel is valid. If all other trigger channels are valid, the WaveBook will be triggered when the signal reaches the shaded region; but if some channels are not valid, this channel will have no effect.

Below-Level Trigger

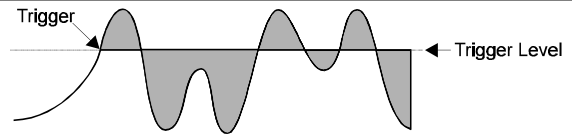
Falling slope
Instantaneous duration
Level initialization



This trigger is valid whenever the signal level is below the trigger level and stays valid until the signal level goes above the hysteresis range (the reverse of above-level triggering). As with all multi-channel trigger types, the WaveBook's actual trigger depends on the combination of this trigger with the other channels' trigger states.

Above-Level-With-Latch Trigger

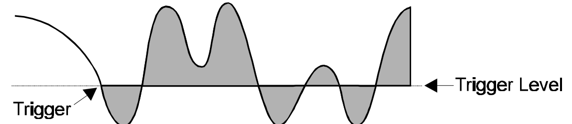
Rising slope
Latched duration
Level initialization



In this type of triggering, the channel becomes valid when the signal level rises above the trigger level and stays valid until the acquisition is complete and the WaveBook is re-armed.

Below-Level-With-Latch Trigger

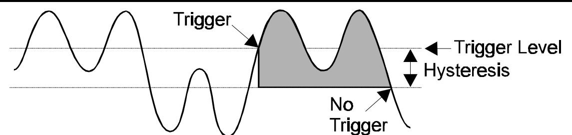
Falling slope
Latched duration
Level initialization



The channel becomes valid when the signal level rises above the trigger level and stays valid until the acquisition is complete and the WaveBook is re-armed (the reverse of above-level-with-latch triggering). Latched triggers are often used in multi-channel "AND" triggering, where the WaveBook is not triggered until all trigger channels are valid. After a latched trigger becomes valid, it stays valid (waiting for the other triggers to become valid) until the WaveBook is triggered and the acquisition completes. If the trigger is non-latched instead latched, the channel may not stay valid and the WaveBook will not trigger until the channel becomes valid again and all channels simultaneously reach their trigger levels. In other words, **latched triggering is used to trigger after something has occurred, but non-latched triggering is used only during the simultaneous occurrence of desired signal levels.** It is possible to combine different trigger types in a single multi-channel trigger. For example, the WaveBook could trigger when channel 3 is below 0.9 volts after channel 2 has gone above -1.3 volts, by configuring channel 3 for below-level triggering and channel 2 for above-level-with-latch triggering.

Rising-Edge Trigger

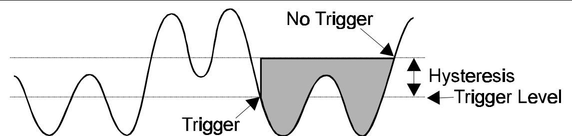
Rising slope
Instantaneous duration
Edge initialization



This trigger becomes valid after the signal level has been below the hysteresis range and then goes above the trigger level. This trigger becomes invalid when the signal level goes below the hysteresis range. Unlike above-level triggering, the channel cannot become valid until the signal level first goes below the hysteresis range. This prevents the false triggering that would occur if the signal is above the trigger level at the start of the acquisition.

Falling-Edge Trigger

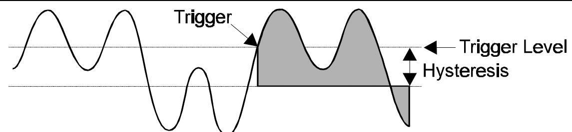
Falling slope
Instantaneous duration
Edge initialization



This trigger is the reverse of the rising-edge trigger: the trigger becomes valid after the signal level has been above the hysteresis range and then goes below the trigger level. This trigger becomes invalid whenever the signal level goes above the hysteresis range. This prevents the false triggering that would occur with below-level triggering if the signal was below the trigger level at the start of the acquisition.

Rising-Edge-With-Latch Trigger

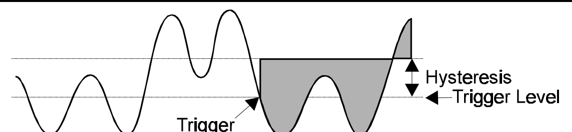
Rising slope
Latched duration
Edge initialization



This trigger becomes valid like a rising-edge trigger: when the signal level goes above the trigger level after first being below the trigger range. However, the rising-edge-*with-latch* trigger does not become invalid, regardless of the signal level, until the acquisition is complete. Rising-edge-with-latch is used to trigger after the channel has reached the trigger level, rather than just while the channel is above the trigger level.

Falling-Edge-With-Latch Trigger

Falling slope
Latched duration
Edge initialization



This trigger is the reverse of the rising-edge-with-latch trigger: the trigger becomes valid after the signal level has been above the hysteresis range and then goes below the trigger level. The trigger remains valid until the acquisition is complete.

Trigger Latency and Jitter

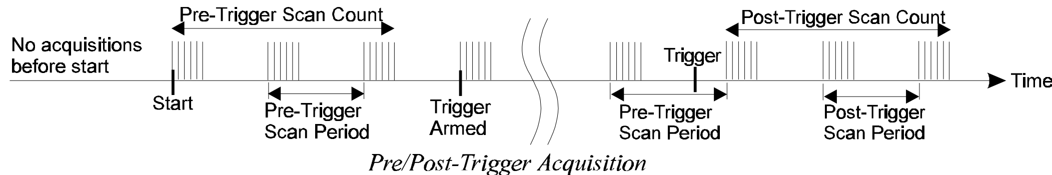
- Trigger **latency** is the time duration between the occurrence of the valid trigger and the start of the triggered acquisition of the data.
- Trigger **jitter** is the variation of the latency: the amount of time the latency can vary from trigger to trigger.

Latency and jitter depend on both the trigger source and the acquisition mode.

As discussed above, the WaveBook has different acquisition modes. Post-trigger modes (N-shot, N-shot with re-arm, and infinite-post-trigger) collect scans only after the trigger has occurred. They are different from the pre/post-trigger mode which collects scans both before and after the trigger. This difference affects the trigger latency and jitter.

In a post-trigger mode, the WaveBook is not scanning while waiting for the trigger. Thus, it is free to respond to the trigger as soon as it occurs. This minimizes the latency and jitter.

In the pre/post-trigger mode, pre-trigger data is being collected while the WaveBook waits for the trigger, and the WaveBook will not respond to a trigger until the current scan is complete. The first scan after the trigger is separated from the last scan before the trigger by the pre-trigger scan period. Thus, all the scans up through the one immediately following the trigger are collected at the pre-trigger rate, and all the subsequent scans are collected at the post-trigger rate. This preserves the integrity of the acquisition timebase as shown in the figure below:



Pre/Post-Trigger Acquisition

The time needed to complete the final pre-trigger scan is part of the trigger latency; and so, in the pre/post-trigger mode, the trigger latency may be greatly increased.

Not only do the trigger latency and jitter depend on the pre- vs post-trigger type of acquisition, they also depend on the trigger source: software, TTL, analog, or multi-channel. The following table gives the latency and jitter for each of the different trigger sources and acquisition modes.

Acquisition Type	Trigger Source	Max. Trigger Latency	Trigger Jitter	Notes
Pre-trigger	Software	100 µs + T	100 µs + T	a, c
	TTL	200 ns + T	50 ns + T	c
	Analog (channel 1)	300 ns + T	50 ns + T	c
	Multi-channel	2 * T - NS µs	T	c, d
N-Shot, N-Shot w/ re-arm, or infinite-post-trigger	Software	100 µs	100 µs	a
	TTL	200 ns	50 ns	
	Analog (channel 1)	300 ns	50 ns	
	Multi-channel	2 * NC + 3 µs	NC +2 µs	b

(a) Software trigger latency and jitter depend greatly on the host computer's speed, operating system, and printer-port protocol. Most systems should take much less than 100 µs.
(b) NC is the number of samples used for multi-channel triggering, from 1 to 72, as specified by the trigger configuration.
(c) T is the pre-trigger scan period.
(d) NS is the number of samples in a scan including, if present, the first "dummy" sample, from 1 to 128.

Data Packing

Normally, the WaveBook/512 transfers each of its 12-bit readings as a 16-bit word which can be easily interpreted by the PC. However, to increase the data transfer efficiency by 25%, 4 readings can be packed into 3 words.

Advantages of Packed Data:

- Less Data to Transfer: By packing 4 samples (12-bit) into 3 words (16-bit), the amount of data transferred from the WaveBook/512 to the PC is reduced 25%, thus reducing the amount of time required to transfer the data by a similar amount. This can increase the maximum data acquisition rate. For example, if the PC's EPP port is limited to 800 KB/sec, then without packing, the maximum average data acquisition rate will be 400 Ksamples/sec, but with packing it will increase to 533 Ksamples/sec.
- Less Data to Store: As packed data takes fewer words, it effectively increases the buffer size. For example, the 65536-word internal data buffer in the WaveBook/512 can hold 87380 samples of packed data instead of only 65536 samples of unpacked data. Similarly, the PC's memory is used more efficiently when storing packed data.
- Less Data to Archive: Storing packed data to disk is both faster (because there are fewer words to transfer) and more economical (the disk files are smaller) than storing unpacked data.

Disadvantages of Packed Data:

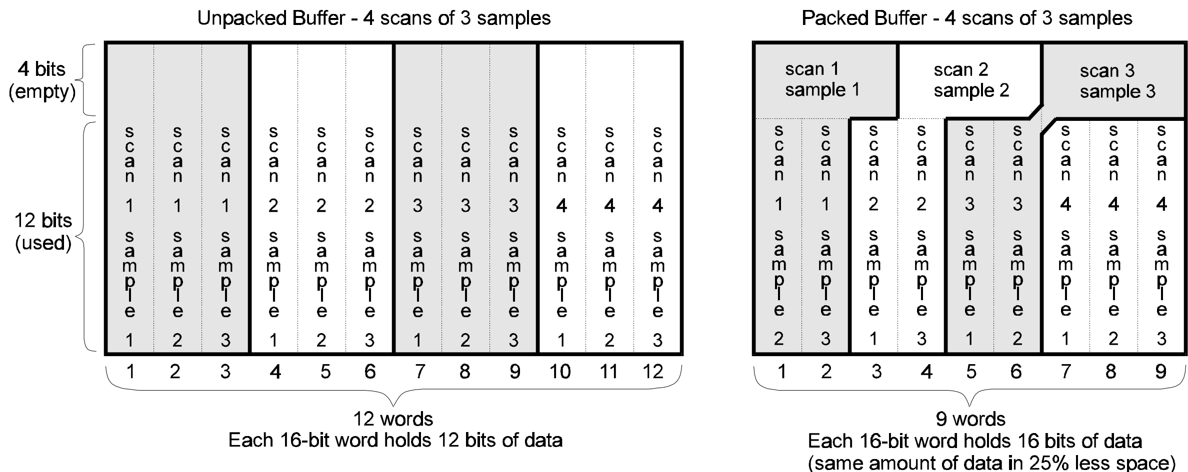
- Extra Processing Steps: The packed data must be unpacked before it can be interpreted, increasing the complexity of programs which use packed data.
- Loss of Resolution: Even though the WaveBook/512 is a 12-bit device, it uses 16-bit arithmetic internally to compensate the digitized data. Rounding these results to 12 bits, so they can be packed, slightly reduces the accuracy (by less than ½LSB).
- Special Cases: If an acquisition is not a multiple of 4 samples long, then the last, 3-word, packed data group will only be partially filled with the final sample(s). These end conditions must be taken into account when directly processing packed data.

Packed-Data Format

The packed-data format packs four 12-bit samples into three 16-bit words. To do so, the first sample is broken into three 4-bit nibbles, H, M, and L, where H is the most-significant (high) nibble, M is the middle nibble, and L is the least-significant (low) nibble. These 3 nibbles are then combined with the three subsequent samples to form three 16-bit words (see table). The `wbkSetDataPacking` function is used to enable/disable data packing.

	Bits	
Word	**15-12**	**11-0**
1	1L	Sample 2
2	1M	Sample 3
3	1H	Sample 4

The next figure shows how 4 scans of 3 samples each are transferred to the PC. If data packing is not used (1st diagram), each of the 12 samples occupies a separate 16-bit word of memory. If data packing is used (2nd diagram), the samples are packed into 9 words without regard for the scan boundaries. Each group of 3 words includes 4 consecutive samples which may come from more than one scan.



Data Packing

Data Transfer

Once an acquisition has been started, the WaveBook/512 will begin acquiring samples either immediately (if pre-trigger is enabled) or after the trigger. These samples are initially placed into the WaveBook's internal 64K-word first-in-first-out (FIFO) buffer. The PC must retrieve the samples from the FIFO soon after acquisition starts; otherwise, the FIFO will become full, samples will be lost, and an error will be indicated. The retrieved samples are transferred into data buffers in the PC's memory.

Time-outs

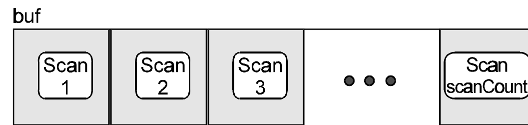
While the PC is controlling and transferring data from the WaveBook, the application program in the PC is unable to execute and is temporarily suspended. To keep the application program from hanging indefinitely, the WaveBook includes a timer which will stop WaveBook operation with an error if the application program has been suspended for more than a specified number of milliseconds. The time-out interval is set with `wbkSetTimeout`.

Buffer Size

The `wbkBufferTransfer` command retrieves the samples from the WaveBook into a buffer in the PC's memory. `wbkBufferTransfer` takes many arguments which are described in full in the command reference chapter. Among those are the ones that describe the buffer:

- `buf` - the address of the buffer that will hold the samples
- `scanCount` - the size of the buffer, in scans
- `cycle` - controls overwriting of old samples

The buffer must have room for at least `scanCount` scans. If data packing is not used, then each sample takes one 16-bit integer, and the buffer must be at least (samples-per-scan)*scanCount words long (see figure).



With data-packing, every 4 samples take 3 integers and the buffer must be at least (3/4)*(samples-per-scan)*scanCount words long. Also with data-packing enabled, the buffer must hold a whole number of 4-sample groups of packed data—the number of samples in the buffer, (samples-per-scan)*scanCount, must be a multiple of 4. See table for some examples of buffer sizes:

Scan Length (samples-per-scan)	Scans per Buffer (scanCount)	Unpacked Buffer Length (16-bit words)	Packed Buffer Length (16-bit words)
1	1	1	—
	2	2	—
	3	3	—
	4	4	3
	5	5	—
	6	6	—
	7	7	—
	8	8	6
2	1	2	—
	2	4	3
	3	6	—
	4	8	6
3	1	3	—
	2	6	—
	3	9	—
	4	12	9
	5	15	—
	6	18	—
	7	21	—
	8	24	18
4	1	4	3
	2	8	6
	3	12	9
	4	16	12
127	1	127	—
	2	254	—
	3	381	—

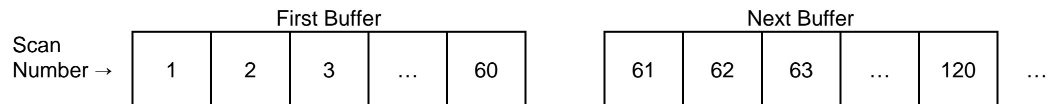
	4	508	381
	5	635	—
	6	762	—
	7	889	—
	8	1016	762
128	1	128	96
	2	256	192
	3	384	288
	4	512	384

The cycle argument is a Boolean (true or false) that specifies if, once the buffer is full, new data is allowed to overwrite old data that is already in the buffer.

- If cycle is **false**, the buffer is used as a **linear buffer**—samples are stored into the buffer until the buffer is full and the transfer stops.
- If cycle is **true**, the buffer is used as a **circular buffer—**once the buffer is full, the transfer begins again at the beginning of the buffer, overwriting the old data.

When the WaveBook transfers data into a linear buffer, it places the first sample into the first word of the buffer, and continues transferring data until the buffer is full. Once the buffer is full, the transfer is stopped, and no more samples are written to the buffer. This preserves every sample from the WaveBook; none are ever overwritten in a linear buffer. Of course, even with the transfer stopped, the WaveBook, as configured by the acquisition mode and scan count, may still be acquiring more scans. If more scans are taken, they are stored in the WaveBook's internal buffer until they can be transferred to the PC. Another `wbkBufferTransfer` command must be used to transfer them into the PC. If the transfer is not restarted, or is not restarted soon enough, the acquired scans will eventually overflow the WaveBook's internal buffer and samples will be lost.

The following diagram shows the scans as they are transferred into the first two of several linear buffers:



When the WaveBook transfers data into a circular buffer, it continues transferring until the acquisition is complete. If the amount of data acquired exceeds the capacity of the buffer, then the oldest samples in the buffer are overwritten. As more samples are acquired, more and more of the buffer is overwritten so that the most recent samples are always kept in the buffer. For example, if an acquisition is configured for 1000 scans and the buffer holds only 60 scans, then the buffer will initially be 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—so, scan 121 is stored at the beginning of the buffer, overwriting scan 61. As shown below, this process of overwriting and re-using the buffer continues until all 1000 scans have been acquired.

1st Data Set												
Buffer Position	1	2	3	…	39	40	41	42	...	58	59	60
Scan	1	2	3	...	39	40	41	42	...	58	59	60
2nd Data Set												
Buffer Position	1	2	3	...	39	40	41	42	...	58	59	60
Scan	61	62	63	...	99	100	101	102	...	118	119	120
3rd Data Set												
Buffer Position	1	2	3	...	39	40	41	42	...	58	59	60
Scan	121	122	123	...	159	160	161	162	...	178	179	180

When the acquisition is complete, the buffer holds only scans 941 through 1000. All of the preceding scans have been overwritten. At this point the buffer has the following contents:

Final Data Set												
Buffer Position	1	2	3	...	39	40	41	42	...	58	59	60
Scan	961	962	963	…	999	1000	941	942	…	958	595	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. For easier processing, the `wbkBufferRotate` command can arrange the buffer so that the oldest scan is at the beginning of the buffer (and the last is at the end). `wbkBufferRotate` only works on unpacked data

(one sample per 16-bit word). If the buffer were transferred as packed data, then `wbkBufferUnpack` must first be used to unpack the buffer. Once `wbkBufferUnpack`, if necessary, and `wbkBufferRotate` have been used, the buffer will be unpacked with the scans in their natural order from oldest to newest (first to last).

Overlapped Execution

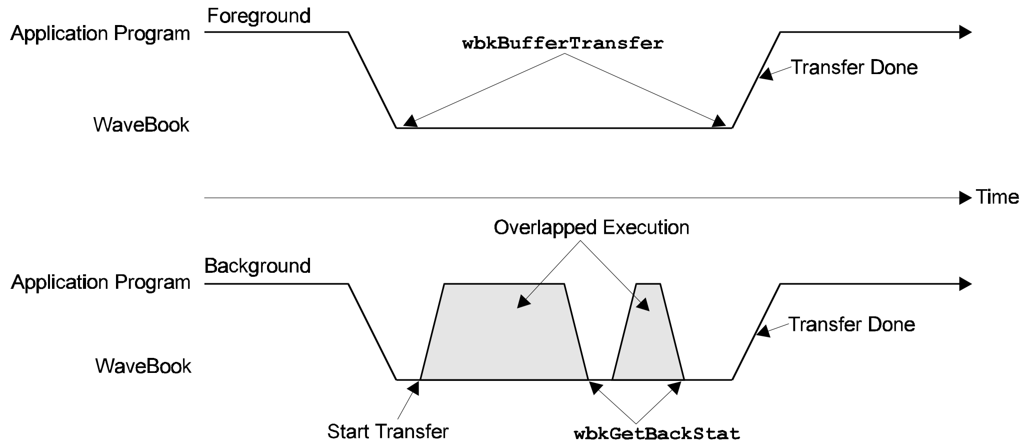
During data acquisition, it is often desirable to continue execution of the application program while the acquisition continues automatically in the background. `wbkBufferTransfer` controls this overlapped execution. `wbkBufferTransfer` configures data transfers to be either foreground transfers, in which the `wbkBufferTransfer` command waits for the transfer to complete before returning to the application program, or background transfers, in which the command returns as soon as possible to the application. The overlapped operation is controlled by the "foreground" argument to `wbkBufferTransfer` which, if true, specifies that `wbkBufferTransfer` is to wait for the transfer to complete or, if false, specifies that `wbkBufferTransfer` is to return immediately.

Foreground transfers are the simplest to use. Once a foreground transfer is started with `wbkBufferTransfer`, that command waits for the transfer to complete before returning control to the application program. This allows the application program to assume that the data is in the buffer and is ready to be analyzed or stored.

Background transfers are more complicated. Once a background transfer is started with `wbkBufferTransfer`, that command returns immediately to the application program. The application cannot assume anything about the contents of the buffer. Instead, it must use the `wbkGetBackStat` command to retrieve the state of the buffer and determine if it contains any data to be processed.

In spite of this added complexity, background transfers are often used because they allow the application program to continue execution during the transfer. Background operation allows the application to continue with other operations such as processing previously acquired data and responding to other events, including user input, during the acquisition. This capability can greatly improve the efficiency and usability of the application.

The following diagram shows how the PC directs its attention during foreground and background transfers:



The choice of the appropriate combination of foreground and cycle operation along with the acquisition mode (as set by `wbkSetAcq`) depends on way the application needs to process the received samples. The four combinations of foreground and cycle (foreground-linear, foreground-circular, background-linear, and background-circular) are each appropriate for a different approach to data processing. The following sections describe each of these approaches and then direct-to-disk transfers.

Foreground-Linear Transfers

Foreground transfers into a linear buffer simply transfer data until the buffer is full or until the acquisition stops. Once this occurs, `wbkBufferTransfer` returns, and the error code, return count, and active flag may be examined to decide what further action (if any) is necessary. The return count holds the number of scans that were transferred. This will be the same as the size of the buffer unless the acquisition stopped before the buffer was filled. The active flag is non-zero if the buffer was not large enough to hold the entire acquisition and the acquisition is still active. In this case, another `wbkBufferTransfer` must be invoked to transfer the additional samples. If the active flag is zero, then the acquisition is complete and the buffer holds the number of scans indicated by the return count. Because the buffer is a linear buffer, the earliest scan is at the beginning of the buffer and the last scan is closest to, or at, the end of the buffer.

If the samples were data-packed for transfer (as controlled by `wbkSetDataPacking`, then the data in the buffer will be in the packed-data format and may be unpacked using `wbkBufferUnpack`.

Foreground linear mode may be used with any of the acquisition modes (N-shot, N-shot with re-arm, pre/post-trigger, and infinite post-trigger) but is most useful in N-shot mode.

In N-shot mode, each acquisition collects exactly N (the post-trigger count) scans. If the buffer is large enough to hold all of the scans, then when `wbkBufferTransfer` returns, the entire acquisition will be ready in the buffer (assuming no error occurred). The scans may then be unpacked, if necessary, and analyzed or stored as desired.

In all other acquisition modes, `wbkBufferTransfer` will return when the buffer is full, but the acquisition will, usually, not yet be completed. Thus, samples will continue to be acquired, and the application program may not have much time before it must invoke another `wbkBufferTransfer` to continue the sample transfer and avoid losing data.

Foreground-Cycle Transfers

Cycle mode is used in foreground transfers to collect only the last buffer-full of data. When the `wbkBufferTransfer` command returns, the application program only has access to as many of the last scans as fit in the buffer. For example, if the buffer is large enough to hold 100 scans, the foreground cycle transfer will complete with the buffer holding the last 100 acquired scans (assuming that at least 100 scans were acquired). This type of transfer is most often used in pre/post-trigger acquisitions. In pre/post-trigger acquisitions, the actual number of scans acquired depends on the timing of the trigger, but is at least the total of the specified number of pre-trigger and post-trigger scans. By using a buffer that is just large enough to hold those scans then, when the acquisition completes, the buffer will hold just the scans of interest.

Foreground cycle transfers may also be used in the N-shot acquisition mode to capture the scans that occur sometime after the trigger. For example, if the acquisition were configured for 1000 scans after the trigger, and the buffer held only 60 scans, then, after the acquisition, the buffer would hold just scans 941 through 1000, and the preceding scans would have been overwritten. Foreground cycle transfers may also be used for direct-to-disk acquisition (discussed at the end of this chapter).

When a circular buffer is used with any acquisition mode, the scans are generally out of chronological order at the completion of an acquisition. As discussed above, `wbkBufferUnpack` and `wbkBufferRotate` may be needed to arrange the buffer into its natural order from oldest to newest (first to last).

Note: the foreground-cycle mode should never be used with the infinite acquisition modes (N-shot with re-arm and infinite post-trigger) as the acquisition would never complete, so `wbkBufferTransfer` would never return, and the application program would be stuck acquiring data until a time-out error occurs.

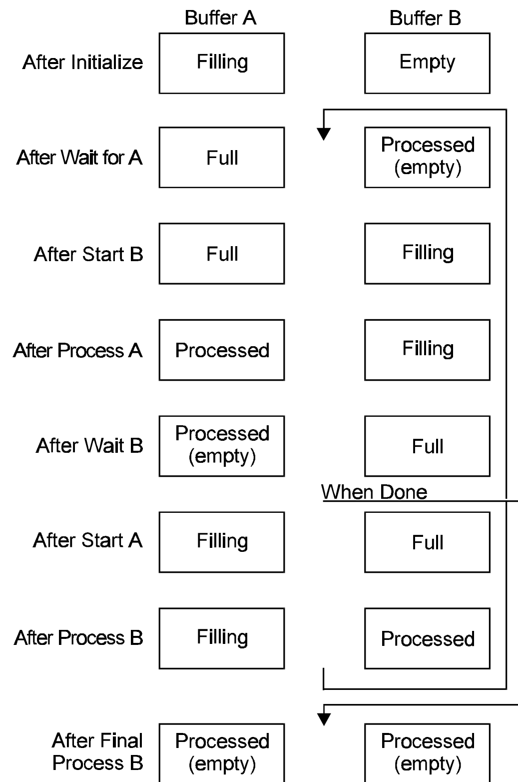
Background-Linear Transfers

Background transfer into a linear buffer is similar to foreground transfer but with the advantage that the application program continues execution while the data is being transferred. This allows the application program to process one buffer while another is being acquired. While the practical amount of processing depends strongly on the PC performance and the total sample rate, the background mode makes multiple buffer acquisitions more feasible.

In a typical application, a long acquisition is transferred using background linear transfer into two buffers, A and B, in an alternating ping-pong fashion. This type of acquisition is very useful when the total amount of data to be processed exceeds the available buffer space and may be appropriate with any of the acquisition modes: N-shot (when N is large), N-shot with re-arm, infinite post-trigger, and pre/post-trigger.

In this way, an infinite amount of data can be processed with two data buffers. As each buffer is filled it may be unpacked, if necessary, with `wbkBufferUnpack` before further processing. A ping-pong transfer can be implemented with the following procedure (see figure):

1. Initialize. The acquisition is configured and started; `wbkBufferTransfer` is invoked to transfer the data into buffer A.
2. Wait for A. The program waits until buffer A is full using `wbkGetBackStat`.
3. Start B. The program uses `wbkBufferTransfer` to start transferring data into buffer B.
4. Process A. The contents of buffer A are processed (analyzed and/or stored).
5. Wait for B. The program waits until buffer B is full using `wbkGetBackStat`. If the transfer is done, then buffer B should be processed and the procedure is complete.
6. Start A. The program uses `wbkBufferTransfer` to start transferring data into buffer A.
7. Process B. The contents of buffer B is processed (analyzed and/or stored). The program then repeats these steps starting with "Wait for A" until the entire acquisition has been processed.



Background-Cycle Transfers

Background cycle transfers may be used to allow processing to continue while acquiring just the last buffer-full of data (like foreground cycle transfers), or they may be used to allow processing of all the data while it is being acquired (like background linear mode). It may also be used to allow the application to continue during direct-to-disk transfers.

If only the final buffer of data is of interest, then background cycle mode can be used just like foreground cycle mode, but instead of the data being ready when `wbkBufferTransfer` returns, the application program will continue while the data is acquired and can use `wbkGetBackStat` to know when the acquisition is done. The application can then use `wbkBufferUnpack`, if needed, and `wbkBufferRotate` to process the data.

If all of the data is of interest, then `wbkGetBackStat` can be used to monitor the acquisition's progress. Once some of the buffer has been filled, it may be processed while the remainder of the buffer is being filled. Then, once the first part has been processed and some of the remainder filled, the new data may be processed while the old data is overwritten. The alternating portions are processed until the acquisition is complete.

Direct-to-Disk Transfers

The WaveBook can automatically copy transferred data to a disk file. If such a direct-to-disk transfer is enabled, with the `wbkSetDiskFile` command, then the WaveBook copies the data to the disk after it has been transferred into the data buffer in the PC's memory. Direct-to-disk is easiest to use with foreground circular-buffer transfers, but it can be used with background transfers or linear buffers.

When used with a foreground circular-buffer transfer, the direct-to-disk function is completely automatic: all of the acquired data is automatically written to the specified disk file. Background circular-buffer transfers are somewhat more difficult: samples are written to disk only when `wbkGetBackStat` is invoked. The application program must invoke `wbkGetBackStat` often enough to allow the data to be written to disk before it is overwritten by new samples and lost.

Direct-to-disk may also be used with foreground or background linear-buffer transfers; but in these cases, the application program must start new transfers as each linear buffer is filled.

Regardless of the type of transfer, the data is written to disk in the same format: a continuous stream of data words, exactly as received from the WaveBook. If data packing is enabled, then the disk file will contain packed data (otherwise, unpacked data). The file does not contain any other type of information. If any other information needs to be stored, such as the scan composition or rate, then the program should write that information to the file before starting the acquisition and configure the direct-to-disk transfer to append to the data file.

Once the acquisition is complete, the application program can analyze the stored data by reading the file and, if necessary, unpacking the data.

Notes

Using WaveView 5

WaveView is a graphical Microsoft Windows application for operating the WaveBook/512 hardware. No programming knowledge is required. WaveView capabilities include:

- Setting up all analog or digital input parameters, then acquiring and saving the data to a disk file or viewing the data in real time.
- Transmitting data to other Windows applications, such as spreadsheets and databases.
- Configuring and operating any connected expansion chassis.
- Launching PostView, an independent application that allows you to graphically view waveforms recorded by WaveView.

Note: WaveView comes in 2 formats (16-bit and 32-bit). Both versions work similarly and have only minor differences as noted in the text.

Application Startup

Loading WaveView

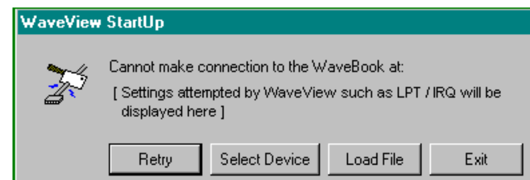
The installation disks that came with the system contain the WaveView program and its associated files. If the software has already been loaded, there is a directory (Win3.1) or folder (Win95/NT) called *wavebook* on your PC's hard drive; and WaveView is ready to run from your Windows environment (via program group or Start menu). If the software has not yet been loaded, follow the software installation procedures in chapter 2.

Starting WaveView

To launch the application, double click on the WaveView icon in the WaveView program group (or from the Start menu). WaveView holds user-configured parameters which can be saved to disk. The default configuration filename is "WAVEVIEW.CFG". When WaveView starts up, it proceeds to search the working directory for this file.

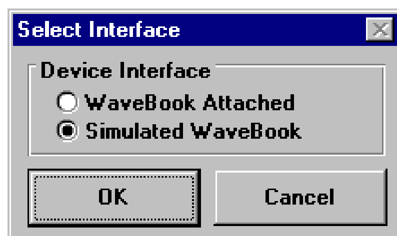
- If the default configuration file is found, all the required setup information will be extracted from it, and the application's main window will open.
- If the default configuration file is not found, WaveView will try to connect the WaveBook hardware with the following default parameters: Printer Port LPT1, Interrupt Level 7, and 4-bit Standard Protocol. If this fails, the program tries LPT2 and Interrupt Level 5.
- If connection is established, the application's main window will open with the default setting.
- If the options above fail, a dialog box will appear asking whether or not you want to open a different setup file (or to retry establishing communications, select another device, load a configuration file, or exit WaveView).

16-bit WaveView



32-bit WaveView

- If no user-configuration file is found or no communication established, a dialog box prompts you to choose a real WaveBook or a simulated WaveBook (or select among available devices).



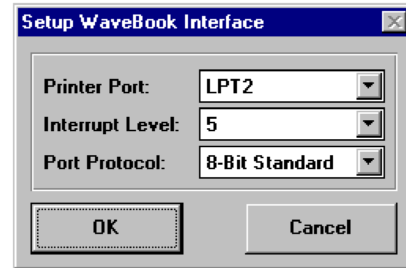
16-bit WaveView

32-bit WaveView

Simulated WaveBook

If the hardware is not available or you just want to try out the software, select Simulated WaveBook. The Simulated WaveBook allows various functions of the software to be exercised without any hardware installed.

WaveBook Attached



If the WaveBook/512 hardware is connected and switched on, select WaveBook Attached. An additional dialog box will appear, prompting you to provide the following parameters:

- **Printer Port** - LPT1, LPT2, LPT3, or LPT4. Select the port connected to the WaveBook. The default is LPT1.
- **Interrupt Level** - 3, 4, 5, 6, or 7. Select the interrupt level used by the selected LPT port. The default is 7.
- **Port Protocol** - The available options are: 8-Bit Standard, 4-Bit Standard, FarPoint F/Port EPP, 82360 SL EPP, SMC 37C66 EPP, EPP BIOS, and FAST EPP.

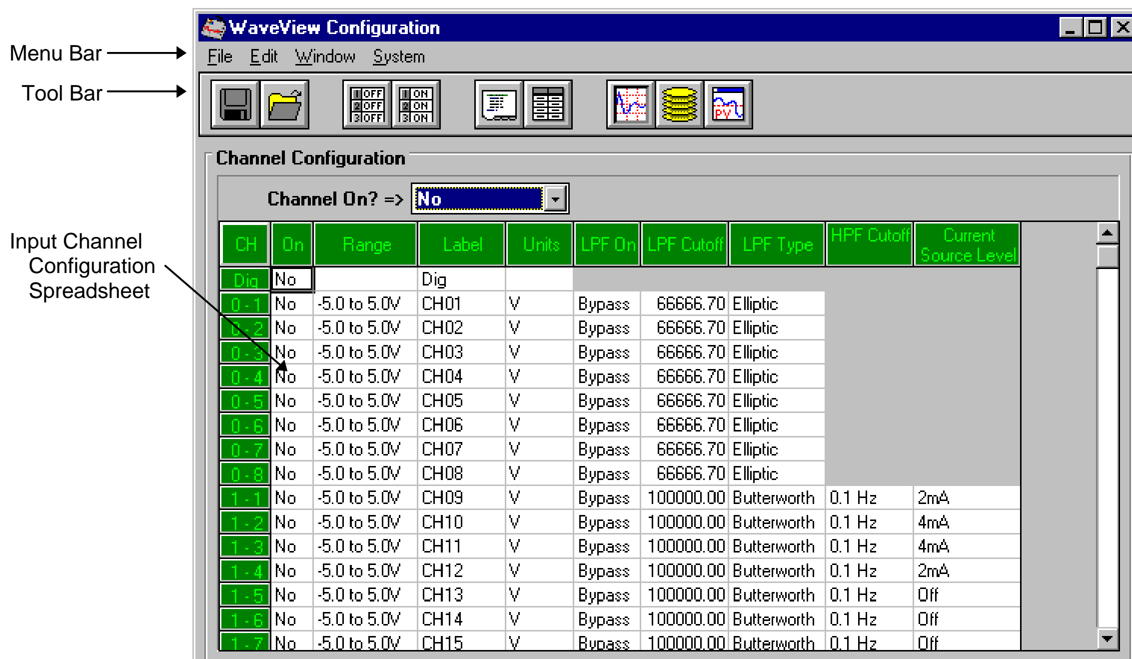
The choice of protocol depends on the hardware you are using. If you have already run the WBKTest software utility (or Test Hardware for 32-bit version), you should have determined if your particular hardware is compatible with the WaveBook. Select one of the above protocols based on those results. The default protocol is 8-Bit Standard.

WaveView will attempt to find the WaveBook/512 at the specified port.

- If the hardware is found, the application's main window will open.
- If no hardware is found, you will be alerted and given another chance to select parameters.
- If the hardware still cannot be identified by the software, exit WaveView and try the WBKTest utility program (or Test Hardware from the control panel applet for 32-bit WaveView).

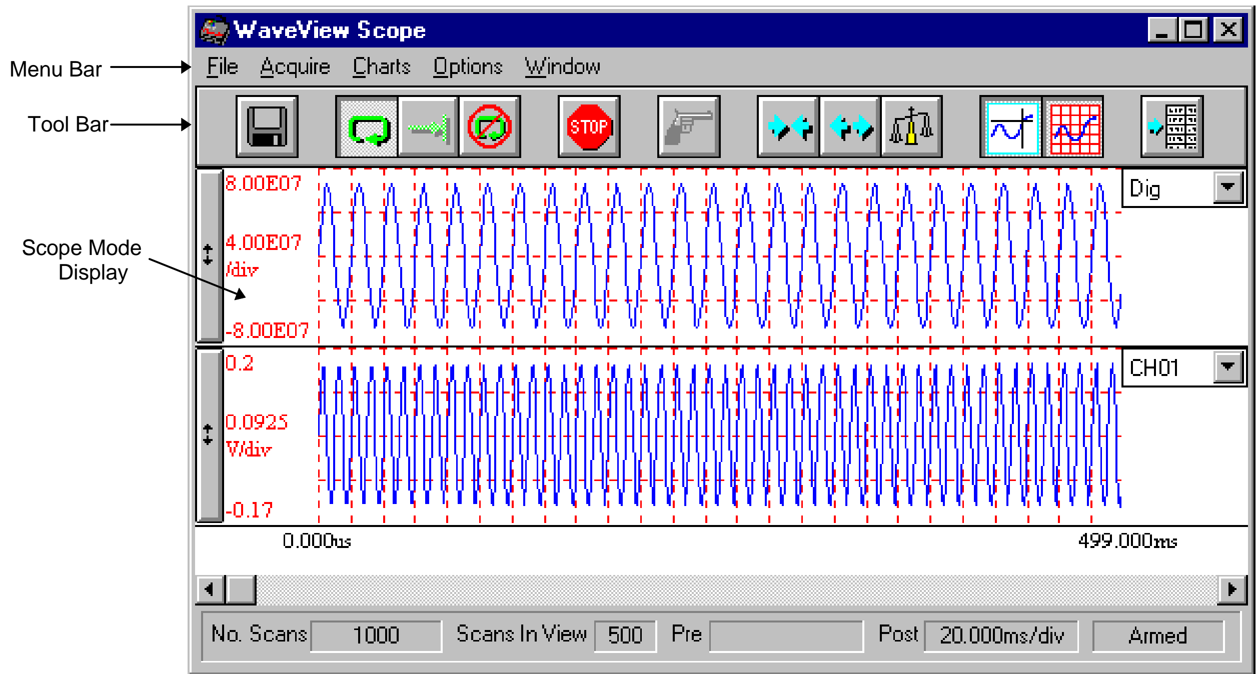
WaveView Main Components

The WaveView program has 3 main components: The Configuration screen, the Scope screen, and the Direct-to-Disk screen. The next figure shows a sample of the WaveView Configuration screen.

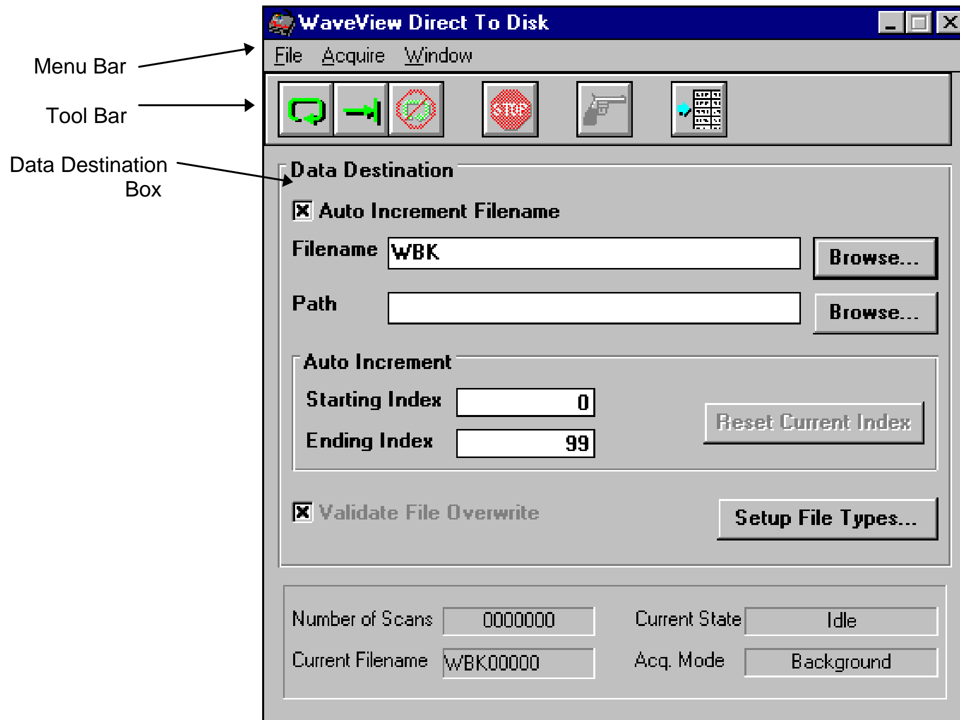


WaveView interrogates the hardware after it starts up to see what options and expansion modules are actually connected to the WaveBook. The total number of channels displayed on the configuration menu corresponds to the number of channels connected.

The next figure shows the WaveView Scope screen. Two channels are displayed in this example (up to 8 channels can be displayed at a time).



The next figure displays the WaveView Direct-to-Disk screen.



Sample Acquisition Using WaveView

The following procedure takes you through the steps involved in performing a simple acquisition. **Note**: in the simulated WaveBook mode, you can explore this guided tour in varying degrees of depth by reading ahead to other menu options as desired.

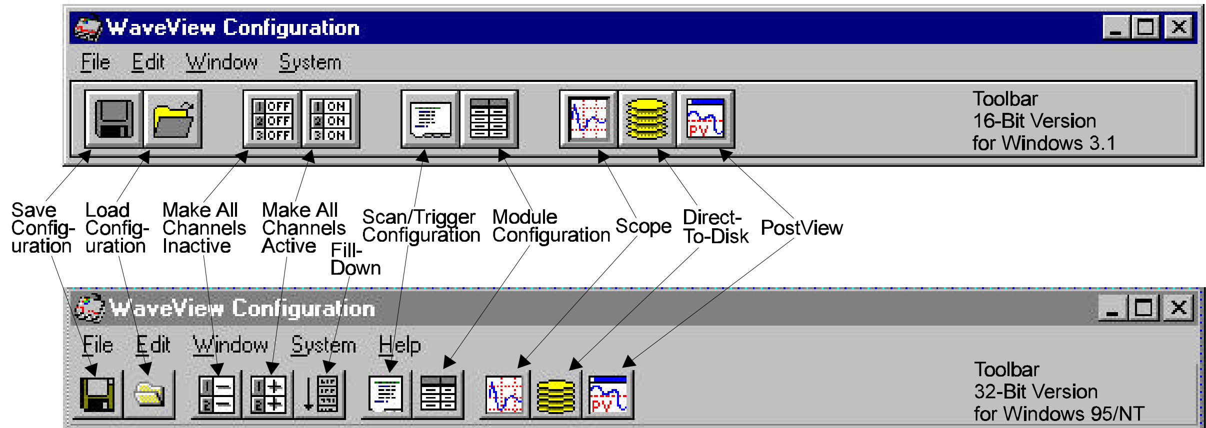
1. From the Windows environment, click on the WaveView icon. The program starts and the configuration screen is displayed.
2. Start by configuring channel 1. Double click the mouse on the box next to channel 1 in the “On” column. The box will change from a “No” to a “Yes”.
3. Move to the “Range” column for channel 1, and click on the box. Move up to the “Select Range” box, and click on it. A drop-down menu appears showing the available range selections. Click on the sixth selection (± 0.10 V). This choice now appears in the “Range” column for channel 1.
4. Move to the “Label” column for channel 1. Click on the box. Type in CHAN 1 for the label of this channel.
5. Move to the “Units” box for channel 1. Double click on the box to change the setting from V to mV. There are only two selections; repeatedly clicking the box toggles the selections.
6. Now, three more channels will be configured using steps similar to those above. Use the settings in the table.

On	Range	Label	Units
YES	± 5.0V	CHAN 2	V
YES	0.0 - 0.20 V	CHAN 3	V
YES	± 0.25 V	CHAN 4	mV

7. Now that the selections for the “Input Channel Configuration” window have been completed, move to the menu bar and click on “Window”. When the selections appear, click on “Scan/Trigger Configuration”. Move to the “Scan Count” window. Click on the “Pre-Trigger” box, and enter 1000 for the number of pre-trigger scans to take.
8. Next click on the “Post-Trigger” box, and enter 5000 for the number of scans to take after the trigger event.
9. The Scan Rate is set next. Move the mouse to the “Convention” box, and click on the “Frequency” button. Next move to the “Pre-trigger” box, and select 50 kHz. Likewise in the “Post-trigger” box, select 50 kHz.
10. Select “Immediate” from the “Trigger” selection box.
11. Move again to the menu bar, and click on “Window”. When the selections appear, click on “Scope”. The Scope screen will display.
12. Next, the screen needs to be configured to display 4 charts since 4 channels were previously selected for the acquisition. Click on the “Charts” menu item on the Scope screen. When the selections appear, click on “Number of Charts”. A flyout appears showing a selection of up to 8 channels for display. Click on 4.
13. The system is now set to start collecting data. Click on the “One Shot” button.
14. The system has now collected 1000 pre-trigger scans and 5000 post-trigger scans.
15. If the waveforms need to be scaled, click on the “Scale All Charts” button. All 4 waveforms should be visible at this time.
16. The waveforms may be examined at any point along the timeline by using the scroll-bar at the bottom of the screen.
17. The number of scans displayed in the window may be varied by using the “Zoom In” or “Zoom Out” buttons.
18. To store the acquisition data to disk, click on the “Save Data File” button and give the file a name.
19. Return to the WaveView Configuration screen by clicking on the “Go to Config Window” button.
20. Once back in the Configuration window, the saved data may be examined using the “PostView” utility. Click on the “Window” menu item. When the selections appear, click on “PostView”. The PostView screen then appears. Any desired data file saved by WaveView may then be recalled for analysis.

WaveView Configuration Menu Items & Buttons

Control functions in the configuration window are available through the pull-down menu or the toolbar. The figure shows the menu, the toolbar, and each tool icon. In the following sections, menu functions are explained in order of the menu structure (not all menu items have a corresponding tool icon).



File

The file menu provides four basic functions:

New	Set all parameters to their default startup setting.
Save	Save the existing configuration for later recall.
Load	Load a saved configuration.
Exit	Leave the WaveView program.

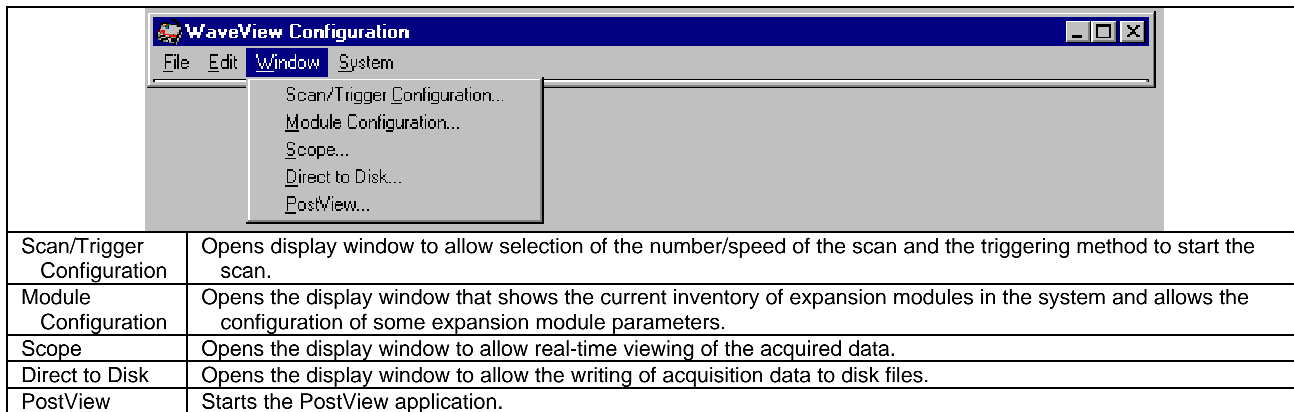
Edit

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.
Go To Row	This command pops up a dialogue box which allows you to enter a channel number to be modified. For hardware configurations that contain a large number of channels, this is a faster method of moving around than using the scroll bars.
Fill Down	When multiple cells within a column are selected, this command takes the top-most selected cell and copies its contents to the selected cells below.

Window

The Window menu includes the following functions:



Function	Description
Scan/Trigger Configuration	Opens display window to allow selection of the number/speed of the scan and the triggering method to start the scan.
Module Configuration	Opens the display window that shows the current inventory of expansion modules in the system and allows the configuration of some expansion module parameters.
Scope	Opens the display window to allow real-time viewing of the acquired data.
Direct to Disk	Opens the display window to allow the writing of acquisition data to disk files.
PostView	Starts the PostView application.

System

The System menu includes the following functions:

Function	Description
Select LPT Port (16-bit version)	Brings up a dialog box prompting you to select the LPT port, interrupt level, and protocol for the WaveBook. After an LPT port is selected, WaveView opens a new session with the WaveBook hardware and attempts to communicate with it. If the hardware is found, the application's main window is opened. If no hardware is found, you will be alerted 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 WaveBook hardware still cannot be identified by the software, exit WaveView and try the WBKTest utility program.
Select Device (32-bit version)	Opens the Select Device dialog box that shows you the devices to choose from currently in the system.
Simulated WaveBook	This command opens a WaveView session but does not attempt to communicate with WaveBook hardware. Instead, the application simulates the interaction between the software and the hardware. If WaveView is presently attached to real WaveBook hardware, this command will close that session.
Factory Calibration Table	The software uses the factory generated calibration constants of each component in the system to achieve calibration of the system as a whole. These calibration constants are stored on each device in the system. This allows for a "plug-and-play" mixing of devices while still allowing the system to remain calibrated. This calibration method is useful if the system configuration changes often.
User Calibration Table	The WaveCal program allows you to perform a calibration of the complete signal path from the input to the A/D stage. The calibration constants are stored in the Calibration Table on the WaveBook main board. Recalibration is required whenever any part of the signal path is changed (eg adding an expansion chassis or WBK11). Slightly better accuracy can be achieved at the price of manually recalibrating the system. This method is preferred when the configuration remains relatively stable and the user wants improved accuracy.
Acquire Data Packed	The normal 16-bit size of the data is compressed to 12 bits so that four 12-bit samples can pack into three 16-bit words. Such packing is useful when extremely fast acquisitions are required; the amount of transfer data is reduced 25% with a corresponding reduction in transfer time. Other advantages of using packed data include less data to store in the buffer and less data to archive. Use packed data if buffer overrun errors are generated during the acquisition. Disadvantages of the packed-data format include extra processing steps for unpacking the data and some loss in resolution (less than 1/2 LSB). The normal recommendation is to use packed data.
Acquire Data Unpacked	Data is acquired in standard 16-bit format. Access to unpacked data is faster than packed data since no data decompression needs to be performed. Normally, unpacked data can be used if no buffer overruns occur during the acquisition. Disadvantages of unpacked data include slower transfer of data from the WaveBook to the PC, and the need for more disk space if the data is to be archived.
Set Foreground Mode	This command forces WaveView to acquire data in foreground mode. Acquiring data in the foreground allows for a faster acquisition rate using full resources of the computer. The trade-off is that WaveView will hold the computer locked during such an acquisition till the acquisition is complete or interrupted by an error (e.g., buffer overrun). Using foreground mode is generally recommended.

WaveView Configuration Screen Components

Input Channel Configuration

The display below shows the layout for the channel configuration spreadsheet.

Channel Configuration

Channel On? => Yes

CH	On	Range	Label	Units	LPF On	LPF Cutoff	LPF Type	HPF Cutoff	Current Source Level
Dig	Yes		Dig						
0 - 1	Yes	-5.0 to 5.0V	CH01	V	Bypass	66666.70	Elliptic		
0 - 2	Yes	-5.0 to 5.0V	CH02	V	Bypass	66666.70	Elliptic		
0 - 3	Yes	-5.0 to 5.0V	CH03	V	Bypass	66666.70	Elliptic		
0 - 4	Yes	-5.0 to 5.0V	CH04	V	Bypass	66666.70	Elliptic		
0 - 5	Yes	-5.0 to 5.0V	CH05	V	Bypass	66666.70	Elliptic		
0 - 6	Yes	-5.0 to 5.0V	CH06	V	Bypass	66666.70	Elliptic		
0 - 7	Yes	-5.0 to 5.0V	CH07	V	Bypass	66666.70	Elliptic		
0 - 8	Yes	-5.0 to 5.0V	CH08	V	Bypass	66666.70	Elliptic		
1 - 1	Yes	-5.0 to 5.0V	CH09	V	Bypass	100000.00	Butterworth	0.1 Hz	2mA
1 - 2	Yes	-5.0 to 5.0V	CH10	V	Bypass	100000.00	Butterworth	0.1 Hz	4mA
1 - 3	Yes	-5.0 to 5.0V	CH11	V	Bypass	100000.00	Butterworth	0.1 Hz	4mA
1 - 4	Yes	-5.0 to 5.0V	CH12	V	Bypass	100000.00	Butterworth	0.1 Hz	2mA
1 - 5	Yes	-5.0 to 5.0V	CH13	V	Bypass	100000.00	Butterworth	0.1 Hz	Off
1 - 6	Yes	-5.0 to 5.0V	CH14	V	Bypass	100000.00	Butterworth	0.1 Hz	Off
1 - 7	Yes	-5.0 to 5.0V	CH15	V	Bypass	100000.00	Butterworth	0.1 Hz	Off

The spreadsheet allows the analog input channels and/or digital channel to be configured and displayed. The spreadsheet consists of rows and columns much like an accounting spreadsheet. The top row is for the high-speed digital input channel. Rows 1 through 72 configure the analog input channels. The number of rows may vary depending on system configuration. The various columns contain the configuration information for each channel. Some columns allow blocks of cells to be altered simultaneously while others only allow one cell to be changed at a time. Some columns may be static and cannot be altered by you. Clicking a column header will select the entire column if applicable.

CH

The channel number column labeled CH is static and cannot be altered. This column identifies the analog (or digital) input channel to be configured in that row. This number includes all channel numbers from the WaveBook/512 and attached expansion chassis (WBK10, WBK14, and/or WBK15). The channels are numbered as follows:

CH	Description	Default Label
Dig	WaveBook Digital Channel	Dig
0-1 to 0-8	WaveBook Analog Channels	CH01 to CH08
1-1 to 1-8	First Expansion WBK10 Channels	CH09 to CH16
2-1 to 2-8	Second Expansion WBK10 Channels	CH17 to CH24
etc.	etc.	etc.

On

This column allows you to include or exclude a channel from the scan list. When a cell is selected, the selection box above the spreadsheet allows “Yes” or “No” 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 either “Yes” or “No”.

Range

This column allows you to set the gain and polarity for the selected channel(s). Clicking the mouse in any of the analog channel Range boxes brings up the Select Range selection box. The range of gains available in the selection box depends on whether or not a WBK11 Simultaneous Sample and Hold option card is installed in the system. The ranges available without the option card are:

0 - 10.0 V	± 5.0 V
0 - 5.0 V	± 2.5 V
0 - 2.5 V	± 1.0 V
0 - 1.0 V	± 0.5 V

If the WBK11 simultaneous sample and hold card is installed, the following additional gain selections are available for the channels of the device (WaveBook/512 or WBK10 Expansion Chassis) containing the option card:

0 - 0.5 V	± 0.25 V
0 - 0.25 V	± 0.10 V
0 - 0.1 V	± 0.05 V

Double clicking on a cell will cycle through the available ranges. The Range selections have no effect on the Digital Input channel. **Note**: The WBK11 ranges apply to the WBK12/13; the WBK14 adds the range ±0.025 V; the WBK15 ranges depend on the particular 5B module used.

Label

This column contains a descriptive name for the input channel. By default, it contains a label which is similar to its channel number, but this can be changed to any combination of 8 characters. Click on the desired cell, and type in the desired label name. This column does not have a selection list above the spreadsheet and does not allow selecting multiple blocks of cells.

Units

This column allows you to change the voltage scale setting of each analog channel displayed when the Scope option is selected. When a cell is selected, a selection box gives you a choice between V or mV. You can also enter user units and mx+b scales from this point. Making a selection sets the choice into the individual cell or block of cells. This option has no effect on the Digital Input channel.

The following options depend on your actual equipment configuration.

LPF On (WBK12/13, WBK14)

This column allows you to include or exclude the low-pass filter from a channel. When selecting a cell, the selection box above the spreadsheet allows “On” or “Bypass” to enable or disable the filter or block of filters. Double clicking a cell in this column will toggle the filter status.

LPF Cut-Off (WBK12/13, WBK14)

This column allows you to set the low-pass filter cut-off frequency for the selected channel(s). If you enter an inappropriate cut-off frequency, the software will round up or down to the next appropriate frequency for your particular hardware. Since the WBK12/13 filters are assigned to banks, setting the value in one channel of a bank will update the others.

LPF Type (WBK12/13, WBK14)

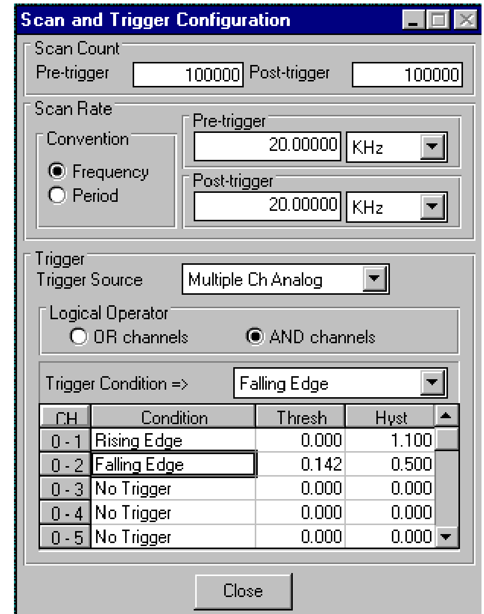
This column allows you to configure the low-pass filter for the selected channel(s). When selecting a cell or block of cells in this column, a selection box above the spreadsheet may or may not appear, depending upon your particular hardware. If the selection box appears, it will display the appropriate low-pass filter selections (such as “Elliptic” or “Linear” for the WBK12/13) allowed by your hardware to configure a filter or block of filters. Double clicking a cell in this column will toggle the filter type status. A change in the low-pass filter type for one channel will appropriately update any other channels that are affected.

HPF Cut-Off (WBK14 only)

This column allows you to set the high-pass filter cut-off frequency for the selected channel(s). When a cell is selected, a selection box above the spreadsheet will display the appropriate cut-off frequency selections (such as “0.1 Hz” or “10 Hz”) to configure the filter(s). Double clicking a cell in this column will toggle the cut-off frequency status. A change in the high-pass filter cut-off frequency for one channel will appropriately update any other channels that are affected.

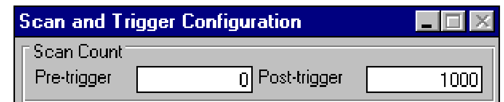
Current Source Level (WBK14 only)
This column allows you to apply or remove the current source level for the selected channel(s). When selecting a cell or block of cells in this column, a selection box above the spreadsheet may or may not appear, depending upon your particular hardware. If the selection box appears, it will display the appropriate current level selections (such as "Off", "2 mA", or "4 mA") allowed by your hardware to configure a source or block of sources. Double clicking a cell in this column will toggle the current level status. A change in the current source level for one channel will appropriately update any other channels that are affected. **Note**: When using an ICP transducer, either 2 mA or 4 mA must be selected. When measuring voltage, set the current-source level to "Off".

Scan and Trigger Configuration



The display below shows the layout for the Scan and Trigger Configuration window. This window is displayed by making the Window menu selection or clicking on the button bar icon.

Scan Count



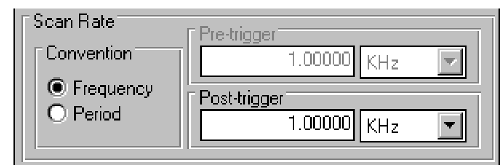
The Scan Count entry box is within the Scan and Trigger Configuration window. This selection box allows you to set the number of scans to take prior to the trigger and following the trigger. These settings will be used when an acquisition is started.

The following formula determines the maximum number of scans that can be stored for use in the Scope mode (does not apply to storage to disk):

$$(\textit{Scan Count}) * (\textit{\# of Channels}) * 4 < \textit{Available PC Memory}$$

A scan includes all of the channels that are marked as "On" in the analog input configuration spreadsheet.

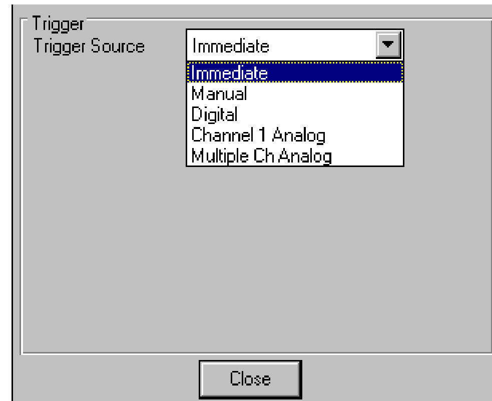
Scan Rate



Just below the Scan Count box is the Scan Rate selection box. The Scan Rate box allows you to select how fast to scan both pre-trigger and post-trigger. It can be set to either Frequency or Period. If Frequency is selected for the timebase, the units can be set to Hz, kHz, or MHz. For Period, the units can be set to seconds, milliseconds, or microseconds.

Trigger

The Trigger selection box allows you to select the triggering method to start the scan. The screen shows the various triggering options available.



Immediate

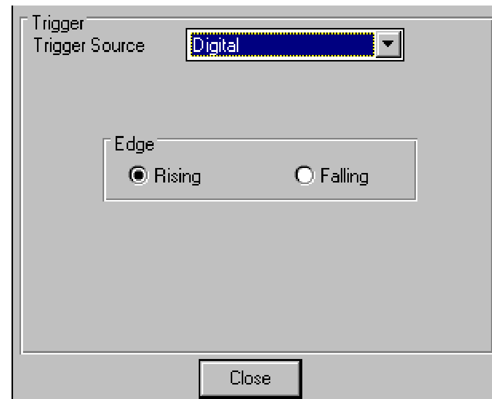
Triggering starts when the One-Shot or Continuous button is clicked.

Manual

Prior to acquiring data, the system must first be armed by clicking on the One-Shot or Continuous button. The Manual button is then clicked to start the acquisition.

Digital

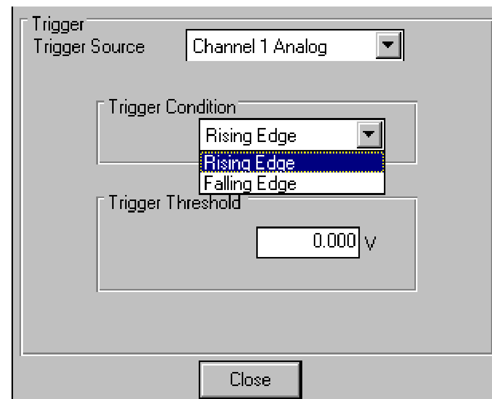
Selecting Digital brings 2 triggering options to the Trigger selection box, allowing you to select either a rising or falling-edge trigger. The TTL trigger signal connects to pin (TTLTRG) of the Digital I/O & Trigger port on the WaveBook front panel.



Channel 1 Analog

This option allows you to set up additional parameters for the acquisition of analog data. Several new items are added to the Trigger selection box, including options for the Trigger Condition.

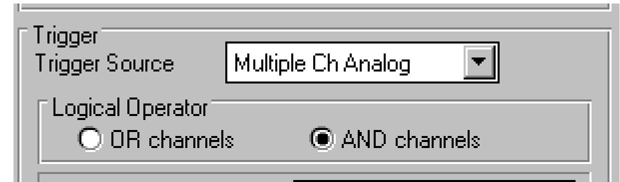
Note that Channel 1 Analog triggering is valid for Channel 1 only.



The selections for trigger correspond to the following trigger conditions:

Channel 1 Analog Triggering Selections and Conditions	
Rising Edge	The signal level must have a positive slope as it crosses the trigger level.
Falling Edge	The signal level must have a negative slope as it crosses the trigger level.

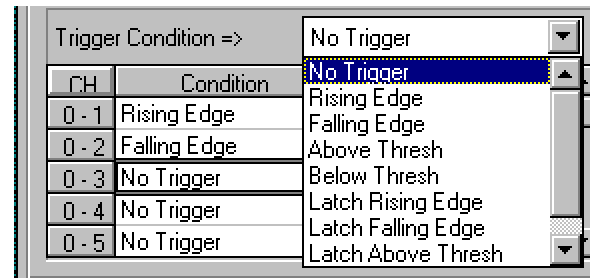
Multiple Ch Analog



Triggering can be further customized by selecting Multiple Ch Analog triggering. This option displays the following selection parameters.

- Selecting OR causes the acquisition to trigger when any of the selected channel conditions become true.
- Selecting AND issues a trigger when all the selected channel conditions become true.

The Trigger Condition option has the following choices:



The triggering selections correspond to the following trigger conditions:

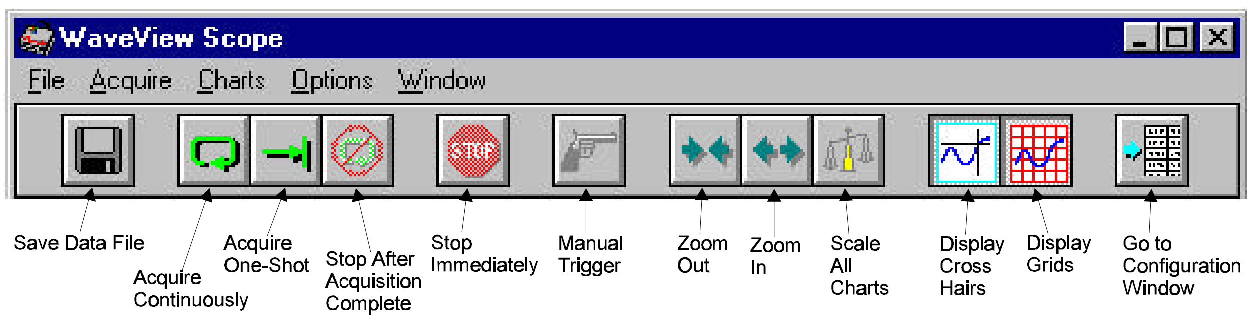
Multiple Ch Analog Triggering Selections and Conditions	
No Trigger	The channel will not be included in the list of channels to examine for trigger conditions.
Rising Edge	The signal level must first go below the trigger level by the user-set hysteresis amount. Then, the trigger channel is valid whenever the signal level is above the trigger level and stays valid until the signal level goes below the trigger level by at least the hysteresis amount.
Falling Edge	The signal level must first go below the trigger level by the user-set hysteresis amount. Then, the trigger channel is valid whenever the signal level is below the trigger level and stays valid until the signal level goes above the trigger level by at least the hysteresis amount.
Above Thresh	A trigger channel is valid whenever the signal level is above the trigger level and stays valid until the signal level goes below the trigger level by at least the user-set hysteresis amount.
Below Thresh	A trigger channel is valid whenever the signal level is below the trigger level and stays valid until the signal level goes above the trigger level by at least the user-set hysteresis amount.
Latch Rising Edge	The signal level must first go below the trigger level by the user-set hysteresis amount. Then, the trigger channel is valid whenever the signal level is above the trigger level and stays valid until the acquisition is complete.
Latch Falling Edge	The signal level must first go below the trigger level by the user-set hysteresis amount. Then, the trigger channel is valid whenever the signal level is below the trigger level and stays valid until the acquisition is complete.
Latch Above Thresh	A trigger channel is valid whenever the signal level is above the trigger level and stays valid until the acquisition is complete.
Latch Below Thresh	A trigger channel is valid whenever the signal level is below the trigger level and stays valid until the acquisition is complete.

The **thresh**old voltage and **hyst**eresis level may be set for each channel as required—position cursor per channel and type in the desired value(s).

WaveView Scope Menu Items & Buttons

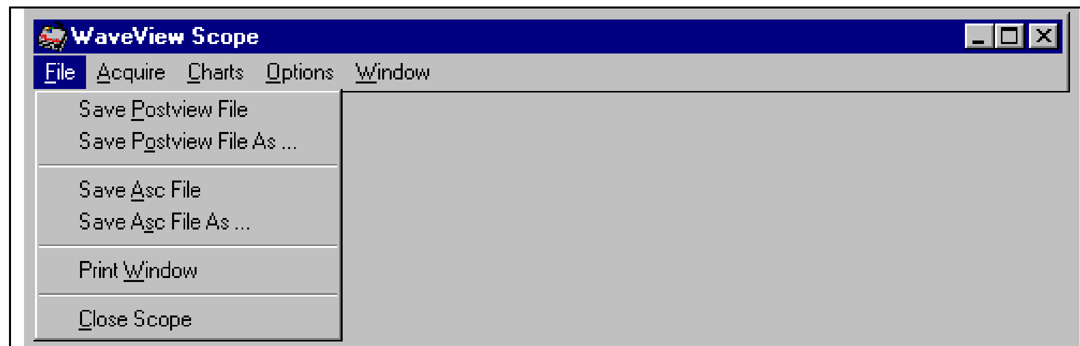
The Scope mode of the WaveView program allows you to obtain a visual representation of the acquired waveforms. An acquisition may consist of all channels physically part of the WaveBook system. A fully-configured system (a WaveBook/512 and 8 WBK10, WBK14, and/or WBK15 expansion chassis) has a total of 72 analog channels. All 72 channels may be viewed in WaveView, although not simultaneously. The maximum number of channels that can be displayed at one time is 8; however, each of the 8 displayed channels can be set to any channel in the system. A "Channel" box is located at the right end of each chart and is used to select the desired channel. Click on the box to display the channel list; then click on the desired channel. The waveform display is actually a window looking at a section of the acquisition. The window size may be increased or decreased and moved to any location on the timeline. The waveforms may be examined at any time, even during the acquisition.

Two selection methods are available for controlling the acquisition process and the Scope display: the menu bar or the tool bar. The next figure shows the menu bar, the tool bar, and the tool icons. The functions of the menu items and tool buttons are explained next.



File

The File menu under Scope Mode provides the following functions.



Save PostView File	Saves data in a PostView format. Same as the Save Data File button (icon # 1).
Save PostView File As	Saves a previously saved PostView file with a new name.
Save Asc File	Saves data in ASCII format.
Save Asc File As	Saves a previously saved ASCII file with a new name
Print Window	Prints the contents of the display screen.
Close Scope	Closes the Scope display and returns to the configuration menu.

Acquire

The menu under Acquire provides the following functions:

Acquire Continuously	Sets WaveView to a looping mode. As soon as an acquisition has completed, it is re-armed and the acquisition repeats. Requires the Stop or Abort button to stop. Same function as the Continuous Acquisition button (icon # 2).
Acquire One Shot	The acquisition starts as soon as the trigger event is satisfied. Same as the Acquire One-Shot button (icon # 3).
Stop After Acquisition Complete	Stops the data acquisition when continuous mode has been selected. The scan will finish taking the last set of samples before stopping. Same as the Stop After Acquisition Complete button (icon # 4).
Stop Immediately	Stops the data acquisition immediately without waiting for the last acquisition to finish. Same function as the Stop Immediately button (icon # 5).
Manual Trigger	Acquisition starts when the manual trigger is selected if in Continuous or One-Shot Mode. Same as Manual Trigger button (icon # 6).

Charts

This menu item allows you to select the number of channels to display. Up to 8 channels can be displayed at one time.

Number of Charts	Sets the number of charts to be displayed simultaneously. A maximum of 8 charts may be displayed.
Zoom In	Halves the visible timebase. Example: if 10 ms of information is visible, clicking Zoom In will show 5 ms. Same as the Zoom In button (icon # 8). Maximum Zoom In is 2 samples.
Zoom Out	Doubles the visible timebase. Example: if 10 seconds of information is visible, clicking Zoom Out will show 20 seconds. Same as the Zoom Out button (icon # 7). Maximum Zoom Out is 2000 samples.
Enable AutoScaling	Continuously adjusts the Y-axis for all channels so that the visible waveform fills 90% of the graph's range.
Scale All Charts	Adjusts the Y axis for all channels so that the visible waveform fills 90% of the graph's range. Same as the Scale All Charts button (icon # 9).

Options

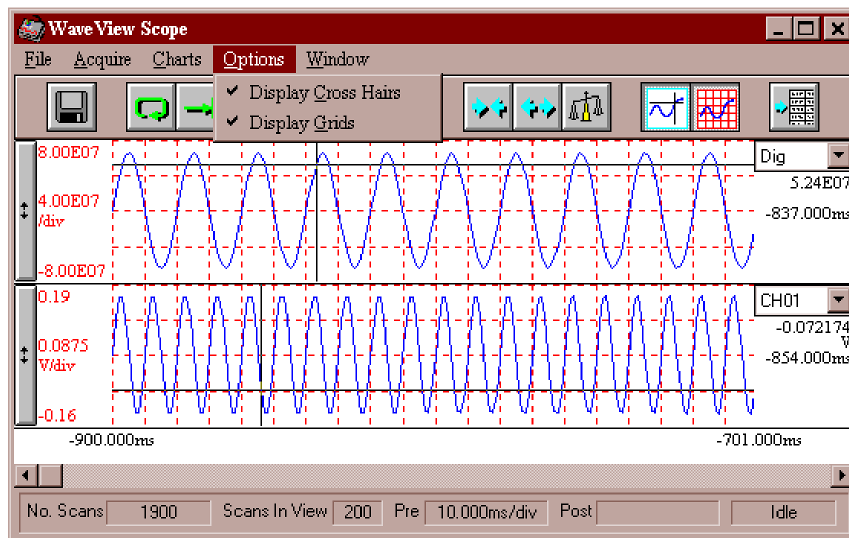
The Options menu provides you with chart display options.

WaveView Scope
File Acquire Charts Options Window
✔ Display Cross Hairs
✔ Display Grids

Display Cross Hairs	A cross hair is a marker that shows the numerical values of time and magnitude at its present location in the waveform. Same as Display Cross Hairs button (icon # 10). Toggle button to turn cross hairs on or off.
Display Grids	Displays a grid for each chart. Same as Display Grids button (icon # 11). Toggle button to turn grids on or off.

Individual Cross-Hairs can be moved by holding down the left mouse button and dragging the selected Cross-Hair to the new location on the chart. Holding the right mouse button and dragging moves all the cross-hairs simultaneously to a new location. The voltage and time display at the side changes as you do this. Cross-hairs are disabled during an acquisition.

The next figure shows the display with both the CROSS HAIRS and GRIDS turned ON.



Window

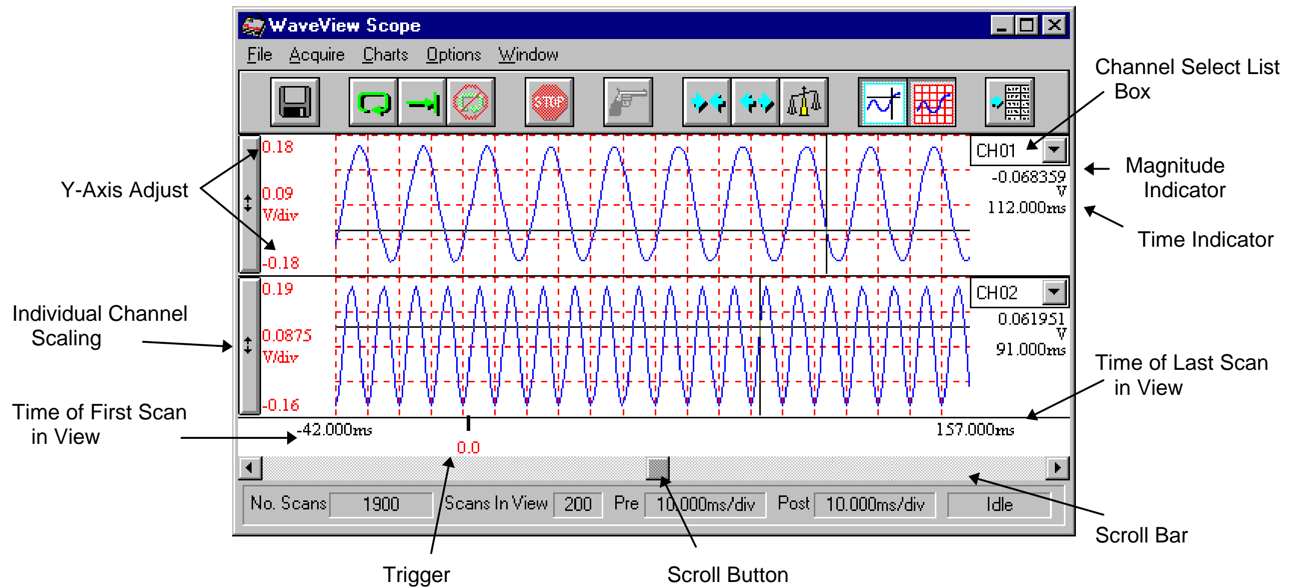
The Window menu provides the following operation.

WaveView Scope
File Acquire Charts Options Window
Configuration...

Go To Configuration Window	Leaves the Scope Mode display and goes back to the configuration menu. Unlike the Close Scope function from the File menu, this option does not close the Scope display. You can toggle back and forth between screens by clicking the mouse on the desired display screen.

WaveView Scope Display

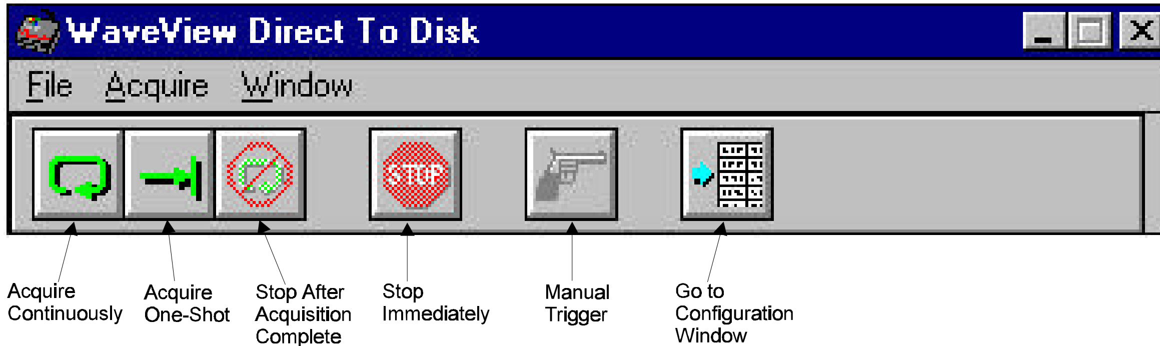
The Scope mode is WaveView's display utility. It allows you to see the data acquisition waveforms in real-time. Before Scope mode can be enabled, at least one channel must have been set to ON in the configuration spreadsheet. The Scope mode display has several indicators as shown in the next figure.



Scope Mode Indicators and Descriptions	
Y-Axis Adjust	Allows adjustment of the displayed range. Clicking on the value highlights the number. Enter desired new value and press Enter.
Individual Channel Scaling	Adjusts the scaling of the individual channels so that the visible waveform fills 90% of the graphs range.
Time of First Scan in View	Displays the starting time of the acquisition prior to the trigger event.
Trigger	Displays the trigger event.
Time of Last Scan in View	Displays the stopping time of the acquisition following the trigger event.
Channel Select List Box	Clicking on this cell displays the list of all channels selected in the WaveView configuration menu. A maximum of 8 channels may be displayed at one time with the remaining channels available through the scroll bar.
Magnitude Indicator	Displays the magnitude of the voltage at the point where the marker cross-hair intersects the waveform. Moving the marker to different locations on the waveform changes the value of the displayed voltage.
Time Indicator	Displays the point on the acquisition time-line where the marker cross-hair intersects the waveform. This value changes as the marker is moved along the X-axis (time scale).
Scroll Bar	Allows the waveform to be scrolled right or left. Click and hold on the arrows on either end of the scroll bar to move the waveform up or down the length of the acquisition.
Scroll Button	Shows the location of the displayed portion of the waveform relative to the entire acquisition. Click and hold on the scroll button, then drag to any location on the scroll bar to quickly move to a new location on the time-line of the acquisition.

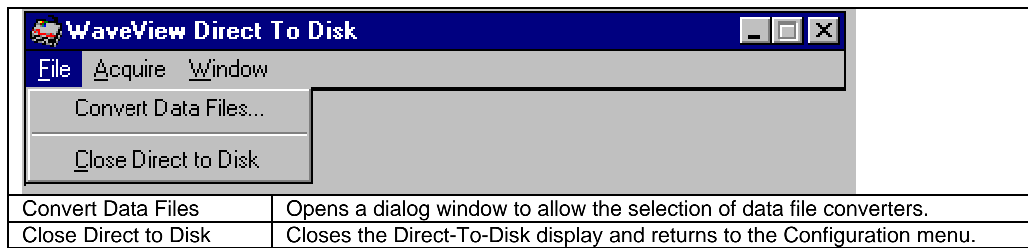
WaveView Direct-To-Disk Menu Items & Buttons

Control functions in the Direct-To-Disk window are available through the pull-down menu or the toolbar. The figure shows the menu, the toolbar, and each tool icon. In the following sections, menu functions are explained in order of the menu structure (not all menu items have a corresponding tool icon).



File

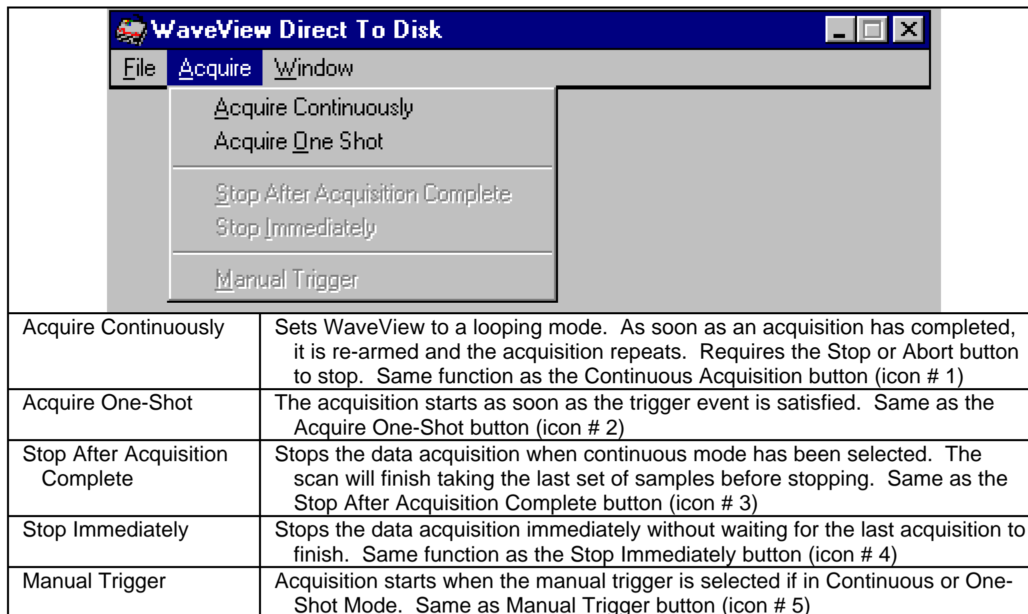
The File menu under Direct-To-Disk Mode provides the following functions.



Convert Data Files	Opens a dialog window to allow the selection of data file converters.
Close Direct to Disk	Closes the Direct-To-Disk display and returns to the Configuration menu.

Acquire

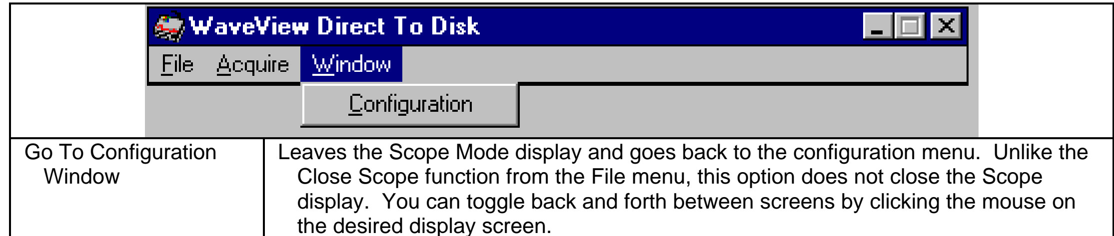
The menu under Acquire provides the following functions.



Acquire Continuously	Sets WaveView to a looping mode. As soon as an acquisition has completed, it is re-armed and the acquisition repeats. Requires the Stop or Abort button to stop. Same function as the Continuous Acquisition button (icon # 1)
Acquire One-Shot	The acquisition starts as soon as the trigger event is satisfied. Same as the Acquire One-Shot button (icon # 2)
Stop After Acquisition Complete	Stops the data acquisition when continuous mode has been selected. The scan will finish taking the last set of samples before stopping. Same as the Stop After Acquisition Complete button (icon # 3)
Stop Immediately	Stops the data acquisition immediately without waiting for the last acquisition to finish. Same function as the Stop Immediately button (icon # 4)
Manual Trigger	Acquisition starts when the manual trigger is selected if in Continuous or One-Shot Mode. Same as Manual Trigger button (icon # 5)

Window

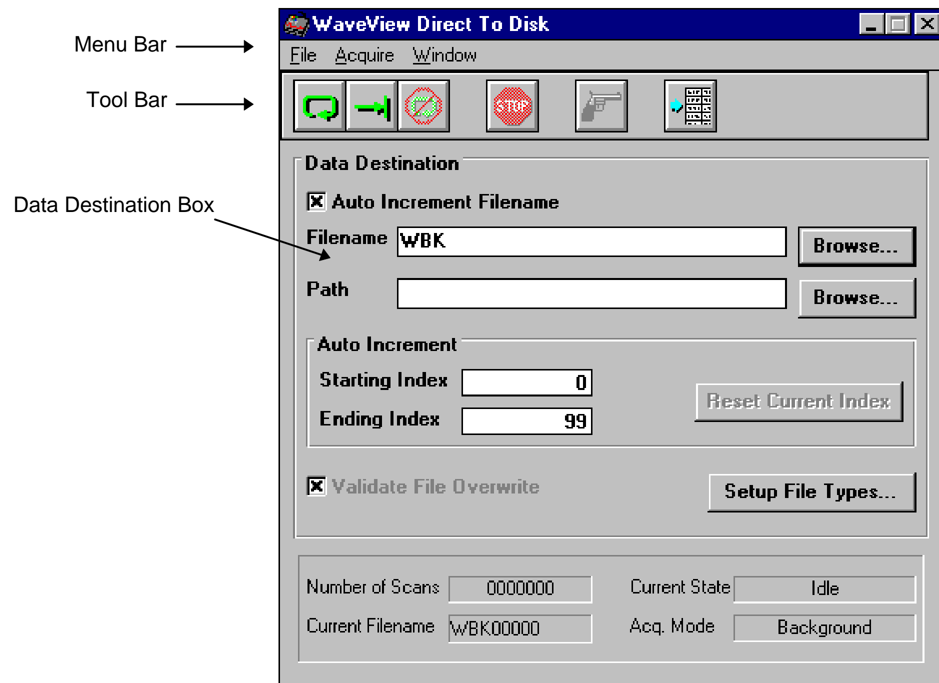
The Window menu provides the following operation.



Go To Configuration Window	Leaves the Scope Mode display and goes back to the configuration menu. Unlike the Close Scope function from the File menu, this option does not close the Scope display. You can toggle back and forth between screens by clicking the mouse on the desired display screen.

WaveView Direct-to-Disk Data Destination Box

Several file-handling options are available in the Data Destination box within the WaveView Direct-to-Disk screen (see next figure and table).



Data Destination Options on Direct-to-Disk Screen	
Auto Increment Filename checkbox	If checked - allows automatic change to the suffix of the Current Filename using the base Filename and the numbers in the "Start Index - Stop Index" range. If not checked - Current Filename will be equal to base Filename. Current Filename is shown at the bottom of the dialog box.
Filename text box	Displays base filename; allows user to input filename using keyboard or Browse button.
Path text box	Displays root path to the directory which contains several subdirectories like \bin ,\PostView, \ASCII, \sm … . Any such subdirectory contains the data files of the appropriate type.
Browse buttons	Allows user to browse and set base filename or root directory.
Reset Current Index button	Resets current index and Current Filename to the "Start Index"
Validate File Overwrite checkbox	If checked - will require confirmation for overwriting any already existing files (binary, ASCII, PostView, SnapMaster …)
Setup Filename button	Opens a dialog window to allow user to select appropriate data formats to which binary data will be converted after the acquisition.

Using PostView

PostView is an application that allows you to graphically view the waveforms recorded by WaveView. 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 WaveView'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 WaveView 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 WaveView, it automatically opens the file selected as the destination file in those applications. To view other files, use Open under the File menu.

PostView Timebase

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

Timebase for WaveView

All WaveView files begin at the trigger point (t = 0), and the time between each scan is constant.

PostView Menu Items

File

The File menu provides the following basic functions.

File Menu Items and Descriptions	
Open Data File	Open a data file created by WaveView. PostView automatically detects whether the file contains ASCII or binary data.
Print Window	Print the present PostView window.
Exit	Leave the PostView program.

Number of Charts

The Number of Charts menu provides one basic function.

Number of Charts Menu Items and Descriptions	
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 4 basic functions.

Go To Menu Items and Descriptions	
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.
Scan Number	Automatically scrolls the waveforms so that the desired scan number is in view. This menu selection invokes a dialogue box which displays the number of scans in the file.
Time	Automatically scrolls the waveforms so that the desired time is in view.
Trigger point	Automatically scrolls the waveforms so that the trigger point (t = 0) is in view.

Options

The Options menu provides 2 basic functions.

Options Menu Items and Descriptions	
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.

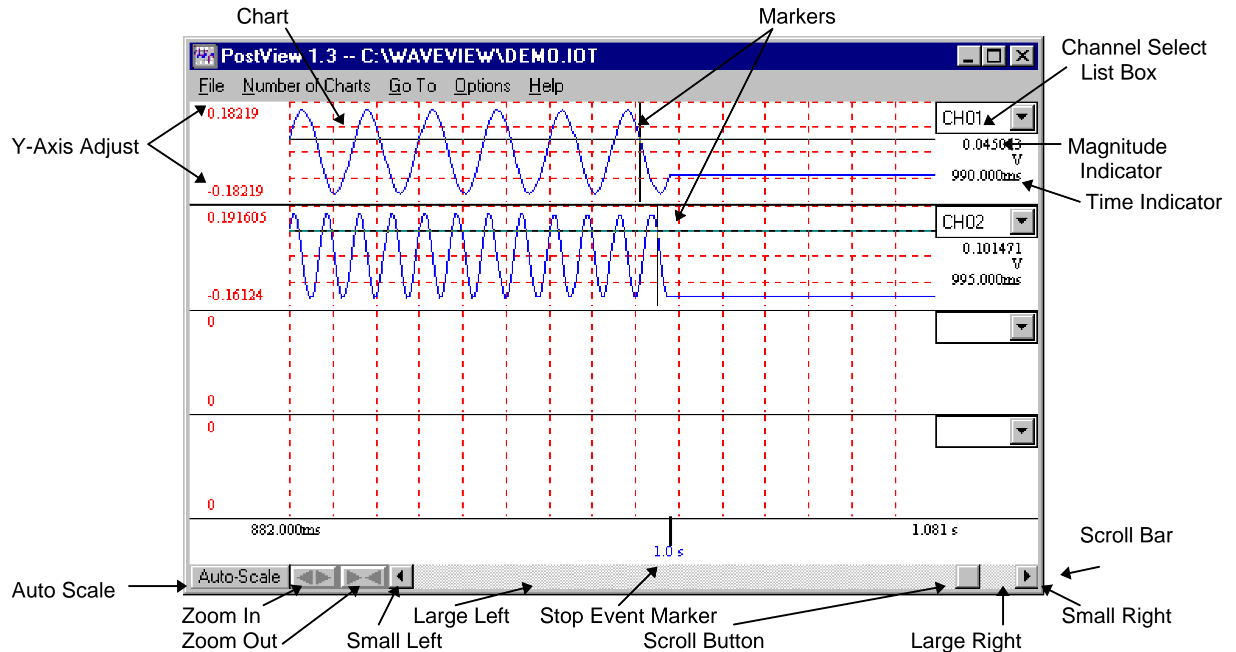
Help

The Help menu provides 3 basic functions.

Help Menu Items and Descriptions	
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.

PostView Display

The PostView display has several indicators as shown in the next figure.



PostView Indicators and Descriptions	
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.
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. • 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 you 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 WaveView.
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 you 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 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.
Stop Event Marker	The Stop Event Marker on the time axis shows the location of the stop point.

Notes

Using WaveBook/512 with C 6

This chapter describes the use of the C language with the **standard** API to develop a basic data acquisition program. For additional functions of the standard API, refer to chapter 10. **Note**: The WaveBook system includes full-featured DOS and Windows software drivers for C, QuickBASIC (chapter 7), Turbo Pascal (chapter 8), and Visual Basic (chapter 9). The enhanced API (for C, Visual Basic and Delphi) is described in chapters 11 and 12.

The same program examples are used for all versions of C, both DOS/Windows and Borland/Microsoft C. The only differences are the files used to create an executable version of the example and the command used to create this executable. The following sections describe different types of C support.

Borland C for DOS

The Borland C for DOS support files are located in the WAVEBOOK\DOS\BC directory. These files include the WBK.H header file, WBKL.LIB large model library, MAKEBC.BAT example batch file and ADCEX*.C example source files. WBK.H, which contains WaveBook function prototypes and definitions, should be referenced at the top of each file that makes calls to the WaveBook driver using the #include C pre-processor command. Each of the ADCEX*.C examples references WBK.H. The WBKL.LIB library should be included as part of the program's project or included explicitly by the linker. This library is a large model library, so all source files should be compiled using the large model flag (-ml) as demonstrated in the MAKEBC.BAT batch file.

Assuming Borland C 3.x or greater has been installed properly, ADCEX1.C can be compiled and linked using the following command line: `bcc -ml adcex1.c wbkl.lib.` Type `MAKEBC` at the DOS prompt from the WAVEBOOK\DOS\BC directory to use the MAKEBC.BAT batch file to compile and link all the example programs.

Microsoft C for DOS

The Microsoft C for DOS support files are located in the WAVEBOOK\DOS\MSC directory. These files include the WBK.H header file, WBKL.LIB large model library, MAKEMSC.BAT example batch file and ADCEX*.C example source files. WBK.H, which contains WaveBook function prototypes and definitions, should be referenced at the top of each file that makes calls to the WaveBook driver using the #include C pre-processor command. Each of the ADCEX*.C examples references WBK.H. The WBKL.LIB library should be included as part of the programs project or included explicitly by the linker. This library is a large model library, so all source files should be compiled using the large model flag (/AL) as demonstrated in the MAKEMSC.BAT batch file.

Assuming Microsoft C 6.x or greater has been installed properly, ADCEX1.C can be compiled and linked using the following command line: `cl /AL adcex1.c wbkl.lib.` Type `MAKEMSC` at the DOS prompt from the WAVEBOOK\DOS\MSC directory to use the MAKEMSC.BAT batch file to compile and link all the example programs.

Borland C for Windows

The Borland C for WIN support files are located in the WAVEBOOK\WIN\C directory. These files include the WBK.H header file, WBK.LIB large model library, MAKEBCW.BAT example batch file and ADCEX*.C example source files. In addition to these files, the WBK.DLL dynamic link library, which should be in the Windows SYSTEM directory, will be used when a program is executed. WBK.H, which contains WaveBook function prototypes and definitions, should be referenced at the top of each file that makes calls to the WaveBook driver using the #include C pre-processor command. Each of the ADCEX*.C examples references WBK.H. The WBK.LIB library should be included as part of the programs project or included explicitly by the linker. This library is a large model library, so all source files should be compiled using the large model flag (-ml) as demonstrated in the MAKEBCW.BAT batch file.

Assuming Borland C 3.x or greater has been installed properly, ADCEX1.C can be compiled and linked using the following command line: `bcc -ml -W adcex1.c wbk.lib`. **Note**: this command line uses the -W option to create a Windows application which uses the Borland EasyWin libraries. Type `MAKEBCW` at the DOS prompt from the WAVEBOOK\WIN\C directory to use the MAKEBCW.BAT batch file to compile and link all the example programs.

Microsoft C for Windows

The Microsoft C for WIN support files are located in the WAVEBOOK\WIN\C directory. These files include the WBK.H header file, WBK.LIB large model library, MAKEMSCW.BAT example batch file and ADCEX*.C example source files. In addition to these files, the WBK.DLL dynamic link library, which should be in the Windows SYSTEM directory, will be used when a program is executed. WBK.H, which contains WaveBook function prototypes and definitions, should be referenced at the top of each file that makes calls to the WaveBook driver using the #include C pre-processor command. Each of the ADCEX*.C examples references WBK.H. The WBK.LIB library should be included as part of the programs project or included explicitly by the linker. This library is a large model library, so all source files should be compiled using the large model flag (-ml) as demonstrated in the MAKEMSC.BAT batch file.

Assuming Microsoft C 3.x or greater has been installed properly, ADCEX1.C can be compiled and linked using the following command line: `cl /AL /Mq adcex1.c wbk.lib`. Note that this command line uses the /Mq option to create a Windows application which uses the Microsoft QuickWin libraries. Type MAKEMSCW at the DOS prompt from the WAVEBOOK\WIN\C directory to use the MAKEMSCW.BAT batch file to compile and link all the example programs.

Initializing WaveBook Communications

To communicate with the WaveBook hardware, the driver must initialize the hardware and itself through the function call `wbkInit`. Most WaveBook commands cannot be accepted until the WaveBook is initialized using `wbkInit`. The `wbkSetDefaultProtocol` is one of the functions that can be called before `wbkInit` can be used to set the most efficient printer port protocol to use. The examples use the slowest 4-bit protocol, but this can be changed to another protocol if the `wbktest` program determines that your system supports one. To close a WaveBook session, use the function `wbkClose`. The following program skeleton shows the usage of the `wbkInit` and `wbkClose` commands. Each of the example programs described later in this chapter initialize and close the WaveBook in this fashion, so this code has been removed from those descriptions.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "wbk.h"

void
main(void)
{
     clrscr();
     printf("\nThis is the beginning of my program.\n");

     // Perform non-WaveBook tasks here.

     // Start a WaveBook session.
     wbkSetDefaultProtocol(wbkProtocol4);
     wbkInit(LPT1, 7);

     // Perform WaveBook and non-WaveBook tasks here.

     //Close  the WaveBook
     wbkClose();
}
```

Error Handling

The WaveBook driver has a built-in facility for handling run-time errors. When an error is detected in the driver, control is automatically vectored to a default error handler which prints an error message

and halts operation. Optionally, the user can provide an error handling routine that will be executed instead of the default handler whenever an error is detected by the driver. When the driver calls the error handler, the error code is passed as a parameter. This error code can be used to decide upon the action to take or it can be ignored. The following program fragment shows an example of a user error handler:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "wbk.h"

void _far _pascal    myhandler(int error_code);

void
main(void)
{
     // Perform non-WaveBook tasks here.

     // Set error handler and initialize WaveBook
     wbkSetErrHandler(myhandler);
     wbkSetDefaultProtocol(wbkProtocol4);
     wbkInit(LPT1, 7);

     // Perform WaveBook and non-WaveBook tasks here.

     //Close and exit
     wbkClose();
}

// This is the user error handler routine.
void  _far  _pascal
myhandler(int error_code)
{
     // Put your error handler code here.

     printf("\nError! Program aborted\nWaveBook Error: 0x%x\n",error_code);
     exit(1);
}
```

One-Step Analog Input

The following excerpts are from the example program ADCEX1.C. This program demonstrates the use of the one-step WaveBook driver functions. These functions are the easiest to use but lack some flexibility. For additional flexibility, the custom analog input functions, discussed later, should be used.

Define constants and a data buffer for the collected data.

```
#define  CHANS       8
#define  SCANS       9
#define  FREQ        1000.0
#define  GAIN        WgcX1
#define BIPOLAR 1
int _huge buf[CHANS * SCANS]
```

wbkRd in this example will get 1 sample from channel 1 at a gain of ×1 (GAIN) in bipolar mode. The value will be returned in the variable sample.

```
wbkRd(1, &sample, GAIN, BIPOLAR);
```

wbkRdN is used to get 9 (SCANS) samples from channel 1 at a gain of ×1 in bipolar mode. This command requires a trigger to be satisfied before the data is collected. A trigger source of Software will start the acquisition immediately. The frequency of data collection is set to 1KHz (FREQ). The data will be returned in the integer array buf.

```
wbkRdN(1, buf, SCANS, WtsSoftware, 0.0f, FREQ, GAIN, BIPOLAR);
```

wbkRdScan collects an entire scan of data comprised of multiple channels starting with channel 1 and ending with channel 8 (CHANS). With all the channels to a gain of X! (GAIN) and bipolar mode.

```
wbkRdScan(1, CHANS, buf, GAIN, BIPOLAR);
```

wbkRdScanN collects 9 (SCANS) scans consisting of multiple channels. Like wbkRdScan, this command reads from channels 1 to 8 at the same gain and unipolar/bipolar setting. Since multiple scans are being collected, a trigger source and timebase are required. The arguments shown set the trigger source to software causing an immediate trigger and a sample frequency of 1KHz (FREQ). And like wbkRdN, a trigger source and timebase are required and set to WtsSoftware and 1KHz (FREQ) respectively.

```
wbkRdScanN(1, CHANS, buf, SCANS, WtsSoftware, 0.0f, FREQ, GAIN, BIPOLAR);
```

The results are then printed.

```
printf("\nResults of RdScanN:");
for(i = 0; i  < CHANS; i++) {
   printf("\nChannel %2d Data:", i + 1);
   for(j = 0; j  < SCANS; j++) printf(" %6d",buf[(j * CHANS) + i]);
}
```

Low-Level Analog Input

The following excerpts are from the example program ADCEX2.C. This program shows several examples of the lowest level WaveBook driver functions. These functions are more complex than the high level functions but allow the greatest flexibility.

Define constants and a data buffer for the collected data.

```
#define CHANS            8
#define SCANS            10
#define BLOCK            6
#define FREQ             5.0
int _huge buf[CHANS * SCANS];
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

The wbkSetMux command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

The pre- and post-trigger sample frequencies are then set. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

wbkGetFreq returns the present settings for the pre- and post-trigger frequencies.

```
   wbkGetFreq(&preTrigFreq, &postTrigFreq);
   printf("Result wbkGetFreq: pre-trigger=%.2fHz, post-trigger=%.2fHz\n",<R>
preTrigFreq, postTrigFreq);
```

Set the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0f);
```

The system is armed to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm();
```

The system is then triggered for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig();
```

Finally, perform a foreground transfer from the WaveBook of size BLOCK. The data is then printed on the screen. The transfers will continue until the acquisition is no longer active.

```
do {
   // Read BLOCK scans from the hardware with cycle mode off,
   // updateSingle on and foreground enabled
   wbkBufferTransfer(buf, BLOCK, 0, 1, 1, &active, &retCount);

   // Print results
   printf("\nResult wbkBufferTransfer: retCount=%lu, active=%d", retCount,
      (int)active);
   for(i = 0; i < retCount; i++) {
      printf("\nScan %5d:", i + 1);
      for(j = 0; j < CHANS; j++) {
         printf(" %6d", buf[i * CHANS + j]);
      }
   }
} while (active != 0);
```

Accessing the High-Speed Digital Input Port

The following excerpts are from the example program ADCEX3.C. This program shows how to collect analog and high speed digital signals concurrently, in the same scan.

Define constants and a data buffer for the collected data.

```
#define FREQ        5
#define SCANS      10
#define  CHANS      3
int _huge buf[CHANS * SCANS];
```

To create a channel scan of non-sequential channels with independent gain and unipolar/bipolar settings, the arrays of channel parameters must be created and passed to wbkSetScan. Channel 0 is the high speed digital port. When added to the channel scan, the high speed digital port is scanned synchronously with the analog signals.

```
chans[0]=0;          // high speed digital channel
chans[1]=5;          // analog channel 5
chans[2]=8;          // analog channel 8
```

The following lines set all of the gains and unipolar/bipolar settings to the same setting. Your program can assign each channel to a different value.

```
for(i=0; i < CHANS; i++) {
     gains[i]=WgcX1;        // unity gain
     polarities[i]=1;       // bipolar
}
```

The following line gets the driver version and prints the information.

```
wbkGetDriverVersion(&version);
printf("Using driver version %f\n\n", 0.01 * version);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

Setup the scan by passing the channel configuration arrays to the wbkSetScan command.

```
wbkSetScan(chans, gains, polarities, CHANS);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetHardwareTrig(WtsSoftware, 0.0f);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm();
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the

internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig();
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete.

```
wbkBufferTransfer(buf, SCANS, 0, 1, 1, &active, &retCount);
```

The following lines print the transferred data.

```
printf("Results of BufferTransfer:\n");
printf("Scan   Digital_ch_0  Analog_ch_5  Analog_ch_8");
for(i = 0; i < retCount; i++) {
   // get the upper (valid) 8 bits of the digital input
   buf[CHANS * i] = (unsigned int)buf[CHANS * i] >> 8;
   printf("\n %2d   ", i + 1);
   for(j = 0; j < CHANS; j++) printf("  %6d      ", buf[CHANS * i + j]);
}
```

Background Processing of Analog Input

The following excerpts are from the example program ADCEX4.C. This program shows how to collect analog samples and transfer them into the PC's memory in the background. Once the background acquisition is configured and armed, your program can perform other operations concurrently with the background data collection. The foreground program can use the wbkGetBackStat command to periodically check on the status of the acquisition.

For performing acquisition that are greater in size than the allocated buffer, the background operation can be set to Cycle mode which will wrap around in the allocated buffer as it becomes full. In this mode, your program must monitor the background and transfer the data out of the allocated buffer before the background operation overwrites it.

Define constants and a data buffer for the collected data.

```
#define CHANS            8
#define SCANS            9
#define FREQ             2.0
int _huge buf[CHANS * SCANS];
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

Set the scan configuration.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the post-trigger scan rates. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetHardwareTrig(WtsSoftware, 0.0f);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm();
```

The following line sets up a background transfer. Regardless of the state of the acquisition, the program will immediately return from this function call and proceed to the next line. As the data is collected by the WaveBook, it is automatically transferred to the buffer buf.

```
wbkBufferTransfer(buf, SCANS, 0, 1, 0, &active, &retCount);
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig();
```

Although your program can begin processing other tasks at this point, our example program simply monitors the background until the user hits a key or the acquisition is complete.

```
while (active) {
   wbkGetBackStat(&active, &retCount);
   printf("Transfer in progress: %21d scans acquired.\r", retCount);
}
```

The following lines print the collected data.

```
printf("Data acquired:\n");
for(i = 0; i < CHANS; i++) {
   printf("\nChannel %2d Data:", i + 1);
   for (j = 0; j < retCount; j++) printf(" %6d", buf[i + j * CHANS]);
}
```

Complex Triggering

The following excerpts are from the example program ADCEX5.C. This program shows how to setup a complex trigger where more than one channel can be combined in a logical trigger equation. The acquisition will start on a rising-edge of channel 1 at 2 volts OR a falling edge on channel 2 at 3 volts.

Define constants and a data buffer for the collected data.

```
#define  FREQ     1000
#define  SCANS    9
#define  CHANS    3
#define  NUM_TRIG 2
int _huge buf[CHANS * SCANS];
```

The following lines are definitions and initialization for the variables used for setting up the trigger equation. The variable chans_tr is an array of channels used in the trigger equation. Channels 1 and 2 are specified. The variable gain_tr is an array that holds the gains for channels 1 and 2. In this case they are both set to X1. Channels 1 and 2 can also be a part of the scan group with the same or different gain assignments. The variable polarity_tr is an array that holds the unipolar/bipolar settings for channels 1 and 2. The variable rising is an array that holds the edge settings for channels 1 and 2. In this case, channel 1 triggers on the rising edge, while channel 2 triggers on the falling edge. The variables levels and hysteresis are arrays that hold the voltage thresholds and hysteresis settings for channels 1 and 2, respectively. The variable opstr holds the Boolean operator for the trigger equation. The + sign indicates an OR operator between channels 1 and 2.

```
unsigned int     chans_tr[NUM_TRIG] =   { 1,          2       };
unsigned char    gains_tr[NUM_TRIG] =   { WgcX1,      WgcX1   },
                 polarity_tr[NUM_TRIG] ={ 1,          1       },
                 rising[NUM_TRIG] =     { WctRisingEdge,WctFallingEdge  };
float            levels[NUM_TRIG] =     { 2.0f,       3.0f    },
                 hysteresis[NUM_TRIG] = { 0.1f,       0.1f            };
char             *opstr = "+";
```

The previous definitions create the following trigger setup:

System Trigger = (CH1 @ ×1, bipolar, rising edge through 2.0V with 0.1V hyst) OR (CH2 @ ×1, bipolar, falling edge through 3.0 V with 0.1 V hyst)

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

Set the scan configuration.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

wbkSetFreq(1.0, FREQ);

The following lines notify the user of the system's status then setup the complex trigger.

<M>printf<D>("Waiting for complex trigger at channels 1 and 2...\n\n");

wbkSetComplexTrig(chans_tr, gains_tr, polarity_tr, rising, levels, hysteresis, NUM_TRIG, opstr);

This command arms the system to acquire data. Since no pre-trigger scans were configured, no data will be available until the trigger is satisfied.

wbkArm();

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete. If the trigger is not satisfied within the programmed time-out, the driver will return control to the program. If you do not want your program to "hang" until the trigger is satisfied, it is recommended that a background transfer be used. Once your program initiates a background transfer, control is passed back to your program to perform other tasks while waiting for a trigger or collecting data.

```
wbkBufferTransfer(buf, SCANS, 0, 1, 1, &active, &retCount);
```

Print the transferred data.

```
printf("Results of BufferTransfer:\n");
for(i = 0; i < CHANS; i++)  {
   printf("\nChannel %2d Data:", i + 1);
   for (j = 0; j < retCount; j++) printf(" %6d", buf[i + j * CHANS]);
}
```

Pre- and Post-Trigger Acquisitions

The following excerpts are from the example program ADCEX6.C. This program shows how to setup and process acquisitions with both pre- and post-trigger scans.

Define constants and a data buffer for the collected data.

```
#define  CHANS       4
#define  PRE_SCANS   5
#define  POST_SCANS  9
#define  PRE_FREQ    100.0
#define  POST_FREQ   200.0
#define  BLOCK       (PRE_SCANS + POST_SCANS)
int _huge buf[BLOCK * CHANS];
```

Define the channels sequenceof the acquisition.

```
chans[0] = 1;           // channel numbers
chans[1] = 3;
chans[2] = 5;
chans[3] = 7;
for(i = 0; i < CHANS; i++) {
   gains[i] = WgcX1;    // unity gain
   polarities[i] = 1;   // bipolar
}
```

Enable data packing

```
wbkSetDataPacking(1);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamPrePost-specifies that both pre- and post-trigger scans are to be collected. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamPrePost, PRE_SCANS, POST_SCANS);
```

Set the scan configuration.

```
wbkSetScan(chans, gains, polarities, CHANS);
```

This command sets the pre- and post-trigger sample frequencies.

```
wbkSetFreq(PRE_FREQ, POST_FREQ);
```

Set the trigger source to an analog trigger on channel 1 at 2 volts

```
wbkSetTrigAnalog(1, WgcX1, 1, WctRisingEdge, 2.0, 0.1);
```

This command arms the system to acquire data. Since pre-trigger scans are to be collected, scans will be immediately available for transfer into the PC's memory.

```
wbkArm();
```

When pre-trigger scans are included in the acquisition, scans begin to be acquired the moment the system is armed. Scans will continue to be acquired until the trigger is satisfied and the post-trigger is complete. Your application program must transfer the acquired data into a buffer in the PC as it is collected. Until the trigger occurs, your application must be prepared to accept data continuously, potentially far in excess of the sum of the specified pre-trigger and post-trigger scan counts. This is best accomplished by setting up a background transfer in cycle mode which will automatically transfer the scans as they are collected and wrap the buffer as it becomes full. The following line sets up a background transfer of the acquired scans into buf. Cycle mode is turned on, allowing the buffer to wrap around as it becomes full.

```
wbkBufferTransfer(buf, BLOCK, 1, 0, 0, &active, &retCount);
```

The following lines monitor the background operation, waiting for the acquisition to be complete.

```
while (active && !kbhit()) {
        wbkGetBackStat(&active, &retCount);
        printf("Transfer in progress: %2ld scans acquired.\r", retCount);
}
printf("\acquisition complete.\n\n");
```

The following line unpacks the data so that each sample occupies an integer.

```
wbkBufferUnpack(buf, buf, BLOCK, CHANS, retCount);
```

Since the buffer has potentially wrapped around, the earliest data is not at the beginning of the buffer. The following line reorganizes the buffer so that the 1st pre-trigger scan occupies the 1st buffer location and the last post-trigger scan occupies the last buffer location.

```
wbkBufferRotate(buf, BLOCK, CHANS, retCount);
```

The following lines print the acquired data.

```
printf("Pre-trigger data acquired:\n");
for(i = 0; i < CHANS; i++) {
   printf("\nChannel %2d Data:", chans[i]);
   for (j = 0; j < PRE_SCANS; j++) printf(" %6d", buf[i + j * CHANS]);
}
printf("\nPost-trigger data acquired:\n");
for(i = 0; i < CHANS; i++) {
   printf("\nChannel %2d Data:", chans[i]);
   for (j = PRE_SCANS; j < BLOCK; j++) printf(" %6d", buf[i + j * CHANS]);
}
```

Buffer Management

The following excerpts are from the example program ADCEX7.C found in the WaveBook directory of your hard drive. 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.

Define constants and a data buffer for the collected data.

```
#define  CHANS    8
#define  SCANS    20000
#define  BLOCK    1000
#define  FREQ     5000.0

int _huge buf0[CHANS * BLOCK];
int _huge buf1[CHANS * BLOCK];
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

The wbkSetMux command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

Next, the pre- and post-trigger sample frequencies are set. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The trigger source is set to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0f);
```

This system is then armed to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm();
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig();
```

Start the data transfer into the first buffer

```
transferBuf = buf0;
processBuf = buf1;
wbkBufferTransfer(transferBuf, BLOCK, 0, 0, 0, &tmpActive, &tmpRetCount);
```

The contents of the two buffers are swapped. The loop continues until the acquisition completes or the buffer fills.

```
do {
   // Swap the previous and next buffer pointers
   tmpBuf = processBuf;
   processBuf = transferBuf;
   transferBuf = tmpBuf;
```

Wait for the acquisition to go inactive or the buffer to be filled.

```
  do {
     wbkGetBackStat(&active, &retCount);
  } while (active && (retCount < BLOCK));
```

If the previous acquisition is still active, another transfer into the next buffer begins.

```
   if (active) {
      wbkBufferTransfer(transferBuf, BLOCK, 0, 0, 0, &tmpActive, &tmpRetCount);
   }
```

The following commands average the data in the process buffer and print the results.

```
  if (retCount) {
     // Average the readings in the process buffer and print the results
     for(j = 0; j < CHANS; j++) {
        totals[j] = 0;
     }
     for(i = 0; i < retCount; i++) {
        for(j = 0; j < CHANS; j++) {
           totals[j] += processBuf[i * CHANS + j];
        }
     }
     printf("Averages:");
     for(j = 0; j < CHANS; j++) {
```

```
            printf(" %6.3f", (5.0 * totals[j]) / (32768.0 * retCount));
         }
         printf("\n");
      }
   } while (active);
```

Direct-to-Disk

The following excerpts are from the example program ADCEX8.C. This example reads multiple scans from multiple channels and writes the data directly to a disk file.

Define constants and a data buffer for the collected data.

```
#define  CHANS    2
#define  SCANS    100000L
#define  FREQ     10000.0
#define  BLOCK    2000     // CHANS * BLOCK must be a multiple of 4

 int _huge buf[CHANS * BLOCK];
```

Enable data packing

```
wbkSetDataPacking(1);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

The wbkSetMux command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

The pre- and post-trigger sample frequencies are next set. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The wbkSetTrigHardware command sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0f);
```

Arm the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm();
```

Create a filename ADCEX8.BIN for the file that will hold the data with the wbkSetDiskFile command. No pre-write is used.

```
wbkSetDiskFile("adcex8.bin", WdfWriteFile, 0L);
```

Start reading data in the background mode with cycle mode on and updateSingle off

```
wbkBufferTransfer(buf, BLOCK, 1, 0, 0, &active, &retCount);
```

The next line triggers the system for data collection.

```
wbkSoftTrig();
```

The following commands monitor the progress of the background transfer. The program prints a running total of the number of scans acquired.

```
while (active) {
   wbkGetBackStat(&active, &retCount);
   printf("Transfer in progress: %21d scans acquired.\r", retCount);
}
```

Once the transfer has finished, a completion message is printed.

```
printf("\nAcquisition complete.\n\n");
```

After the acquisition has finished, the collected binary data will be converted to ascii format for display as a text file. The process starts with reading binary data from the ADCEX8.BIN file. If the file cannot be opened, an error is issued

```
printf("Converting adcex8.bin to adcex8.txt...\n");
in = open("adcex8.bin", O_RDONLY | O_BINARY);
if (in ==  -1) {
   printf("Unable to open adcex8.bin\n");
   exit(1);
}
```

Next, a text file is created to hold the converted data. If the file cannot be created, an error is issued.

```
fout = fopen("adcex8.txt", "wt+");
if (fout == NULL) {
   printf("Unable to open adcex8.txt\n");
   exit(1);
}
```

The binary data is converted to ascii and transferred to the text file. Errors are generated if the program cannot read from ADCEX8.BIN or write to ADCEX8.TXT.

Convert BLOCK unpacked scans to packed bytes

```
do {
   scanCount = BLOCK;
   sampleCount = scanCount * CHANS;
   wordCount = sampleCount * 3 / 4;
   byteCount = sizeof(int) * wordCount;
```

Read the packed bytes from the input file and get the number of bytes actually read

```
   byteCount = read(in, buf, byteCount);
   if (byteCount == -1) {
      printf("Unable to read from adcex8.bin\n");
      exit(1);
   }
```

Convert the number of bytes read from packed bytes to unpacked scans

```
   wordCount = byteCount / sizeof(int);
   sampleCount = wordCount * 4 / 3;
   scanCount = sampleCount / CHANS;
```

Unpack the packed data using the same buffer. This command can be called even if the WaveBook if not online or connected.

```
   wbkBufferUnpack(buf, buf, BLOCK, CHANS, scanCount);
```

The contents of the text file containing the data are displayed.

```
   for (i = 0; i < scanCount; i++) {
      for (j = 0; j < CHANS; j++) {
         // Send a tab between channels and a newline after each scan
         if (j < CHANS - 1) {
            termChar = '\t';
         } else {
            termChar = '\n';
         }
```

The voltage values are calculated and printed.

```
         voltage = (float)buf[i * CHANS + j] * 5.0f / 32768.0f;
         if (fprintf(fout, "%.3f%c", voltage, termChar) == EOF) {
            printf("Unable to write to adcex8.txt\n");
            exit(1);
         }
      }
   }
```

The program prints a character to indicate that it is still active.

printf(".");

```
} while (byteCount > 0);   // A byteCount of 0 indicates end-of-file
```

The files are closed

```
close(in);
fclose(fout);
```

A completion message is printed.

```
printf("complete.\n");
```

Sample Programs

ADCEX1.C

```
// This example demonstrates the use of the WaveBook's one-step
// acquisition functions and user error handling.
// Functions used:
//    wbkRd(chan, sample, gain, polarity);
//    wbkRdN(chan, buf, count, trigger, level, freq, gain, polarity);
//    wbkRdScan(startChan, endChan, buf, gain, polarity);
//    wbkRdScanN(startChan, endChan, buf, count, trigger, level, freq,
//               gain, polarity);
//    wbkSetErrHandler(wbkErrorHandler);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include <stdlib.h>
#include "wbk.h"
#define  CHANS       8
#define  SCANS       9
#define  FREQ        1000.0
#define  GAIN        WgcX1
#define  BIPOLAR     1
void _far _pascal myhandler(int error_code);
int _huge buf[CHANS * SCANS];
void
main(void)
{
   unsigned int   i, j;
   int            sample;
printf("\nADCEX1.C\n");
// Set error handler and initialize WaveBook
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
// Get a single sample from a single channel
   wbkRd(1, &sample, GAIN, BIPOLAR);
// Print result
   printf("Result of Rd: %6d\n", sample);
// Get multiple samples from a single channel, triggered by a software trigger
   wbkRdN(1, buf, SCANS, WtsSoftware, 0.0f, FREQ, GAIN, BIPOLAR);
// Print results
   printf("Results of RdN:");
   for(i = 0; i < SCANS; i++) printf(" %6d", buf[i]);

   // Get a single sample from multiple channels
   wbkRdScan(1, CHANS, buf, GAIN, BIPOLAR);

   // Print results
   printf("\n\nResults of RdScan:\n");
   for(i = 0; i < CHANS; i++) printf("Channel %2d Data: %6d\n", i + 1, buf[i]);

   // Get multiple samples from multiple channels, triggered by a software trigger
   wbkRdScanN(1, CHANS, buf, SCANS, WtsSoftware, 0.0f, FREQ, GAIN, BIPOLAR);

   // Print results
   printf("\nResults of RdScanN:");
   for(i = 0; i < CHANS; i++) {
      printf("\nChannel %2d Data:", i + 1);
      for(j = 0; j < SCANS; j++) printf(" %6d",buf[(j * CHANS) + i]);
   }

   // Close and exit
   wbkClose();
}

void _far _pascal
myhandler(int error_code)
{
   printf("\nError! Program aborted\nWaveBook Error: 0x%02x\n",error_code);
   exit(1);
}
```

ADCEX2.C

```
// This example demonstrates the use of WaveBook's custom acquistion functions.
// Functions used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetMux(startChan, endChan, gain, polarity);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkGetFreq(preTrigFreq, postTrigFreq);
//    wbkGetPeriod(preTrigPeriod, postTrigPeriod);
//    wbkSetTrigHardware(source, level);
//    wbkArm();
//    wbkSoftTrig();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  CHANS    8
#define  SCANS    10
#define  BLOCK    6
#define  FREQ     5.0
int _huge buf[CHANS * BLOCK];
void
main(void)
{
   unsigned int   i, j;
   unsigned char  active;
   unsigned long  retCount;
   double         preTrigFreq, postTrigFreq;
   double         preTrigPeriod, postTrigPeriod;
printf("\nADCEX2.C\n\n");
// Initialize WaveBook
   wbkInit(LPT1, 7);
// Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);
// Set the scan configuration
   wbkSetMux(1, CHANS, WgcX1, 1);
// Set the post-trigger scan rates
   wbkSetFreq(1.0, FREQ);
// Get the pre-trigger and post-trigger scan rates in frequency and period
   wbkGetFreq(&preTrigFreq, &postTrigFreq);
   printf("Result wbkGetFreq: pre-trigger=%.2fHz, post-trigger=%.2fHz\n",
      preTrigFreq, postTrigFreq);
   wbkGetPeriod(&preTrigPeriod, &postTrigPeriod);
   printf("Result wbkGetPeriod: pre-trigger=%.0fns, post-trigger=%.0fns\n",
      preTrigPeriod, postTrigPeriod);
// Set the trigger source to a software trigger command
   wbkSetTrigHardware(WtsSoftware, 0.0f);
// Arm the acquisition
   wbkArm();
// Issue a software trigger command to the hardware
   wbkSoftTrig();

   do {
      // Read BLOCK scans from the hardware with cycle mode off,
      // updateSingle on and foreground enabled
      wbkBufferTransfer(buf, BLOCK, 0, 1, 1, &active, &retCount);

      // Print results
      printf("\nResult wbkBufferTransfer: retCount=%lu, active=%d", retCount,
         (int)active);
      for(i = 0; i < retCount; i++) {
         printf("\nScan %5d:", i + 1);
         for(j = 0; j < CHANS; j++) {
            printf(" %6d", buf[i * CHANS + j]);
         }
      }
   } while (active != 0);

   //Close and exit
   wbkClose();
}
```

ADCEX3.C

```
// This example takes multiple scans from hardware using a software trigger.
// Each scan includes the high speed digital I/O port (channel 0) and
// two analog channels: 5 and 8.
// Functions used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetScan(chans, gains, polarities, count);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetTrigHardware(source, level);
//    wbkArm();
//    wbkSoftTrig();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkGetDriverVersion(version);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  FREQ     5
#define  SCANS    10
#define  CHANS    3
int _huge buf[CHANS * SCANS];
void
main(void)
{
   unsigned int   i, j;
   unsigned int   version;
   unsigned int   chans[CHANS];
   unsigned char  gains[CHANS], polarities[CHANS];
   unsigned char  active;
   unsigned long  retCount;
   printf("\nADCEX3.C\n\n");
   // Scan sequence definition
   chans[0] = 0;        // high speed digital channel
   chans[1] = 5;        // analog channel 5
   chans[2] = 8;        // analog channel 8
   // Channel gains and polarities setting
   for(i = 0; i < CHANS; i++) {
      gains[i] = WgcX1;    // unity gain
      polarities[i] = 1;   // bipolar
   }
   // Get driver version
   wbkGetDriverVersion(&version);
   printf("Using driver version %f\n\n", 0.01 * version);
   // Initialize WaveBook
   wbkInit(LPT1, 7);
   // Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);
   // Set the scan configuration
   wbkSetScan(chans, gains, polarities, CHANS);
   // Set the post-trigger scan rates
   wbkSetFreq(1.0, FREQ);
   // Set the trigger source to a software trigger command
   wbkSetTrigHardware(WtsSoftware, 0.0f);
   // Arm the acquisition
   wbkArm();
   // Issue a software trigger command to the hardware
   wbkSoftTrig();
   // Read SCANS scans from the hardware with cycle mode off,
   // updateSingle on and foreground enabled
   wbkBufferTransfer(buf, SCANS, 0, 1, 1, &active, &retCount);
   // Print results
   printf("Results of BufferTransfer:\n");
   printf("Scan   Digital_ch_0  Analog_ch_5  Analog_ch_8");
   for(i = 0; i < retCount; i++) {
      // get the upper (valid) 8 bits of the digital input
      buf[CHANS * i] = (unsigned int)buf[CHANS * i] >> 8;
      printf("\n %2d   ", i + 1);
      for(j = 0; j < CHANS; j++) printf("  %6d      ", buf[CHANS * i + j]);
   }
   //Close and exit
   wbkClose();
}
```

ADCEX4.C

```
// This example reads scans of multiple channels in the background mode
// and uses a software trigger to start the acquisition.
// Functions used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetMux(startChan, endChan, gain, polarity);
//    wbkSetTrigHardware(source, level);
//    wbkArm();
//    wbkSoftTrig();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkGetBackStat(active, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  CHANS    8
#define  SCANS    9
#define  FREQ     2.0
int _huge buf[CHANS * SCANS];

void
main(void)
{
   unsigned int   i, j;
   unsigned char  active;
   unsigned long  retCount;

   printf("\nADCEX4.C\n\n");

   // Initialize WaveBook
   wbkInit(LPT1, 7);

   // Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);

   // Set the scan configuration
   wbkSetMux(1, CHANS, WgcX1, 1);

   // Set the post-trigger scan rates
   wbkSetFreq(1.0, FREQ);

   // Set the trigger source to a software trigger command
   wbkSetTrigHardware(WtsSoftware, 0.0f);

   // Arm the acquisition
   wbkArm();

   // Start reading data in the background mode with cycle mode off
   // and updateSingle on
   wbkBufferTransfer(buf, SCANS, 0, 1, 0, &active, &retCount);

   // Issue a software trigger command to the hardware
   wbkSoftTrig();

   // Monitor the progress of the background transfer
   while (active) {
      wbkGetBackStat(&active, &retCount);
      printf("Transfer in progress: %2ld scans acquired.\r", retCount);
   }
   printf("\nAcquisition complete.\n\n");

   // Print results
   printf("Data acquired:\n");
   for(i = 0; i < CHANS; i++) {
      printf("\nChannel %2d Data:", i + 1);
      for (j = 0; j < retCount; j++) printf(" %6d", buf[i + j * CHANS]);
   }

   // Close and Exit
   wbkClose();
}
```

ADCEX5.C

```
// This example takes multiple scans from hardware using a complex analog
// trigger.  The acquisition will start on a rising-edge of channel 1 at
// 2 volts OR a falling edge on channel 2 at 3 volts.
// Functions used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetMux(startChan, endChan, gain, polarity);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetTrigComplex(chans, gains, polarities, rising, levels,
//                      hysteresis, count, opstr);
//    wbkArm();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  FREQ     1000
#define  SCANS    9
#define  CHANS    3
#define  NUM_TRIG 2
int _huge buf[CHANS * SCANS];
void
main(void)
{
   unsigned int   i, j;
   unsigned char  active;
   unsigned long  retCount;

   // Initialize the complex trigger arrays for a rising-edge on channel 1
   // at 2 volts OR a falling-edge on channel 2 at 3 volts
   unsigned int   chans_tr[NUM_TRIG] =    { 1,              2              };
   unsigned char  gains_tr[NUM_TRIG] =    { WgcX1,          WgcX1          },
                  polarity_tr[NUM_TRIG] = { 1,              1              },
                  rising[NUM_TRIG] =      { WctRisingEdge,  WctFallingEdge };
   float          levels[NUM_TRIG] =      { 2.0f,           3.0f           },
                  hysteresis[NUM_TRIG] =  { 0.1f,           0.1f           };
   char           *opstr = "+";

   printf("\nADCEX5.C\n\n");

   // Initialize WaveBook
   wbkInit(LPT1, 7);

   // Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);

   // Set the scan configuration
   wbkSetMux(1, CHANS, WgcX1, 1);

   // Set the post-trigger scan rates
   wbkSetFreq(1.0, FREQ);

   // Set the trigger source to the complex trigger previously defined
   printf("Waiting for complex trigger of channels 1 or 2...\n\n");
   wbkSetTrigComplex(chans_tr, gains_tr, polarity_tr, rising, levels,
      hysteresis, NUM_TRIG, opstr);

   // Arm the acquisition
   wbkArm();

   // Read SCANS scans from the hardware with cycle mode off,
   // updateSingle on and foreground enabled
   wbkBufferTransfer(buf, SCANS, 0, 1, 1, &active, &retCount);
   // Print results
   printf("Results of BufferTransfer:\n");
   for(i = 0; i < CHANS; i++)  {
      printf("\nChannel %2d Data:", i + 1);
      for (j = 0; j < retCount; j++) printf(" %6d", buf[i + j * CHANS]);
   }
   //Close and exit
   wbkClose();
```

ADCEX6.C

```
// This example demonstrates an acquisition made up of pre-trigger and
// post-trigger scans from multiple channels using a DSP-based analog
// trigger.  It also uses data packing and rotating.
// Functions used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetScan(chans, gains, polarities, chanCount);
//    wbkSetTrigAnalog(chan, gain, polarity, rising, level, hysteresis);
//    wbkArm();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount, retCount);
//    wbkBufferRotate(buf, scanCount, chanCount, retCount);
//    wbkGetBackStat(active, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  CHANS       4
#define  PRE_SCANS   5
#define  POST_SCANS  9
#define  PRE_FREQ    100.0
#define  POST_FREQ   200.0
#define  BLOCK       (PRE_SCANS + POST_SCANS)
int _huge buf[BLOCK * CHANS];
void
main(void)
{
   unsigned int   i, j;
   unsigned int   chans[CHANS];
   unsigned char  gains[CHANS], polarities[CHANS];
   unsigned char  active;
   unsigned long  retCount;
   printf("\nADCEX6.C\n");
   // Scan definition
   chans[0] = 1;           // channel numbers
   chans[1] = 3;
   chans[2] = 5;
   chans[3] = 7;
   for(i = 0; i < CHANS; i++) {
      gains[i] = WgcX1;    // unity gain
      polarities[i] = 1;   // bipolar
   }
   // Initialize WaveBook
   wbkInit(LPT1, 7);
   // Enable data packing
   wbkSetDataPacking(1);
   // Set the acquisition for pre/post-trigger mode and the scan counts
   wbkSetAcq(WamPrePost, PRE_SCANS, POST_SCANS);
   // Set the scan configuration
   wbkSetScan(chans, gains, polarities, CHANS);
   // Set the pre-trigger and post-trigger scan rates
   wbkSetFreq(PRE_FREQ, POST_FREQ);
   // Set the trigger source to an analog trigger on channel 1 at 2 volts
   wbkSetTrigAnalog(1, WgcX1, 1, WctRisingEdge, 2.0f, 0.1f);
   // Arm the acquisition
   wbkArm();
   // Start reading data in the background mode with cycle mode on
   // and updateSingle off
   wbkBufferTransfer(buf, BLOCK, 1, 0, 0, &active, &retCount);
   // Monitor the progress of the background transfer
   while (active) {
      wbkGetBackStat(&active, &retCount);
      printf("Transfer in progress: %2ld scans acquired.\r", retCount);
   }
   printf("\nAcquisition complete.\n");
   // Unpack the packed data using the same buffer
   wbkBufferUnpack(buf, buf, BLOCK, CHANS, retCount);
   // Rotate the unpacked data so that the earliest data starts at the
   // beginning of the buffer and the latest is at the end
   wbkBufferRotate(buf, BLOCK, CHANS, retCount);
   // Print results
```

```
   printf("Pre-trigger data acquired:\n");
   for(i = 0; i < CHANS; i++) {
      printf("\nChannel %2d Data:", chans[i]);
      for (j = 0; j < PRE_SCANS; j++) printf(" %6d", buf[i + j * CHANS]);
   }
   printf("\nPost-trigger data acquired:\n");
   for(i = 0; i < CHANS; i++) {
      printf("\nChannel %2d Data:", chans[i]);
      for (j = PRE_SCANS; j < BLOCK; j++) printf(" %6d", buf[i + j * CHANS]);
   }
   // Close and Exit
   wbkClose();
}
```

ADCEX7.C

```
// 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:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetMux(startChan, endChan, gain, polarity);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetTrigHardware(source, level);
//    wbkArm();
//    wbkSoftTrig();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//                      active, retCount);
//    wbkGetBackStat(active, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include "wbk.h"
#define  CHANS    8
#define  SCANS    20000
#define  BLOCK    1000
#define  FREQ     5000.0
int _huge buf0[CHANS * BLOCK];
int _huge buf1[CHANS * BLOCK];
void
main(void)
{
   unsigned int   i, j;
   unsigned char  active;
   unsigned long  retCount;
   unsigned char  tmpActive;
   unsigned long  tmpRetCount;
   long           totals[CHANS];
   int _huge      *tmpBuf;
   int _huge      *transferBuf;
   int _huge      *processBuf;
   printf("\nADCEX7.C\n\n");

   // Initialize WaveBook
   wbkInit(LPT1, 7);

   // Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);

   // Set the scan configuration
   wbkSetMux(1, CHANS, WgcX1, 1);

   // Set the post-trigger scan rates
   wbkSetFreq(1.0, FREQ);

   // Set the trigger source to a software trigger command
   wbkSetTrigHardware(WtsSoftware, 0.0f);

   // Arm the acquisition
   wbkArm();

   // Issue a software trigger command to the hardware
```

```
   wbkSoftTrig();

   // Start reading data into the first buffer
   transferBuf = buf0;
   processBuf = buf1;
   wbkBufferTransfer(transferBuf, BLOCK, 0, 0, 0, &tmpActive, &tmpRetCount);

   do {
      // Swap the previous and next buffer pointers
      tmpBuf = processBuf;
      processBuf = transferBuf;
      transferBuf = tmpBuf;

      // Wait for the acquisition to go inactive or the buffer to be filled
      do {
         wbkGetBackStat(&active, &retCount);
      } while (active && (retCount < BLOCK));

      // If the previous acquisition is still active, start another transfer
      // into the next buffer
      if (active) {
         wbkBufferTransfer(transferBuf, BLOCK, 0, 0, 0, &tmpActive, &tmpRetCount);
      }

      // Process the data into the process buffer
      if (retCount) {
         // Average the readings in the process buffer and print the results
         for(j = 0; j < CHANS; j++) {
            totals[j] = 0;
         }
         for(i = 0; i < retCount; i++) {
            for(j = 0; j < CHANS; j++) {
               totals[j] += processBuf[i * CHANS + j];
            }
         }
         printf("Averages:");
         for(j = 0; j < CHANS; j++) {
            printf(" %6.3f", (5.0 * totals[j]) / (32768.0 * retCount));
         }
         printf("\n");
      }
   } while (active);

   //Close and exit
   wbkClose();
}
```

ADCEX8.C

```
// This example reads multiple scans from multiple channels and writes the
// data directly a disk file in a packed format.
//
// Function used:
//    wbkSetAcq(mode, preTrigCount, postTrigCount);
//    wbkSetFreq(preTrigFreq, postTrigFreq);
//    wbkSetMux(startChan, endChan, gain, polarity);
//    wbkSetTrigHardware(source, level);
//    wbkArm();
//    wbkSoftTrig();
//    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,
//       active, retCount);
//    wbkGetBackStat(active, retCount);
//    wbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount, retCount);
//    wbkInit(lptPort, lptIntr);
//    wbkClose();
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <fcntl.h>
//#include <sys/stat.h>
#include "wbk.h"
#define  CHANS    2
#define  SCANS    100000L
```

```
#define  FREQ      10000.0
#define  BLOCK     2000     // CHANS * BLOCK must be a multiple of 4
int _huge buf[CHANS * BLOCK];
void
main(void)
{
   unsigned int   i, j;
   unsigned char  active;
   unsigned long  retCount;
   int            in;
   FILE           *fout;
   int            byteCount;
   unsigned int   wordCount, sampleCount, scanCount;
   char           termChar;
   float          voltage;

   printf("\nADCEX8.C\n\n");

   // Set error handler and initialize WaveBook
   wbkInit(LPT1, 7);

   // Enable data packing
   wbkSetDataPacking(1);

   // Set the acquisition to NShot on trigger and the post-trigger scan count
   wbkSetAcq(WamNShot, 0, SCANS);

   // Set the scan configuration
   wbkSetMux(1, CHANS, WgcX1, 1);

   // Set the post-trigger scan rate
   wbkSetFreq(1.0, FREQ);

   // Set the trigger source to a software trigger command
   wbkSetTrigHardware(WtsSoftware, 0.0f);

   // Arm the acquisition
   wbkArm();

   // Set the direct-to-disk filename with no pre-write
   wbkSetDiskFile("adcex8.bin", WdfWriteFile, 0L);

   // Start reading data in the background mode with cycle mode on
   // and updateSingle off
   wbkBufferTransfer(buf, BLOCK, 1, 0, 0, &active, &retCount);

   // Issue a software trigger command to the hardware
   wbkSoftTrig();

   // Monitor the progress of the background transfer
   while (active) {
      wbkGetBackStat(&active, &retCount);
      printf("Transfer in progress: %2ld scans acquired.\r", retCount);
   }
   printf("\nAcquisition complete.\n\n");

   // Close and Exit
   wbkClose();

   // Convert the binary file just read to a text file
   printf("Converting adcex8.bin to adcex8.txt\n");

   // Open the binary input file
   in = open("adcex8.bin", O_RDONLY | O_BINARY);
   if (in == -1) {
      printf("Unable to open adcex8.bin\n");
      exit(1);
   }

   // Open the text output file
   fout = fopen("adcex8.txt", "wt+");
   if (fout == NULL) {
      printf("Unable to open adcex8.txt\n");
      exit(1);
   }
```

```
   do {
      // Convert BLOCK unpacked scans to packed bytes
      scanCount = BLOCK;
      sampleCount = scanCount * CHANS;
      wordCount = sampleCount * 3 / 4;
      byteCount = sizeof(int) * wordCount;

      // Read the packed bytes from the input file and get the number
      // of bytes actually read
      byteCount = read(in, buf, byteCount);
      if (byteCount == -1) {
         printf("Unable to read from adcex8.bin\n");
         exit(1);
      }

      // Convert the number of bytes read from packed bytes to unpacked scans
      wordCount = byteCount / sizeof(int);
      sampleCount = wordCount * 4 / 3;
      scanCount = sampleCount / CHANS;

      // Unpack the packed data using the same buffer. This command
      // can be called even if the WaveBook if not online or connected.
      wbkBufferUnpack(buf, buf, BLOCK, CHANS, scanCount);

      // Write the scans read and unpacked to the text file
      for (i = 0; i < scanCount; i++) {
         for (j = 0; j < CHANS; j++) {
            // Send a tab between channels and a newline after each scan
            if (j < CHANS - 1) {
               termChar = '\t';
            } else {
               termChar = '\n';
            }

            // calculate and write out the voltage value
            voltage = (float)buf[i * CHANS + j] * 5.0f / 32768.0f;
            if (fprintf(fout, "%.3f%c", voltage, termChar) == EOF) {
               printf("Unable to write to adcex8.txt\n");
               exit(1);
            }
         }
      }

      // Print something so that the program doesn't appear locked
      printf(".");
   } while (byteCount > 0);   // A byteCount of 0 indicates end-of-file

   // Close the input and output files
   close(in);
   fclose(fout);

   printf("complete.\n");
}
```

Using WaveBook/512 with QuickBASIC 7

This chapter describes the use of the QuickBASIC language with the **standard** API to develop a basic data acquisition program. For additional functions of the standard API, refer to chapter 10. **Note**: The WaveBook system includes full-featured DOS and Windows software drivers for C (chapter 6), QuickBASIC, Turbo Pascal (chapter 8), and Visual Basic (chapter 9). The enhanced API (for C, Visual Basic and Delphi) is described in chapters 11 and 12.

There is a library, a quick library and a basic interface file located in the WAVEBOOK\DOS\QB directory. The Basic interface file, WBK.BI, must be included at the top of a QuickBASIC program using the `'$INCLUDE` command. This will allow QuickBASIC to know what WaveBook/512 functions and constants are available. The library WAVEBOOK.LIB must be included during the link process when creating a program from the DOS command line. The /NOE option of the linker may be necessary when linking the WaveBook/512 library. Alternatively, the quick library, WAVEBOOK.QLB can be used to access the WaveBook/512 from within the QuickBASIC environment. Use the `/L` option of QuickBASIC to load the appropriate Quick Library. See the QuickBASIC documentation for the various command line options.

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

Using Multiple Quick Libraries with QuickBASIC

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

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

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

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

Simple Analog Input

The following program, ADCEX1.BAS, shows the usage of several high level analog input routines.

This program will initialize the WaveBook hardware, then take readings from the the analog input channels in the base unit (not the expansion cards).

Every program begins with an INCLUDE directive which defines useful constants and declarations that are used throughout the program.

```
'$INCLUDE: 'wbk.bi'
```

For transporting data in and out of the WaveBook driver, arrays are dimensioned.

```
    DIM buf%(SCANS& * CHANNELS%)
DIM i%, j%
DIM sample%
DIM ret%
```

Although not necessary, this program includes an error handler which vectors program control to a user defined routine when a WaveBook error is detected. If no error handler is supplied, QuickBASIC will receive and handle the error, posting the error on the screen and terminating the program.

QuickBASIC provides an integer variable, ERR, which contains the error code of the most recent error. This variable can be used in user defined error handlers to detect the error source and take the appropriate action. The function QBwbkSetErrHandler tells QuickBASIC to assign ERR to a specific value when a WaveBook error is encountered.

The following line tells QuickBASIC to set ERR to 100 when a WaveBook error is encountered.

```
ret% = QBwbkSetErrHandler%(100)
```

The ON ERROR GOTO command is part of the QuickBASIC command set. It allows a user defined error handler to be provided, rather than the standard error handler that QuickBASIC uses automatically. The program uses ON ERROR GOTO to vector program control to the label ErrorHandler if an error is encountered.

```
ON ERROR GOTO ErrorHandler
```

The next line tells the driver library which LPT port or base address and what interrupt line is being used, then initializes the WaveBook hardware.

```
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

The next command will retrieve 1 sample at a gain of 1. The polarity setting set to bipolar.

```
ret% = QBwbkRd%(1, sample%, GAIN%, BIPOLAR%)
```

The results of the are then printed.

```
PRINT "Result  of Rd :", sample%
```

Next, 8 samples are acquired from channel 1 using software triggering. A 1000Hz sampling frequancy is used with unity gain for bipolar signals. The data returned data is stored in the integer array buffer. The results are then printed.

```
ret% = QBwbkRdN%(1, buf%(), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%, BIPOLAR%)

' Print results
PRINT "Results of RdN:"
PRINT "Channel 1 Data: ";
FOR i% = 0 TO SCANS& - 1
   PRINT TAB(i% * 7 + 17); buf%(i%);
NEXT i%
```

The following command will get 8 samples from a single channel. Channel 1 at a gain of X1 in bipolar mode is selected. This command requires a trigger to be satisfied before the data is collected. A trigger source of Software will start the acquisition immediately. The frequency of data collection is set to 1KHz. The data will be returned in the integer array buf.

```
ret% = QBwbkRdN%(1, buf%(), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%, BIPOLAR%)

' Print results
PRINT "Results of RdN:"
PRINT "Channel 1 Data: ";
FOR i% = 0 TO SCANS& - 1
   PRINT TAB(i% * 7 + 17); buf%(i%);
NEXT i%
```

The following command collects 1 sample from multiple channels. The following function arguments define a scan starting with channel 1 and ending with channel 8. This command sets all the channels to the same gain and unipolar/bipolar setting. The data is returned in the array buffer.

```
ret% = QBwbkRdScan%(1, CHANNELS%, buf%(), GAIN%, BIPOLAR%)

' Print results
PRINT "Results of RdScan:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data: "; buf%(i%)
NEXT i%
```

The following command collects multiple scans consisting of multiple channels. Like wbkRdScan, this command sets all the channels to the same gain and unipolar/bipolar setting. Since multiple scans are being collected, a trigger source and timebase is required. The arguments shown set the trigger source to Software causing an immediate trigger. The sample frequency of 1Khz.

```
ret% = QBwbkRdScanN%(1, CHANNELS%, buf%(), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%,
BIPOLAR%)
```

```
' Print results
PRINT "Results of RdScanN:"
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%
```

Low-Level Analog Input

The following excerpts are from the example program ADCEX2.BAS found in the WaveBook directory of your hard drive. This program shows several examples of the lowest level WaveBook driver functions. These functions are more complex than the high level functions but allow the greatest flexibility.

Every program begins with an INCLUDE directive which defines useful constants and declarations that are used throughout the program.

```
'$INCLUDE: 'wbk.bi'
```

Constants and arrays are defined.

```
CONST CHANNELS% = 8
CONST SCANS& = 10
CONST BLOCK% = 6
CONST FREQ# = 5#

DIM buf%(BLOCK% * CHANNELS%)

DIM i%, j%
DIM active%
DIM retCount&
DIM preTrigFreq#, postTrigFreq#
DIM preTrigPeriod#, postTrigPeriod#
DIM ret%
```

Set the error handler and initialize the WaveBook.

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = QBwbkSetFreq%(1#, FREQ#)
```

The following line shows the usage of the wbkGetFreq command which returns the present settings for the pre- and post-trigger frequencies.

```
ret% = QBwbkGetFreq%(preTrigFreq#, postTrigFreq#)
PRINT "Result of GetFreq: pre-trigger="; preTrigFreq#; "Hz, post-trigger=";
postTrigFreq#; "Hz"
```

```
ret% = QBwbkGetPeriod%(preTrigPeriod#, postTrigPeriod#)
PRINT "Result of GetPeriod: pre-trigger="; preTrigPeriod#; "ns, post-trigger=";
postTrigPeriod#; "ns"
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = QBwbkArm%
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = QBwbkSoftTrig%
```

A software trigger command is issued to the hardware

```
ret% = QBwbkSoftTrig%
```

The following lines perform a foreground transfer from the WaveBook of size BLOCK. The data is then printed on the screen. The transfers will continue until the acquisition is no longer active.

```
DO
   ' Read BLOCK scans from the hardware with cycle mode off,
   ' updateSingle on and foreground enabled
   ret% = QBwbkBufferTransfer%(buf%(), BLOCK%, 0, 1, 1, active%, retCount&)

   ' Print results
   PRINT "Result of BufferTransfer: retCount="; retCount&; " active="; active%
   FOR i% = 0 TO retCount& - 1
      PRINT "Scan"; i% + 1; "Data:";
      FOR j% = 0 TO CHANNELS% - 1
         PRINT TAB(j% * 7 + 17); buf%(i% * CHANNELS% + j%);
      NEXT j%
      PRINT
   NEXT i%
   PRINT
LOOP WHILE active% <> 0
Close the WaveBook and exit.
ret% = QBwbkClose%
```

Accessing the High-Speed Digital Input Port

The following excerpts are from the example program ADCEX3.BAS found in the WaveBook directory of your hard drive. This program shows how to collect analog and high-speed digital signals concurrently, in the same scan.

Every program begins with an INCLUDE directive which defines useful constants and declarations that are used throughout the program.

```
'$INCLUDE: 'wbk.bi'
```

Constants and arrays are defined.

```
CONST FREQ# = 5
CONST SCANS& = 10
CONST CHANNELS% = 3

DIM buf%(SCANS& * CHANNELS%)

DIM i%, j%
```

```
DIM active%
DIM retCount&

DIM version%
DIM chans%(CHANNELS%)
DIM gains%(CHANNELS%), polarities%(CHANNELS%)
DIM ret%
```

Set channel definitions.

```
    chans%(0) = 0     ' high speed digital channel
    chans%(1) = 5     ' analog channel 5
    chans%(2) = 8     ' analog channel 8
```

Set gains and polarities.

```
FOR i% = 0 TO CHANNELS% - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
NEXT i%
```

Set error handler and initialize WaveBook.

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
 ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

Setup the scan by passing the channel configuration arrays to the wbkSetScan command.

```
ret% = QBwbkSetScan%(chans%(), gains%(), polarities%(), CHANNELS%)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
 ret% = QBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
 ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
    ret% = QBwbkArm%
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
    ret% = QBwbkSoftTrig%
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete.

```
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 1, active%, retCount&)
```

The following lines print the transferred data.

```
 PRINT "Results of BufferTransfer:"
```

```
PRINT "                Digital_ch_0  Analog_ch_5  Analog_ch_8"
FOR i% = 0 TO retCount& - 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%
```

Close the WaveBook and exit.

```
ret% = QBwbkClose%
```

Background Processing of Analog Input

The following excerpts are from the example program ADCEX4.BAS found in the WaveBook directory of your hard drive. This program shows how to collect analog samples and transfer them into the PC's memory in the background. Once the background acquisition is configured and armed, your program can perform other operations concurrently with the background data collection. The foreground program can use the wbkGetBackStat command to periodically check the status of the acquisition.

For performing acquisitions that are greater in size than the allocated buffer, the background operation can be set to Cycle mode which will wrap around in the allocated buffer as it becomes full. In this mode, your program must monitor the background and transfer the data out of the allocated buffer before the background operation overwrites it.

Constants and arrays are first defined.

```
CONST CHANNELS% = 8
CONST SCANS& = 9
CONST FREQ# = 2

DIM buf%(SCANS& * CHANNELS%)

DIM i%, j%
DIM active%
DIM retCount&
DIM ret%
```

Set error handler and initialize WaveBook.

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

Set the scan's configuration.

```
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = QBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = QBwbkArm%
```

The scan starts taking readings in the background mode with cycle mode off and updateSingle on.

```
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 0, active%, retCount&)
```

A software trigger is issued.

```
ret% = QBbkSoftTrig%
```

Although your program can begin processing other tasks at this point, our example program simply monitors the background until the user hits a key or the acquisition is complete.

```
PRINT "Waiting for trigger."
WHILE retCount& = 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Triggered.  Transfer in progress."
WHILE active% <> 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Acquisition complete:"; retCount&; "scans acquired."
```

The following lines print the collected data.

```
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%
```

Close the WaveBook and exit.

```
ret% = QBwbkClose%
```

Complex Triggering

The following segments are from the example program ADCEX5.BAS found in the WaveBook directory of your hard drive. This program shows how to setup a complex trigger where more than one channel can be combined in a logical trigger equation.

Define constants and declarations and set up arrays.

```
 '$INCLUDE: 'wbk.bi'
CONST FREQ# = 1000
CONST SCANS& = 9
CONST CHANNELS% = 3
CONST NUMTRIG% = 2

DIM buf%(SCANS& * CHANNELS%)

DIM i%, j%
DIM active%
DIM retCount&
DIM chans%(NUMTRIG%)
DIM gains%(NUMTRIG%), polarity%(NUMTRIG%)
DIM rising%(NUMTRIG%)
DIM levels!(NUMTRIG%), hysteresis!(NUMTRIG%)
DIM opstr$
DIM ret%
```

The following lines are definitions and initialization for the variables used for setting up the trigger equation. The variable chans_tr is an array of channels used in the trigger equation. Channels 1 and 2 are specified. The variable gain_tr is an array that holds the gains for channels 1 and 2. In this case they are both set to X1. Channels 1 and 2 can also be a part of the scan group with the same or different gain assignments. The variable polarity_tr is an array that holds the unipolar/biploar settings

for channels 1 and 2. The variable rising is an array that holds the edge settings for channels 1 and 2. In this case, channel 1 triggers on the rising edge, while channel 2 triggers on the falling edge. The variables levels and hysteresis are arrays that hold the voltage thresholds and hysteresis settings for channels 1 and 2, respectively. The variable opstr holds the boolean operator for the trigger equation. The + sign indicates an OR operator between channels 1 and 2.

```
chans%(0) = 1
gains%(0) = WgcX1%
polarity%(0) = 1
rising%(0) = WctRisingEdge%
levels!(0) = 2
hysteresis!(0) = .1

chans%(1) = 2
gains%(1) = WgcX1%
polarity%(1) = 1
rising%(1) = WctFallingEdge%
levels!(1) = 3
hysteresis!(1) = .1

opstr$ = "+"
```

The previous definitions create the following trigger setup:

System Trigger = (CH1 @ X1, bipolar, rising edge through 2.0V with 0.1V hyst) OR (CH2 @ X1, bipolar, falling edge through 3.0V with 0.1V hyst)

The following lines set up the error handler and initialize WaveBook

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

To create a channel scan of non sequential channels with independent gain and unipolar/bipolar settings, the arrays of channel parameters must be created and passed to wbkSetScan.

```
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = QBwbkSetFreq%(1#, FREQ#)
```

The following lines notify the user of the system's status then setup the complex trigger.

```
PRINT "Waiting for complex trigger of channels 1 or 2..."
```

Next, a complex trigger is set at channels 1 and 2

```
ret% = QBwbkSetTrigComplex%(chans%(), gains%(), polarity%(), rising%(), levels!(),
hysteresis!(), NUMTRIG%, opstr$)
```

This command arms the system to acquire data. Since no pre-trigger scans were configured, no data will be available until the trigger is satisfied.

```
ret% = QBwbkArm%
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete. If the trigger is not satisfied within the programmed timeout, the driver will return control to the program. If you do not want your program to "hang" until the trigger is satisfied, it is recommended that a

background transfer be used. Once your program initiates a background transfer, control is passed back to your program to perform other tasks while waiting for a trigger or collecting data.

```
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 1, active%, retCount&)
```

Print the transferred data.

```
 PRINT "Results of BufferTransfer:"
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%
```

Close the WaveBook and exit.

```
ret% = QBwbkClose%
```

Pre- and Post-Trigger Acquisitions

The following excerpts are from the example program ADCEX6.BAS found in the WaveBook directory of your hard drive. This program shows how to setup and process acquisitions with both pre- and post-trigger scans.

Constants, declarations and arrays are defined.

```
'$INCLUDE: 'wbk.bi'

CONST CHANNELS% = 4
CONST PRESCANS& = 5
CONST POSTSCANS& = 9
CONST PREFREQ# = 100#
CONST POSTFREQ# = 200#
CONST BLOCK% = (PRESCANS& + POSTSCANS&)

DIM buf%(BLOCK% * CHANNELS%)

DIM i%, j%
DIM active%
DIM retCount&
DIM chans%(CHANNELS%)
DIM gains%(CHANNELS%), polarities%(CHANNELS%)
DIM ret%
```

The following lines set up the error handler and initialize WaveBook

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Scan definitions:

```
chans%(0) = 1        ' channel numbers
chans%(1) = 3
chans%(2) = 5
chans%(3) = 7
FOR i% = 0 TO CHANNELS% - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
NEXT i%
```

Data packing is enabled.

```
ret% = QBwbkSetDataPacking%(1)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamPrePost-specifies that both pre- and post-trigger scans are to be collected. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = QBwbkSetAcq%(WamPrePost%, PRESCANS&, POSTSCANS&)
```

To create a channel scan of non sequential channels with independent gain and unipolar/bipolar settings, the arrays of channel parameters must be created and passed to wbkSetScan.

```
ret% = QBwbkSetScan%(chans%(), gains%(), polarities%(), CHANNELS%)
```

This command sets the pre- and post-trigger sample frequencies.

```
ret% = QBwbkSetFreq%(PREFREQ#, POSTFREQ#)
```

The following line sets up a simple analog trigger using channel 1 as the trigger source.

```
ret% = QBwbkSetTrigAnalog%(1, WgcX1%, 1, WctRisingEdge%, 2!, .1)
```

This command arms the system to acquire data. Since pre-trigger scans are to be collected, scans will be immediately available for transfer into the PCs memory.

```
ret% = QBwbkArm%
```

When pre-trigger scans are included in the acquisition, scans begin to be acquired the moment the system is armed. Scans will continue to be acquired until the trigger is satisfied and the post-trigger is complete. Your application program must transfer the acquired data into a buffer in the PC as it is collected. Until the trigger occurs, your application must be prepared to accept data continuously, potentially far in excess of the sum of the specified pre-trigger and post-trigger scan counts. This is best accomplished by setting up a background transfer in cycle mode which will automatically transfer the scans as they are collected and wrap the buffer as it becomes full. The following line sets up a background transfer of the acquired scans into buf. Cycle mode is turned on, allowing the buffer to wrap around as it becomes full.

```
ret% = QBwbkBufferTransfer%(buf%(), BLOCK%, 1, 0, 0, active%, retCount&)
```

The following lines monitor the background operation, waiting for the acquisition to be complete.

```
PRINT "Waiting for trigger."
WHILE retCount& = 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
```

Once the trigger has been detected, the transfer begins.

```
PRINT "Triggered.  Transfer in progress."
WHILE active% <> 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Acquisition complete:"; retCount&; "scans acquired."
```

The following line unpacks the data so that each sample occupies an integer. The same buffer is used.

```
ret% = QBwbkBufferUnpack%(buf%(), buf%(), BLOCK%, CHANNELS%, retCount&)
```

Since the buffer has potentially wrapped around, the earliest data is not at the beginning of the buffer. The following line reorganizes the buffer so that the 1st pre-trigger scan occupies the 1st buffer location and the last post-trigger scan occupies the last buffer location.

```
ret% = QBwbkBufferRotate%(buf%(), BLOCK%, CHANNELS%, retCount&)
```

Print pre-trigger acquired data.

```
PRINT "Pre-trigger data acquired:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data:";
   FOR j% = 0 TO PRESCANS& - 1
      PRINT TAB(j% * 7 + 17); buf%(j% * CHANNELS% + i%);
   NEXT j%
   PRINT
NEXT i%
```

Print post-trigger acquired data.

```
PRINT "Post-trigger data acquired:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data:";
```

```
      FOR j% = PRESCANS& TO BLOCK% - 1
         PRINT TAB((j% - PRESCANS&) * 7 + 17); buf%(j% * CHANNELS% + i%);
      NEXT j%
      PRINT
   NEXT i%
```

Close the WaveBook and exit.

```
ret% = QBwbkClose%
```

Buffer Management

The following excerpts are from the example program ADCEX7.BAS found in the WaveBook directory of your hard drive. This example demonstrates double buffering in the background mode, so that data can be read into one buffer while another buffer can be processed in the foreground.

The program begins with an INCLUDE directive which defines useful constants and declarations that are used throughout the program.

```
'$INCLUDE: 'wbk.bi'
```

Define constants.

```
CONST CHANNELS% = 8
CONST SCANS& = 20000
CONST BLOCK% = 1000
CONST FREQ# = 5000#
```

Arrays used by the program are set up.

```
DIM buf0%(CHANNELS% * BLOCK%)
DIM buf1%(CHANNELS% * BLOCK%)
DIM i%, j%
DIM active%
DIM retCount&
DIM tmpActive%
DIM tmpRetCount&
DIM ret%
DIM totals&(CHANNELS%)
DIM whichBuf%
```

Set up the error handler.

```
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
```

The following command initializes the WaveBook and puts it online. LPT1 specifies the port number which the WaveBook is connected to and 7 is the interrupt level used.

```
ret% = QBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = QBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = QBwbkArm%
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = QBwbkSoftTrig%
```

Start the data transfer into the first buffer

```
ret% = QBwbkBufferTransfer%(buf0%(), BLOCK%, 0, 0, 0, tmpActive%, tmpRetCount&)
whichBuf% = 0

DO
   ' Swap the buffer selector, whichBuf selects the transfer 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% = QBwbkGetBackStat(active%, retCount&)
   LOOP WHILE ((active% <> 0) AND (retCount& < BLOCK%))
```

If the previous acquisition is still active, another transfer into the next buffer begins.

```
   IF (active% <> 0) THEN
      IF whichBuf% = 0 THEN
         ret% = QBwbkBufferTransfer(buf0%(), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      ELSE
         ret% = QBwbkBufferTransfer(buf1%(), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      END IF
   END IF
```

The following commands average the data in the process buffer.

```
   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
            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%
```

The results are printed.

```
      PRINT "Averages:";
      FOR j% = 0 TO CHANNELS% - 1
         PRINT TAB(j% * 7 + 17);
         PRINT USING "##.###"; (5# / 32768#) * totals&(j%) / retCount&;
      NEXT j%
      PRINT
   END IF
LOOP WHILE (active% <> 0)
```

The WaveBook is taken offline and reset.

```
ret% = QBwbkClose%
```

Sample Programs

ACDEX1.BAS

```
' This example demonstrates the use of the WaveBook's one-step
' acquisition functions and user error handling.
' Function used:
'    QBwbkRd(chan, sample, gain, polarity)
'    QBwbkRdN(chan, buf, count, trigger, level, freq, gain, polarity)
'    QBwbkRdScan(startChan, endChan, buf, gain, polarity)
'    QBwbkRdScanN(startChan, endChan, buf, count, trigger, level, freq, gain,
polarity)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST FREQ# = 1000#
CONST GAIN% = WgcX1%
CONST BIPOLAR% = 1
CONST SCANS& = 9
CONST CHANNELS% = 8
DIM buf%(SCANS& * CHANNELS%)
DIM i%, j%
DIM sample%
DIM ret%
CLS
PRINT "ADCEX1.BAS"
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Get a single sample from a single channel
ret% = QBwbkRd%(1, sample%, GAIN%, BIPOLAR%)
' Print result
PRINT "Result of Rd: "; sample%
' Get multiple samples from a single channel, triggered by a software trigger
ret% = QBwbkRdN%(1, buf%(), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%, BIPOLAR%)
' Print results
PRINT "Results of RdN:"
PRINT "Channel 1 Data: ";
FOR i% = 0 TO SCANS& - 1
   PRINT TAB(i% * 7 + 17); buf%(i%);
NEXT i%
PRINT
' Get a single sample from multiple channels
ret% = QBwbkRdScan%(1, CHANNELS%, buf%(), GAIN%, BIPOLAR%)
' Print results
PRINT "Results of RdScan:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data: "; buf%(i%)
NEXT i%
' Get multiple samples from multiple channels, triggered by a software trigger
ret% = QBwbkRdScanN%(1, CHANNELS%, buf%(), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%,
BIPOLAR%)
' Print results
PRINT "Results of RdScanN:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data: ";
   FOR j% = 0 TO SCANS& - 1
      PRINT TAB(j% * 7 + 17); buf%(j% * CHANNELS% + i%);
   NEXT j%
   PRINT
NEXT i%
'Close and exit
ret% = QBwbkClose%
END
ErrorHandler:
PRINT "ERROR in ADCEX1.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX2.BAS

```
' This example demonstrates the use of WaveBook's custom acquistion
' functions.
' Function used:
'    QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'    QBwbkSetMux(startChan, endChan, gain, polarity)
'    QBwbkSetFreq(preTrigFreq, postTrigFreq)
'    QBwbkGetFreq(preTrigFreq, postTrigFreq)
'    QBwbkGetPeriod(preTrigPeriod, postTrigPeriod)
'    QBwbkSetTrigHardware(source, level)
'    QBwbkArm()
'    QBwbkSoftTrig()
'    QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST CHANNELS% = 8
CONST SCANS& = 10
CONST BLOCK% = 6
CONST FREQ# = 5#
DIM buf%(BLOCK% * CHANNELS%)
DIM i%, j%
DIM active%
DIM retCount&
DIM preTrigFreq#, postTrigFreq#
DIM preTrigPeriod#, postTrigPeriod#
DIM ret%
CLS
PRINT "ADCEX2.BAS"
PRINT
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
' Set the scan configuration
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
' Set the post-trigger scan rates
ret% = QBwbkSetFreq%(1#, FREQ#)
' Get the pre-trigger and post-trigger scan rates in frequency and period
ret% = QBwbkGetFreq%(preTrigFreq#, postTrigFreq#)
PRINT "Result of GetFreq: pre-trigger="; preTrigFreq#; "Hz, post-trigger=";
postTrigFreq#; "Hz"
ret% = QBwbkGetPeriod%(preTrigPeriod#, postTrigPeriod#)
PRINT "Result of GetPeriod: pre-trigger="; preTrigPeriod#; "ns, post-trigger=";
postTrigPeriod#; "ns"
PRINT
' Set the trigger source to a software trigger command
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
' Arm the acquisition
ret% = QBwbkArm%
' Issue a software trigger command to the hardware
ret% = QBwbkSoftTrig%
DO
   ' Read BLOCK scans from the hardware with cycle mode off,
   ' updateSingle on and foreground enabled
   ret% = QBwbkBufferTransfer%(buf%(), BLOCK%, 0, 1, 1, active%, retCount&)
   ' Print results
   PRINT "Result of BufferTransfer: retCount="; retCount&; " active="; active%
   FOR i% = 0 TO retCount& - 1
      PRINT "Scan"; i% + 1; "Data:";
      FOR j% = 0 TO CHANNELS% - 1
         PRINT TAB(j% * 7 + 17); buf%(i% * CHANNELS% + j%);
      NEXT j%
      PRINT
   NEXT i%
   PRINT
LOOP WHILE active%  0
'Close and exit
ret% = QBwbkClose%
```

```
END
ErrorHandler:
PRINT "ERROR in ADCEX2.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX3.BAS

```
' This example takes multiple scans from hardware using a software trigger.
' Each scan includes the high speed digital I/O port (channel 0) and
' two analog channels: 5 and 8.
' Function used:
'    QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'    QBwbkSetScan(chans, gains, polarities, count)
'    QBwbkSetFreq(preTrigFreq, postTrigFreq)
'    QBwbkSetTrigHardware(source, level)
'    QBwbkArm()
'    QBwbkSoftTrig()
'    QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'    QBwbkGetDriverVersion(version)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST FREQ# = 5
CONST SCANS& = 10
CONST CHANNELS% = 3
DIM buf%(SCANS& * CHANNELS%)
DIM i%, j%
DIM active%
DIM retCount&
DIM version%
DIM chans%(CHANNELS%)
DIM gains%(CHANNELS%), polarities%(CHANNELS%)
DIM ret%
CLS
PRINT "ADCEX3.BAS"
PRINT
' Scan sequence definition
chans%(0) = 0        ' high speed digital channel
chans%(1) = 5        ' analog channel 5
chans%(2) = 8        ' analog channel 8
' Channel gains and polarities setting
FOR i% = 0 TO CHANNELS% - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
NEXT i%
' Get driver version
ret% = QBwbkGetDriverVersion%(version%)
PRINT USING "Using driver version: #.##"; .01 * version%
PRINT
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
' Set scan configuration
ret% = QBwbkSetScan%(chans%(), gains%(), polarities%(), CHANNELS%)
' Set the post-trigger scan rates
ret% = QBwbkSetFreq%(1#, FREQ#)
' Set the trigger source to a software trigger command
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
' Arm the acquisition
ret% = QBwbkArm%
' Issue a software trigger command to the hardware
ret% = QBwbkSoftTrig%
' Read SCANS& number of scans from the hardware
' with cycle mode off, updateSingle on and foreground enabled
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 1, active%, retCount&)
' Print results
```

```
PRINT "Results of BufferTransfer:"
PRINT "                Digital_ch_0  Analog_ch_5  Analog_ch_8"
FOR i% = 0 TO retCount& - 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%
'Close and exit
ret% = QBwbkClose%
END
ErrorHandler:
PRINT "ERROR in ADCEX3.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX4.BAS

```
' This example reads scans of multiple channels in the background mode
' and uses a software trigger to start the acquisition.
' Function used:
'    QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'    QBwbkSetFreq(preTrigFreq, postTrigFreq)
'    QBwbkSetMux(startChan, endChan, gain, polarity)
'    QBwbkSetTrigHardware(source, level)
'    QBwbkArm()
'    QBwbkSoftTrig()
'    QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'    QBwbkGetBackStat(active, retCount)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST CHANNELS% = 8
CONST SCANS& = 9
CONST FREQ# = 2
DIM buf%(SCANS& * CHANNELS%)
DIM i%, j%
DIM active%
DIM retCount&
DIM ret%
CLS
PRINT "ADCEX4.BAS"
PRINT
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
' Set scan's configuration
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
' Set the post-trigger scan rates
ret% = QBwbkSetFreq%(1#, FREQ#)
' Set the trigger source to a software trigger command
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)
' Arm the acquisition
ret% = QBwbkArm%
' Start reading data in the background mode with cycle mode off
' and updateSingle on
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 0, active%, retCount&)
' Issue a software trigger command to the hardware
ret% = QBwbkSoftTrig%
' Monitor the progress of the background transfer
PRINT "Waiting for trigger."
WHILE retCount& = 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
```

```
WEND
PRINT "Triggered.  Transfer in progress."
WHILE active%  0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Acquisition complete:"; retCount&; "scans acquired."
PRINT
' Print results
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%
' Close and Exit
ret% = QBwbkClose%
END
ErrorHandler:
PRINT "ERROR in ADCEX4.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX5.BAS

```
' This example takes multiple scans from hardware using a complex analog
' trigger.  The acquisition will start on a rising-edge of channel 1 at
' 2 volts OR a falling edge on channel 2 at 3 volts.
' Function used:
'    QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'    QBwbkSetMux(startChan, endChan, gain, polarity)
'    QBwbkSetFreq(preTrigFreq, postTrigFreq)
'    QBwbkSetTrigComplex(chans, gains, polarities, rising, levels, hysteresis,
count, opstr)
'    QBwbkArm()
'    QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST FREQ# = 1000
CONST SCANS& = 9
CONST CHANNELS% = 3
CONST NUMTRIG% = 2
DIM buf%(SCANS& * CHANNELS%)
DIM i%, j%
DIM active%
DIM retCount&
DIM chans%(NUMTRIG%)
DIM gains%(NUMTRIG%), polarity%(NUMTRIG%)
DIM rising%(NUMTRIG%)
DIM levels!(NUMTRIG%), hysteresis!(NUMTRIG%)
DIM opstr$
DIM ret%
' Initialize the complex trigger arrays for a rising-edge on channel 1
' at 2 volts OR a falling-edge on channel 2 at 3 volts
chans%(0) = 1
gains%(0) = WgcX1%
polarity%(0) = 1
rising%(0) = WctRisingEdge%
levels!(0) = 2
hysteresis!(0) = .1
chans%(1) = 2
gains%(1) = WgcX1%
polarity%(1) = 1
rising%(1) = WctFallingEdge%
levels!(1) = 3
hysteresis!(1) = .1
opstr$ = "+"
```

```
CLS
PRINT "ADCEX5.BAS"
PRINT

' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)
' Set the scan configuration
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)
' Set the post-trigger scan rates
ret% = QBwbkSetFreq%(1#, FREQ#)
' Set a complex trigger at channels 1 and 2
PRINT "Waiting for complex trigger of channels 1 or 2..."
PRINT
ret% = QBwbkSetTrigComplex%(chans%(), gains%(), polarity%(), rising%(), levels!(),
hysteresis!(), NUMTRIG%, opstr$)
' Arm the acquisition
ret% = QBwbkArm%
' Read SCANS& number of scans from the hardware
' with cycle mode off, updateSingle on and foreground enabled
ret% = QBwbkBufferTransfer%(buf%(), SCANS&, 0, 1, 1, active%, retCount&)
' Print results
PRINT "Results of BufferTransfer:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data:";
   FOR j% = 0 TO SCANS& - 1
      PRINT TAB(j% * 7 + 17); buf%(j% * CHANNELS% + i%);
   NEXT j%
   PRINT
NEXT i%
'Close and exit
ret% = QBwbkClose%
END
ErrorHandler:
PRINT "ERROR in ADCEX5.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX6.BAS

```
' This example demonstrates an acquisition made up of pre-trigger and
' post-trigger scans from multiple channels using a DSP-based analog
' trigger.  It also uses data packing and rotating.
' Function used:
'     QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'     QBwbkSetFreq(preTrigFreq, postTrigFreq)
'     QBwbkSetScan(chans, gains, polarities, chanCount)
'     QBwbkSetTrigAnalog(chan, gain, polarity, rising, level, opstr)
'     QBwbkArm()
'     QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'     QBwbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount, retCount)
'     QBwbkBufferRotate(buf, scanCount, chanCount, retCount)
'     QBwbkGetBackStat(active, retCount)
'     QBwbkSetErrHandler(errNum)
'     QBwbkInit(lptPort, lptIntr)
'     QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST CHANNELS% = 4
CONST PRESCANS& = 5
CONST POSTSCANS& = 9
CONST PREFREQ# = 100#
CONST POSTFREQ# = 200#
CONST BLOCK% = (PRESCANS& + POSTSCANS&)
DIM buf%(BLOCK% * CHANNELS%)
DIM i%, j%
DIM active%
DIM retCount&
DIM chans%(CHANNELS%)
```

```
DIM gains%(CHANNELS%), polarities%(CHANNELS%)
DIM ret%
CLS
PRINT "ADCEX6.BAS"
PRINT

' Scan definition
chans%(0) = 1        ' channel numbers
chans%(1) = 3
chans%(2) = 5
chans%(3) = 7
FOR i% = 0 TO CHANNELS% - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
NEXT i%
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)
' Enable data packing
ret% = QBwbkSetDataPacking%(1)
' Set the acquisition for pre/post-trigger mode and the scan counts
ret% = QBwbkSetAcq%(WamPrePost%, PRESCANS&, POSTSCANS&)
' Set the scan configuration
ret% = QBwbkSetScan%(chans%(), gains%(), polarities%(), CHANNELS%)
' Set the pre-trigger and post-trigger scan rates
ret% = QBwbkSetFreq%(PREFREQ#, POSTFREQ#)
' Set the trigger source to an analog trigger on channel 1 at 2 volts
ret% = QBwbkSetTrigAnalog%(1, WgcX1%, 1, WctRisingEdge%, 2!, .1)
' Arm the acquisition
ret% = QBwbkArm%
' Start reading data in the background mode with cycle mode on
' and updateSingle off
ret% = QBwbkBufferTransfer%(buf%(), BLOCK%, 1, 0, 0, active%, retCount&)
' Monitor the progress of the background transfer
PRINT "Waiting for trigger."
WHILE retCount& = 0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Triggered.  Transfer in progress."
WHILE active%  0
   ret% = QBwbkGetBackStat%(active%, retCount&)
WEND
PRINT "Acquisition complete:"; retCount&; "scans acquired."
PRINT
' Unpack the packed data using the same buffer
ret% = QBwbkBufferUnpack%(buf%(), buf%(), BLOCK%, CHANNELS%, retCount&)
' Rotate the unpacked data so that the earliest data starts at the
' beginning of the buffer and the latest is at the end
ret% = QBwbkBufferRotate%(buf%(), BLOCK%, CHANNELS%, retCount&)
' Print results
PRINT "Pre-trigger data acquired:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data:";
   FOR j% = 0 TO PRESCANS& - 1
      PRINT TAB(j% * 7 + 17); buf%(j% * CHANNELS% + i%);
   NEXT j%
   PRINT
NEXT i%
PRINT
PRINT "Post-trigger data acquired:"
FOR i% = 0 TO CHANNELS% - 1
   PRINT "Channel"; i% + 1; "Data:";
   FOR j% = PRESCANS& TO BLOCK% - 1
      PRINT TAB((j% - PRESCANS&) * 7 + 17); buf%(j% * CHANNELS% + i%);
   NEXT j%
   PRINT
NEXT i%
' Close and Exit
ret% = QBwbkClose%
END
ErrorHandler:
PRINT "ERROR in ADCEX6.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)
```

```
'Close and exit
ret% = QBwbkClose%
END
```

ADCEX7.BAS

```
' 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:
'    QBwbkSetAcq(mode, preTrigCount, postTrigCount)
'    QBwbkSetMux(startChan, endChan, gain, polarity)
'    QBwbkSetFreq(preTrigFreq, postTrigFreq)
'    QBwbkSetTrigHardware(source, level)
'    QBwbkArm()
'    QBwbkSoftTrig()
'    QBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
retCount)
'    QBwbkGetBackStat(active, retCount)
'    QBwbkSetErrHandler(errNum)
'    QBwbkInit(lptPort, lptIntr)
'    QBwbkClose()
'$INCLUDE: 'wbk.bi'
CONST CHANNELS% = 8
CONST SCANS& = 20000
CONST BLOCK% = 1000
CONST FREQ# = 5000#
DIM buf0%(CHANNELS% * BLOCK%)
DIM buf1%(CHANNELS% * BLOCK%)
DIM i%, j%
DIM active%
DIM retCount&
DIM tmpActive%
DIM tmpRetCount&
DIM ret%
DIM totals&(CHANNELS%)
DIM whichBuf%
CLS
PRINT "ADCEX7.BAS"
PRINT
' Set error handler and initialize WaveBook
ret% = QBwbkSetErrHandler%(100)
ON ERROR GOTO ErrorHandler
ret% = QBwbkInit%(LPT1%, IRQ7%)

' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = QBwbkSetAcq%(WamNShot%, 0, SCANS&)

' Set the scan configuration
ret% = QBwbkSetMux%(1, CHANNELS%, WgcX1%, 1)

' Set the post-trigger scan rates
ret% = QBwbkSetFreq%(1#, FREQ#)

' Set the trigger source to a software trigger command
ret% = QBwbkSetTrigHardware%(WtsSoftware%, 0!)

' Arm the acquisition
ret% = QBwbkArm%

' Issue a software trigger command to the hardware
ret% = QBwbkSoftTrig%

' Start reading data into the first buffer
ret% = QBwbkBufferTransfer%(buf0%(), BLOCK%, 0, 0, 0, tmpActive%, tmpRetCount&)
whichBuf% = 0

DO
   ' Swap the buffer selector, whichBuf selects the transfer 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% = QBwbkGetBackStat(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% = QBwbkBufferTransfer(buf0%(), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      ELSE
         ret% = QBwbkBufferTransfer(buf1%(), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      END IF
   END IF

   ' Process the data into the process buffer
   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
            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%
      PRINT "Averages:";
      FOR j% = 0 TO CHANNELS% - 1
         PRINT TAB(j% * 7 + 17);
         PRINT USING "##.###"; (5# / 32768#) * totals&(j%) / retCount&;
      NEXT j%
      PRINT
   END IF
LOOP WHILE (active%  0)

'Close and exit
ret% = QBwbkClose%

END

ErrorHandler:

PRINT "ERROR in ADCEX7.BAS"
PRINT "BASIC Error :" + STR$(ERR)
IF ERR = 100 THEN PRINT "WaveBook Error : " + HEX$(wbkErrno%)

'Close and exit
ret% = QBwbkClose%

END
```

Using WaveBook/512 with Turbo Pascal 8

This chapter describes the use of the Turbo Pascal (version 7) language with the **standard** API to develop a basic data acquisition program. For additional functions of the standard API, refer to chapter 10. **Note**: The WaveBook system includes full-featured DOS and Windows software drivers for C (chapter 6), QuickBASIC (chapter 7), Turbo Pascal, and Visual Basic (chapter 9). The enhanced API (for C, Visual Basic and Delphi) is described in chapters 11 and 12.

To use the example programs located in the WBK\DOS\TP7 directories, make sure that your program specifies WBK.TPU unit in the uses clause. Also be sure that the WBK.TPU unit file is in the Turbo Pascal search path. The temperature examples will use TBKTC.TPU in addition to WBK.TPU.

Simple Analog Input

The following excerpts are from the example program ADCEX1.PAS found in the WaveBook directory of your hard drive. This program shows several examples of the highest level WaveBook driver functions. These functions are the easiest to use at the cost of flexibility. For additional flexibility, lower level functions, discussed later, should be used.

Define constants used by the program.

```
const
   CHANS      = 8;
   SCANS      = 10;
   FREQ       = 1000.0;
   GAIN       = WgcX1;
   BIPOLAR    = 1;
```

Buffers and variables used by program are declared.

```
var
   buf:array[0..CHANS * SCANS - 1] of integer;
   sample:integer;
   i, j:word;
```

Create custom error handler routine.

```
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

The following line sets the error handler to use the custom error handling routine.

```
   wbkSetErrHandler(myhandler);
```

The WaveBook is initilized and put on-line. LPT1 and interrupt level 7 are used.

```
   wbkInit(LPT1, 7);
```

The following command will get one sample from channel 1 at a gain of X1 with its unipolar/bipolar setting set to bipolar. The value will be returned in the variable "sample".

```
   wbkRd(1, @sample, GAIN, BIPOLAR);
```

The result of the acquisition is printed.

```
   writeln('Result of Rd:', sample:6);
```

The following command will get 8 samples from a single channel. Channel 1 at a gain of X1 in bipolar mode is selected. This command requires a trigger to be satisfied before the data is collected. A trigger source of software will start the acquisition immediately. The frequency of data collection is set to 1 kHz. The data will be returned in the integer array "buf".

```
wbkRdN(1, @buf, SCANS, WtsSoftware, 0.0, FREQ, GAIN, BIPOLAR);
```

The result of the acquisition is printed.

```
 (* Print results *)
write('Results of RdN: ');
for i := 0 to CHANS - 1 do write(' ', buf[i]:6);
```

The following command collects an entire scan of data comprised of multiple channels. The following function arguments define a scan starting with channel 1 and ending with channel 8. This command sets all the channels to the same gain and unipolar/bipolar setting. The data is returned in the array buf.

```
wbkRdScan(1, CHANS, @buf, GAIN, BIPOLAR);
```

Print the results.

```
 writeln('Results of RdScan:');
for i := 0 to CHANS - 1  do
   writeln('Channel ', (i + 1):2, ' Data: ', buf[i]:6);
```

The following command collects multiple scans consisting of multiple channels. Like wbkRdScan, this command sets all the channels to the same gain and unipolar/bipolar setting. Since multiple scans are being collected, a trigger source and timebase is required. The arguments shown set the trigger source to Software causing an immediate trigger. The sample frequency of 1Khz.

```
wbkRdScanN(1, CHANS, @buf, SCANS, WtsSoftware, 0.0, FREQ, GAIN, BIPOLAR);
```

Print the results.

```
writeln;
writeln('Results of RdScanN:');
for i := 0 to CHANS - 1 do begin
   write('Channel ', (i + 1):2, ' Data: ');
   for j := 0 to SCANS - 1 do write(' ', buf[(j * CHANS) + i]:6);
   writeln;
end;
```

The WaveBook is taken off-line and reset.

```
wbkClose;
```

Low-Level Analog Input

The following excerpts are from the example program ADCEX2.PAS found in the WaveBook directory of your hard drive. This program shows several examples of the lowest level WaveBook driver functions. These functions are more complex than the high-level functions but allow the greatest flexibility.

Constants are defined. CHANS is the number of channels in the scan, SCANS is the number of scans in the acquisition, BLOCK is the number of scans read at one time, and FREQ is the post-trigger scan rate in Hz.

Define constants used by the program.

```
const
   CHANS       = 8;
   SCANS       = 10;
   BLOCK       = 6;
   FREQ        = 5.0;
```

Buffers and variables used by program are declared.

```
var
   buf:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   active:byte;
   retCount:longint;
   pre_trig_freq, post_trig_freq:double;
   pre_trig_period, post_trig_period:double;
```

Create custom error handler routine.

```
begin
```

```
      writeln('Error! Program aborted');
      writeln('WaveBook Error: ', error_code);
      halt;
   end;
```

The following line sets the error handler to use the custom error handling routine.

```
wbkSetErrHandler(myhandler);
```

Initialize WaveBook and put on-line. LPT1 and IRQ7 are used.

```
wbkInit(LPT1, 7);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following line shows the usage of the wbkGetFreq command which returns the present settings for the pre- and post-trigger frequencies.

```
 wbkGetFreq(@pre_trig_freq, @post_trig_freq);
wbkGetFreq: pre-trigger=', pre_trig_freq:0:2,
writeln('Result wbkGetFreq: pre-trigger=', pre_trig_freq:0:2,
   'Hz, post-trigger=', post_trig_freq:0:2, 'Hz');
wbkGetPeriod(@pre_trig_period, @post_trig_period);
writeln('Result wbkGetPeriod: pre-trigger=', pre_trig_period:0:0,
   'ns, post-trigger=', post_trig_period:0:0, 'ns');
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm;
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig;
```

The following lines perform a foreground transfer from the WaveBook of size BLOCK. The data is then printed on the screen. The transfers will continue until the acquisition is no longer active.

```
 repeat
   (* Read BLOCK scans from the hardware with cycle mode off, *)
   (* updateSingle on and foreground enabled                  *)
   wbkBufferTransfer(@buf, BLOCK, 0, 1, 1, @active, @retCount);
```

```
        (* Print results *)
        writeln;
        writeln('Result wbkBufferTransfer: retCount=', retCount, ', active=',
    active);
        for i := 0 to retCount - 1 do begin
           write('Scan ', (i + 1):5, ':');
           for j := 0 to CHANS - 1 do write(' ', buf[i * CHANS + j]:6);
           writeln;
        end;
     until (active = 0);
```

The WaveBook is taken off-line and reset.

```
     wbkClose;
```

Accessing the High-Speed Digital Input Port

The following excerpts are from the example program ADCEX3.PAS found in the WaveBook directory of your hard drive. This program shows how to collect analog and high-speed digital signals concurrently, in the same scan.

Constants are defined.

```
const
   FREQ   = 5;
   SCANS  = 10;
   CHANS  = 3;
```

Buffers and variables used by program are declared.

```
var
   buf:array[0..CHANS * SCANS - 1] of integer;
   i, j:word;
   version:word;
   chs:array[0..CHANS - 1] of word;
   gains:array[0..CHANS - 1] of byte;
   polarities:array[0..CHANS - 1] of byte;
   active:byte;
   retCount:longint;
```

Create custom error handler routine.

```
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

To create a channel scan of non sequential channels with independent gain and unipolar/bipolar settings, the arrays of channel parameters must be created and passed to wbkSetScan. Channel 0 is the high speed digital port. When added to the channel scan, the high speed digital port is scanned synchronously with the analog signals.

```
   chs[0]:=0;       (* high speed digital channel *)
   chs[1]:=5;       (* analog channel 5           *)
   chs[2]:=8;       (* analog channel 8           *)
```

The following lines set all of the gains and unipolar/bipolar settings to the same setting. Your program can assign each channel to a different value.

```
   for i:=0 to 2 do begin
     gains[i]:=WgcX1;  (* unity gain *)
     polarities[i]:=1; (* bipolar    *)
   end;
```

The following line gets the driver version and prints the information.

```
wbkGetDriverVersion(@version);
writeln('Using driver version ', 0.01 * version);
```

The next line sets the error handler to use the custom error handling routine.

```
wbkSetErrHandler(myhandler);
```

Initialize the WaveBook and put it on line. LPT port 1 and interrupt level 7 are used.

```
wbkInit(LPT1, 7);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

Setup the scan by passing the channel configuration arrays to the wbkSetScan command.

```
wbkSetScan(@chs, @gains, @polarities, CHANS);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm;
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig;
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete. The results are printed.

```
   wbkBufferTransfer(@buf, SCANS, 0, 1, 1, @active, @retCount);

(* Print results *)
writeln('Results of BufferTransfer:');
writeln('Scan   Digital_ch_0  Analog_ch_5  Analog_ch_8');
for i := 0 to retCount - 1 do begin
   (* get the upper (valid) 8 bits of the digital input *)
   buf[CHANS * i] := buf[CHANS * i] shr 8;
   write(' ', (i + 1):2, '   ');
   for j := 0 to 2 do write('  ', buf[CHANS * i + j]:6, '      ');
   writeln;
end;
```

The WaveBook is taken off-line and reset.

```
wbkClose;
```

Background Processing of Analog Input

The following excerpts are from the example program ADCEX4.PAS found in the WaveBook directory of your hard drive. This program shows how to collect analog samples and transfer them into the PC's memory in the background. Once the background acquisition is configured and armed, your program can perform other operations concurrently with the background data collection. The foreground program can use the wbkGetBackStat command to periodically check on the status of the acquisition.

For performing acquisition that are greater in size than the allocated buffer, the background operation can be set to Cycle mode which will wrap around in the allocated buffer as it becomes full. In this mode, your program must monitor the background and transfer the data out of the allocated buffer before the background operation overwrites it.

Define program constants.

```
      CHANS  = 8;
      SCANS  = 9;
      FREQ   = 2.0;
```

Buffers and variables used by program are declared.

```
   var
      buf:array[0..CHANS * SCANS - 1] of integer;
      i, j:word;
      active:byte;
      retCount: longint;
```

Create custom error handler routine.

```
   procedure myhandler( error_code:integer );
   begin
      writeln('Error! Program aborted');
      writeln('WaveBook Error: ', error_code);
      halt;
   end;
```

The next line sets the error handler to use the custom error handling routine.

```
      wbkSetErrHandler(myhandler);
```

Initialize the WaveBook and put it on line. LPT port 1 and interrupt level 7 are used.

```
      wbkInit(LPT1, 7);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
      wbkSetAcq(WamNShot, 0, SCANS);
```

Set the scan configuration.

```
      wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the post-trigger scan rates. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
      wbkSetFreq(1.0, FREQ);
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
      wbkSetTrigHardware(WtsSoftware, 0.0);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
      wbkArm;
```

The following line sets up a background transfer. Regardless of the state of the acquisition, the program will immediately return from this function call and proceed to the next line. As the data is collected by the WaveBook, it is automatically transferred to the buffer À ÀbufÀ À

```
wbkBufferTransfer(@buf, SCANS, 0, 1, 0, @active, @retCount);
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig;
```

Although your program can begin processing other tasks at this point, our example program simply monitors the background until the user hits a key or the acquisition is complete.

```
  repeat
   wbkGetBackStat(@active, @retCount);
   write('Transfer in progress: ', retCount, ' scans acquired.', chr(13));
until active = 0;
```

Once the acquisition has completed, a message is printed.

```
writeln('Acquisition complete.');
```

The following lines print the collected data.

```
writeln('Data acquired:');
for i := 0 to CHANS - 1 do begin
   write('Channel ', (i + 1):2, ' Data:');
   for j := 0 to retCount - 1 do write(' ', buf[(j * CHANS) + i]:6);
   writeln;
end;
```

The WaveBook is taken off-line and reset.

```
wbkClose;
```

Complex Triggering

The following excerpts are from the example program ADCEX5.PAS found in the WaveBook directory of your hard drive. This program shows how to setup a complex trigger where more than one channel can be combined in a logical trigger equation. The acquisition will start on a rising edge of channel 1 at 2 volts OR a falling edge on channel 2 at 3 volts.

Define program constants.

```
const
   FREQ    = 1000;
   SCANS   = 9;
   CHANS   = 3;
   NUM_TRIG= 2;
```

Buffers and variables used by program are declared.

```
var

   buf:array[0..CHANS * SCANS - 1] of integer;
   i, j: integer;
   retCount:longint;
   active:byte;
   chans_tr:array[0..NUM_TRIG - 1] of word;
   gains_tr:array[0..NUM_TRIG - 1] of byte;
   polarity_tr:array[0..NUM_TRIG - 1] of byte;
   rising:array[0..NUM_TRIG - 1] of byte;
   levels:array[0..NUM_TRIG - 1] of single;
   hysteresis:array[0..NUM_TRIG - 1] of single;
   opstr:string;
```

Create custom error handler routine.

```
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

The following lines are definitions and initialization for the variables used for setting up the trigger equation. The variable chans_tr is an array of channels used in the trigger equation. Channels 1 and 2 are specified. The variable gain_tr is an array that holds the gains for channels 1 and 2. In this case they are both set to X1. Channels 1 and 2 can also be a part of the scan group with the same or different gain assignments. The variable polarity_tr is an array that holds the unipolar/bipolar settings for channels 1 and 2. The variable rising is an array that holds the edge settings for channels 1 and 2. In this case, channel 1 triggers on the rising edge, while channel 2 triggers on the falling edge. The variables levels and hysteresis are arrays that hold the voltage thresholds and hysteresis settings for channels 1 and 2, respectively. The variable opstr holds the boolean operator for the trigger equation. The + sign indicates an OR operator between channels 1 and 2.

```
chans_tr[0] := 1;
gains_tr[0] := WgcX1;
polarity_tr[0] := 1;
rising[0] := WcrRisingEdge;
levels[0] := 2.0;
hysteresis[0] := 0.1;

chans_tr[1] := 2;
gains_tr[1] := WgcX1;
polarity_tr[1] := 1;
rising[1] := WcrFallingEdge;
levels[1] := 3.0;
hysteresis[1] := 0.1;

opstr := '+';
```

The previous definitions create the following trigger setup:

System Trigger = (CH1 @ X1, bipolar, rising edge through 2.0V with 0.1V hyst) OR (CH2 @ X1, bipolar, falling edge through 3.0V with 0.1V hyst)

The next line sets the error handler to use the custom error handling routine.

```
wbkSetErrHandler(myhandler);
```

Initialize the WaveBook and put it on line. LPT port 1 and interrupt level 7 are used.

```
wbkInit(LPT1, 7);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

Set the scan configuration.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following lines notify the user of the system's status then setup the complex trigger.

```
writeln('Waiting for complex trigger at channels 1 and 2...');
wbkSetTrigComplex(@chans_tr, @gains_tr, @polarity_tr, @rising, @levels,
        @hysteresis, NUM_TRIG, opstr);
```

This command arms the system to acquire data. Since no pre-trigger scans were configured, no data will be available until the trigger is satisfied.

```
wbkArm;
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete. If the trigger is not satisfied within the programmed timeout, the driver will return control to the program. If you do not want your program to "hang" until the trigger is satisfied, it is recommended that a background transfer be used. Once your program initiates a background transfer, control is passed back to your program to perform other tasks while waiting for a trigger or collecting data.

```
wbkBufferTransfer(@buf, SCANS, 0, 1, 1, @active, @retCount);
```

Print the transferred data.

```
 writeln('Results of BufferTransfer:');
for i := 0 to CHANS - 1 do begin
   write('Channel ', (i + 1):2, ' Data: ');
   for j := 0 to retCount - 1 do write(' ', buf[(j * CHANS) + i]:6);
   writeln;
end;
```

The WaveBook is taken off-line and reset.

```
wbkClose;
```

Pre- and Post-Trigger Acquisitions

The following excerpts are from the example program ADCEX6.PAS found in the WaveBook directory of your hard drive. This program shows how to setup and process acquisitions with both pre- and post-trigger scans.

Define program constants.

```
const
   CHANS       = 4;
   PRE_SCANS   = 5;
   POST_SCANS  = 9;
   PRE_FREQ    = 100.0;
   POST_FREQ   = 200.0;
   BLOCK       = PRE_SCANS+POST_SCANS;
```

Buffers and variables used by program are declared.

```
var

   buf:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   chs:array[0..CHANS-1] of word;
   gains:array[0..CHANS-1] of byte;
   polarities:array[0..CHANS-1] of byte;
   active:byte;
   retCount:longint;
   pre_trig_freq, post_trig_freq:double;
   pre_trig_period, post_trig_period:double;
```

Create custom error handler routine.

```
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

Define the sequence of the channels for the acquisition.

```
    chs[0] := 1;
```

```
chs[1] := 3;
chs[2] := 5;
chs[3] := 7;

for i := 0 to CHANS-1 do begin
  gains[i] := WgcX1;   (* unity gain *)
  polarities[i] := 1;  (* bipolar    *)
end;
```

The next line sets the error handler to use the custom error handling routine.

```
wbkSetErrHandler(myhandler);
```

Initialize the WaveBook and put it on line. LPT port 1 and interrupt level 7 are used.

```
wbkInit(LPT1, 7);
```

Enable data packing

```
wbkSetDataPacking(1);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamPrePost-specifies that both pre- and post-trigger scans are to be collected. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamPrePost, PRE_SCANS, POST_SCANS);
```

Set the scan configuration

```
wbkSetScan(@chs, @gains, @polarities, CHANS);
```

This command sets the pre- and post-trigger sample frequencies.

```
wbkSetFreq(PRE_FREQ, POST_FREQ);
```

Set the trigger source to an analog trigger on channel 1 at 2 volts

```
wbkSetTrigAnalog(1, WgcX1, 1, WctRisingEdge, 2.0, 0.1);
```

This command arms the system to acquire data. Since no pre-trigger scans were configured, no data will be available until the trigger is satisfied.

```
wbkArm;
```

When pre-trigger scans are included in the acquisition, scans begin to be acquired the moment the system is armed. Scans will continue to be acquired until the trigger is satisfied and the post-trigger is complete. Your application program must transfer the acquired data into a buffer in the PC as it is collected. Until the trigger occurs, your application must be prepared to accept data continuously, potentially far in excess of the sum of the specified pre-trigger and post-trigger scan counts. This is best accomplished by setting up a background transfer in cycle mode which will automatically transfer the scans as they are collected and wrap the buffer as it becomes full. The following line sets up a background transfer of the acquired scans into buf. Cycle mode is turned on, allowing the buffer to wrap around as it becomes full.

```
wbkBufferTransfer(@buf, BLOCK, 1, 1, 0, @active, @retCount);
```

The following lines monitor the background operation, waiting for the acquisition to be complete.

```
while (active <> 0) do begin
   wbkGetBackStat(@active, @retCount);
   write('Transfer in progress: ', retCount, ' scans acquired.', chr(13));
end;
writeln;
writeln('Acquisition complete.');
```

The following line unpacks the data so that each sample occupies an integer.

```
wbkBufferUnpack(@buf, @buf, BLOCK, CHANS, retCount);
```

Since the buffer has potentially wrapped around, the earliest data is not at the beginning of the buffer. The following line reorganizes the buffer so that the 1st pre-trigger scan occupies the 1st buffer location and the last post-trigger scan occupies the last buffer location.

```
wbkBufferRotate(@buf, BLOCK, CHANS, retCount);
```

The following lines print the acquired data.

```
   writeln('Pre_trigger data acquired:');
for i := 0 to CHANS - 1 do begin
   write('Channel ', chs[i]:2, ' Data:');
   for j := 0 to PRE_SCANS - 1 do write(' ', buf[(j * CHANS) + i]:6);
   writeln;
end;
writeln('Post-trigger data acquired:');
for i := 0 to CHANS - 1 do begin
   write('Channel ', chs[i]:2, ' Data:');
   for j := PRE_SCANS to BLOCK - 1 do write(' ', buf[(j * CHANS) + i]:6);
   writeln;
end;
```

The WaveBook is taken off-line and reset.

```
wbkClose;
```

Buffer Management

The following excerpts are from the example program ADCEX7.PAS found in the WaveBook directory of your hard drive. 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.

Define program constants.

```
const
   CHANS       = 8;
   SCANS       = 20000;
   BLOCK       = 1000;
   FREQ        = 5000.0;
```

Buffers and variables used by program are declared.

```
var

   buf0:array[0..CHANS * BLOCK - 1] of integer;
   buf1:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   active:byte;
   retCount:longint;
   tmpActive:byte;
   tmpRetCount:longint;
   totals:array[0..CHANS - 1] of longint;
   whichBuf:integer;
```

Create custom error handler routine.

```
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

The next line sets the error handler to use the custom error handling routine.

```
wbkSetErrHandler(myhandler);
```

Initialize the WaveBook and put it on line. LPT port 1 and interrupt level 7 are used.

```
wbkInit(LPT1, 7);
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
wbkSetAcq(WamNShot, 0, SCANS);
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
wbkSetMux(1, CHANS, WgcX1, 1);
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
wbkSetFreq(1.0, FREQ);
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
wbkSetTrigHardware(WtsSoftware, 0.0);
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
wbkArm;
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
wbkSoftTrig;
```

Start the data transfer into the first buffer

```
wbkBufferTransfer(@buf0, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount);
whichBuf := 0;
```

The contents of the two buffers are swapped. The loop continues until the acquisition completes or the buffer fills.

```
repeat
   if whichBuf = 1 then whichBuf := 0 else whichBuf := 1;
```

Wait for the acquisition to go inactive or the buffer to be filled.

```
   repeat
      wbkGetBackStat(@active, @retCount);
   until (active = 0)  or (retCount = BLOCK);
```

If the previous acquisition is still active, another transfer into the next buffer begins.

```
if (active <> 0) then begin
         if (whichBuf = 0) then
            wbkBufferTransfer(@buf0, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount)
         else
            wbkBufferTransfer(@buf1, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount);
      end;
```

The following commands average the data in the process buffer and print the results.

```
   if (retCount > 0) then begin
      (* Average the readings in the process buffer and print the results *)
      for j := 0 to CHANS - 1 do totals[j] := 0;
```

```
         for i := 0 to retCount - 1 do begin
            for j := 0 to CHANS - 1 do begin
               if (whichBuf = 1) then
                  totals[j] := totals[j] + buf0[i * CHANS + j]
               else
                  totals[j] := totals[j] + buf1[i * CHANS + j];
            end;
         end;
         write('Averages:');
         for j := 0 to CHANS - 1 do begin
            write(' ',((5.0 * totals[j]) / (32768.0 * retCount)) :6:3);
         end;
         writeln;
      end;
   until (active = 0);
```

The WaveBook is taken off-line and reset.

```
   wbkClose;
```

Sample Programs

ADCEX1.PAS

```
(* This example demonstrates the use of the WaveBook's one-step            *)
(* acquisition functions and user error handling.                          *)
(* Function used:                                                          *)
(*    wbkRd(chan, &sample, gain, polarity);                                *)
(*    wbkRdN(chan, buf, count, trigger, level, freq, gain, polarity);  *)
(*    wbkRdScan(startChan, endChan, buf, gain, polarity);                  *)
(*    wbkRdScanN(startChan, endChan, buf, count, trigger, level, freq, *)
(*               gain, polarity);                                          *)
(*    wbkSetErrHandler(wbkErrorHandler);                                   *)
(*    wbkInit(lptPort, lptIntr);                                           *)
(*    wbkClose();                                                          *)
program adcex1;
uses wbk;
const
   CHANS       = 8;
   SCANS       = 10;
   FREQ        = 1000.0;
   GAIN        = WgcX1;
   BIPOLAR     = 1;
var
   buf:array[0..CHANS * SCANS - 1] of integer;
   sample:integer;
   i, j:word;
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
writeln;
   writeln('ADCEX1.PAS');
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Get a single sample from a single channel *)
   wbkRd(1, @sample, GAIN, BIPOLAR);
   (* Print result *)
   writeln('Result of Rd: ', sample:6);
   (*Get multiple samples from a single channel, triggered by a software trigger*)
   wbkRdN(1, @buf, SCANS, WtsSoftware, 0.0, FREQ, GAIN, BIPOLAR);
   (* Print results *)
   write('Results of RdN: ');
   for i := 0 to CHANS - 1 do write(' ', buf[i]:6);
   (* Get a single sample from multiple channels *)
   wbkRdScan(1, CHANS, @buf, GAIN, BIPOLAR);

   (* Print results *)
   writeln;
   writeln;
```

```
   writeln('Results of RdScan:');
   for i := 0 to CHANS - 1  do
      writeln('Channel ', (i + 1):2, ' Data: ', buf[i]:6);

   (*Get multiple samples from multiple channels, triggered by a software trigger*)
   wbkRdScanN(1, CHANS, @buf, SCANS, WtsSoftware, 0.0, FREQ, GAIN, BIPOLAR);

   (* Print results *)
   writeln;
   writeln('Results of RdScanN:');
   for i := 0 to CHANS - 1 do begin
      write('Channel ', (i + 1):2, ' Data: ');
      for j := 0 to SCANS - 1 do write(' ', buf[(j * CHANS) + i]:6);
      writeln;
   end;
   (* Close and exit *)
   wbkClose;
end.
```

ADCEX2.PAS

```
(* This example demonstrates the use of WaveBook's custom acquistion      *)
(* functions.                                                              *)
(* Functions used:                                                         *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                        *)
(*    wbkSetMux(startChan, endChan, gain, polarity);                       *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                               *)
(*    wbkGetFreq(preTrigFreq, postTrigFreq);                               *)
(*    wbkGetPeriod(preTrigPeriod, postTrigPeriod);                         *)
(*    wbkSetTrigHardware(source, level);                                   *)
(*    wbkArm();                                                            *)
(*    wbkSoftTrig();                                                       *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,  *)
(*                      active, retCount);                                 *)
(*    wbkInit(lptPort, lptIntr);                                           *)
(*    wbkClose();                                                          *)
program adcex2;
uses wbk;
const
   CHANS       = 8;
   SCANS       = 10;
   BLOCK       = 6;
   FREQ        = 5.0;
var
   buf:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   active:byte;
   retCount:longint;
   pre_trig_freq, post_trig_freq:double;
   pre_trig_period, post_trig_period:double;

procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
   writeln;
   writeln('ADCEX2.PAS');
   writeln;
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Set the acquisition to NShot on trigger and the post-trigger scan count *)
   wbkSetAcq(WamNShot, 0, SCANS);
   (* Set the scan configuration *)
   wbkSetMux(1, CHANS, WgcX1, 1);
   (* Set the post-trigger scan rates *)
   wbkSetFreq(1.0, FREQ);
   (* Get the pre-trigger and post-trigger scan rates in frequency and period *)
   wbkGetFreq(@pre_trig_freq, @post_trig_freq);
   writeln('Result wbkGetFreq: pre-trigger=', pre_trig_freq:0:2,
      'Hz, post-trigger=', post_trig_freq:0:2, 'Hz');
```

```
   wbkGetPeriod(@pre_trig_period, @post_trig_period);
   writeln('Result wbkGetPeriod: pre-trigger=', pre_trig_period:0:0,
      'ns, post-trigger=', post_trig_period:0:0, 'ns');
   (* Set the trigger source to a software trigger command *)
   wbkSetTrigHardware(WtsSoftware, 0.0);
   (* Arm the acquisition *)
   wbkArm;
   (* Issue a software trigger command to the hardware *)
   wbkSoftTrig;
   repeat
      (* Read BLOCK scans from the hardware with cycle mode off, *)
      (* updateSingle on and foreground enabled                  *)
      wbkBufferTransfer(@buf, BLOCK, 0, 1, 1, @active, @retCount);
      (* Print results *)
      writeln;
      writeln('Result wbkBufferTransfer: retCount=', retCount, ', active=',
active);
      for i := 0 to retCount - 1 do begin
         write('Scan ', (i + 1):5, ':');
         for j := 0 to CHANS - 1 do write(' ', buf[i * CHANS + j]:6);
         writeln;
      end;
   until (active = 0);
   (* Close and exit *)
   wbkClose;
end.
```

ADCEX3.PAS

```
(* This example takes multiple scans from hardware using a software trigger.  *)
(* Each scan includes the high speed digital I/O port (channel 0) and        *)
(* two analog channels: 5 and 8.                                             *)
(* Functions used:                                                           *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                          *)
(*    wbkSetScan(chans, gains, polarities, count);                           *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                                 *)
(*    wbkSetTrigHardware(source, level);                                     *)
(*    wbkArm();                                                              *)
(*    wbkSoftTrig();                                                         *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,     *)
(*                      active, retCount);                                   *)
(*    wbkGetDriverVersion(version);                                          *)
(*    wbkInit(lptPort, lptIntr);                                             *)
(*    wbkClose();                                                            *)
program adcex3;
uses wbk;
const
   FREQ   = 5;
   SCANS  = 10;
   CHANS  = 3;
var
   buf:array[0..CHANS * SCANS - 1] of integer;
   i, j:word;
   version:word;
   chs:array[0..CHANS - 1] of word;
   gains:array[0..CHANS - 1] of byte;
   polarities:array[0..CHANS - 1] of byte;
   active:byte;
   retCount:longint;
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
   writeln;
   writeln('ADCEX3.PAS');
   writeln;
   (* Scan sequence definition *)
   chs[0] := 0;       (* high speed digital channel *)
   chs[1] := 5;       (* analog channel 5          *)
   chs[2] := 8;       (* analog channel 8          *)
```

```
   (* Channel gains and polarities setting *)
   for i := 0 to 2 do begin
     gains[i] := WgcX1;  (* unity gain *)
     polarities[i] := 1; (* bipolar    *)
   end;
   (* Get driver version *)
   wbkGetDriverVersion(@version);
   writeln('Using driver version ', (0.01 * version):0:2);
   writeln;
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Set the acquisition to NShot on trigger and the post-trigger scan count *)
   wbkSetAcq(WamNShot, 0, SCANS);
   (* Set the scan configuration *)
   wbkSetScan(@chs, @gains, @polarities, CHANS);
   (* Set the post-trigger scan rates *)
   wbkSetFreq(1.0, FREQ);
   (* Set the trigger source to a software trigger command *)
   wbkSetTrigHardware(WtsSoftware, 0.0);
   (* Arm the acquisition *)
   wbkArm;
   (* Issue a software trigger command to the hardware *)
   wbkSoftTrig;
   (* Read SCANS scans from the hardware with cycle mode off, *)
   (* updateSingle on and foreground enabled                  *)
   wbkBufferTransfer(@buf, SCANS, 0, 1, 1, @active, @retCount);
   (* Print results *)
   writeln('Results of BufferTransfer:');
   writeln('Scan   Digital_ch_0  Analog_ch_5  Analog_ch_8');
   for i := 0 to retCount - 1 do begin
      (* get the upper (valid) 8 bits of the digital input *)
      buf[CHANS * i] := buf[CHANS * i] shr 8;
      write(' ', (i + 1):2, '   ');
      for j := 0 to 2 do write('  ', buf[CHANS * i + j]:6, '      ');
      writeln;
   end;
   (* Close and exit *)
   wbkClose;
end.
```

ADCEX4.PAS

```
(* This example reads scans of multiple channels in the background mode   *)
(* and uses a software trigger to start the acquisition.                   *)
(* Functions used:                                                         *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                        *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                               *)
(*    wbkSetMux(startChan, endChan, gain, polarity);                       *)
(*    wbkSetTrigHardware(source, level);                                   *)
(*    wbkArm();                                                            *)
(*    wbkSoftTrig();                                                       *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,  *)
(*                        active, retCount);                               *)
(*    wbkGetBackStat(active, retCount);                                    *)
(*    wbkInit(lptPort, lptIntr);                                           *)
(*    wbkClose();                                                          *)
program adcex4;
uses wbk;
const
   CHANS  = 8;
   SCANS  = 9;
   FREQ   = 2.0;
var
   buf:array[0..CHANS * SCANS - 1] of integer;
   i, j:word;
   active:byte;
   retCount: longint;
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
```

```
begin
   writeln;
   writeln('ADCEX4.PAS');
   writeln;
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Set the acquisition to NShot on trigger and the post-trigger scan count *)
   wbkSetAcq(WamNShot, 0, SCANS);
   (* Set the scan configuration *)
   wbkSetMux(1, CHANS, WgcX1, 1);
   (* Set the post-trigger scan rates *)
   wbkSetFreq(1.0, FREQ);
   (* Set the trigger source to a software trigger command *)
   wbkSetTrigHardware(WtsSoftware, 0.0);
   (* Arm the acquisition *)
   wbkArm;
   (* Start reading data in the background mode with cycle mode off *)
   (* and updateSingle on                                            *)
   wbkBufferTransfer(@buf, SCANS, 0, 1, 0, @active, @retCount);
   (* Issue a software trigger command to the hardware *)
   wbkSoftTrig;
   (* Monitor the progress of the background transfer *)
   repeat
      wbkGetBackStat(@active, @retCount);
      write('Transfer in progress: ', retCount, ' scans acquired.', chr(13));
   until active = 0;
   writeln;
   writeln('Acquisition complete.');
   writeln;
   (* Print results *)
   writeln('Data acquired:');
   for i := 0 to CHANS - 1 do begin
      write('Channel ', (i + 1):2, ' Data:');
      for j := 0 to retCount - 1 do write(' ', buf[(j * CHANS) + i]:6);
      writeln;
   end;
   (* Close and Exit *)
   wbkClose;
end.
```

ADCEX5.PAS

```
(* This example takes multiple scans from hardware using a complex analog *)
(* trigger.  The acquisition will start on a rising-edge of channel 1 at  *)
(* 2 volts OR a falling edge on channel 2 at 3 volts.                     *)
(* Functions used:                                                        *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                       *)
(*    wbkSetMux(startChan, endChan, gain, polarity);                      *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                              *)
(*    wbkSetTrigComplex(chans, gains, polarities, rising, levels,         *)
(*                      hysteresis, count, opstr);                        *)
(*    wbkArm();                                                           *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,  *)
(*                      active, retCount);                                *)
(*    wbkInit(lptPort, lptIntr);                                          *)
(*    wbkClose();                                                         *)
program adcex5;
uses wbk;
const
   FREQ    = 1000;
   SCANS   = 9;
   CHANS   = 3;
   NUM_TRIG= 2;
var

   buf:array[0..CHANS * SCANS - 1] of integer;
   i, j: integer;
   retCount:longint;
   active:byte;
   chans_tr:array[0..NUM_TRIG - 1] of word;
   gains_tr:array[0..NUM_TRIG - 1] of byte;
   polarity_tr:array[0..NUM_TRIG - 1] of byte;
```

```
   rising:array[0..NUM_TRIG - 1] of byte;
   levels:array[0..NUM_TRIG - 1] of single;
   hysteresis:array[0..NUM_TRIG - 1] of single;
   opstr:string;

procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
   writeln;
   writeln('ADCEX5.PAS');
   writeln;
   (* Initialize the complex trigger arrays for a rising-edge on channel 1 *)
   (* at 2 volts OR a falling-edge on channel 2 at 3 volts                 *)
   chans_tr[0] := 1;
   gains_tr[0] := WgcX1;
   polarity_tr[0] := 1;
   rising[0] := WcrRisingEdge;
   levels[0] := 2.0;
   hysteresis[0] := 0.1;
   chans_tr[1] := 2;
   gains_tr[1] := WgcX1;
   polarity_tr[1] := 1;
   rising[1] := WcrFallingEdge;
   levels[1] := 3.0;
   hysteresis[1] := 0.1;
   opstr := '+';
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Set the acquisition to NShot on trigger and the post-trigger scan count *)
   wbkSetAcq(WamNShot, 0, SCANS);
   (* Set the scan configuration *)
   wbkSetMux(1, CHANS, WgcX1, 1);
   (* Set post-trigger scan rates *)
   wbkSetFreq(1.0, FREQ);
   writeln('Waiting for complex trigger at channels 1 and 2...');
   writeln;
   (* Set the trigger source to the complex trigger previously defined *)
   wbkSetTrigComplex(@chans_tr, @gains_tr, @polarity_tr, @rising, @levels,
           @hysteresis, NUM_TRIG, opstr);
   (* Arms the acquisition *)
   wbkArm;
   (* Read SCANS scans from the hardware with cycle mode off, *)
   (* updateSingle on and foreground enabled                 *)
   wbkBufferTransfer(@buf, SCANS, 0, 1, 1, @active, @retCount);
   (* Print results *)
   writeln('Results of BufferTransfer:');
   for i := 0 to CHANS - 1 do begin
      write('Channel ', (i + 1):2, ' Data: ');
      for j := 0 to retCount - 1 do write(' ', buf[(j * CHANS) + i]:6);
      writeln;
   end;
   (* Close and exit *)
   wbkClose;
end.
```

ADCEX6.PAS

```
(* This example demonstrates an acquisition made up of pre-trigger and    *)
(* post-trigger scans from multiple channels using a DSP-based analog     *)
(* trigger.  It also uses data packing and rotating.                      *)
(* Functions used:                                                        *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                       *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                              *)
(*    wbkSetScan(chans, gains, polarities, chanCount);                    *)
(*    wbkSetTrigAnalog(chan, gain, polarity, rising, level, hysteresis);  *)
(*    wbkArm();                                                           *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,  *)
(*                      active, retCount);                                *)
(*    wbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount,       *)
```

```
(*                    retCount);                                        *)
(*    wbkBufferRotate(buf, scanCount, chanCount, retCount);             *)
(*    wbkGetBackStat(active, retCount);                                 *)
(*    wbkInit(lptPort, lptIntr);                                        *)
(*    wbkClose();                                                       *)
program adcex6;
uses wbk;
const
   CHANS       = 4;
   PRE_SCANS   = 5;
   POST_SCANS  = 9;
   PRE_FREQ    = 100.0;
   POST_FREQ   = 200.0;
   BLOCK       = PRE_SCANS+POST_SCANS;
var
   buf:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   chs:array[0..CHANS-1] of word;
   gains:array[0..CHANS-1] of byte;
   polarities:array[0..CHANS-1] of byte;
   active:byte;
   retCount:longint;
   pre_trig_freq, post_trig_freq:double;
   pre_trig_period, post_trig_period:double;
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
   writeln;
   writeln('ADCEX6.PAS');
   writeln;
   (* Scan sequence definition *)
   chs[0] := 1;
   chs[1] := 3;
   chs[2] := 5;
   chs[3] := 7;
   for i := 0 to CHANS-1 do begin
     gains[i] := WgcX1;   (* unity gain *)
     polarities[i] := 1;  (* bipolar    *)
   end;
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Enable Data Packing *)
   wbkSetDataPacking(1);
   (* Set the acquisition for pre/post-trigger mode and the scan counts *)
   wbkSetAcq(WamPrePost, PRE_SCANS, POST_SCANS);
   (* Set the scan configuration *)
   wbkSetScan(@chs, @gains, @polarities, CHANS);
   (* Set the pre-trigger and post-trigger scan rates *)
   wbkSetFreq(PRE_FREQ, POST_FREQ);
   (* Set the trigger source to an analog trigger on channel 1 at 2 volts *)
   wbkSetTrigAnalog(1, WgcX1, 1, WctRisingEdge, 2.0, 0.1);
   (* Arm the acquisition *)
   wbkArm;
   (* Start reading data in the background mode with cycle mode on *)
   (* and updateSingle off                                         *)
   wbkBufferTransfer(@buf, BLOCK, 1, 1, 0, @active, @retCount);
   (* Monitor the progress of the background transfer *)
   while (active  0) do begin
      wbkGetBackStat(@active, @retCount);
      write('Transfer in progress: ', retCount, ' scans acquired.', chr(13));
   end;
   writeln;
   writeln('Acquisition complete.');
   (* Unpack the packed data using the same buffer *)
   wbkBufferUnpack(@buf, @buf, BLOCK, CHANS, retCount);
   (* Rotate the unpacked buffer data so that the earliest is at the start *)
   (* and the latest is at the end of the buffer.                          *)
   wbkBufferRotate(@buf, BLOCK, CHANS, retCount);
   (* Print results *)
   writeln('Pre_trigger data acquired:');
```

```
   for i := 0 to CHANS - 1 do begin
      write('Channel ', chs[i]:2, ' Data:');
      for j := 0 to PRE_SCANS - 1 do write(' ', buf[(j * CHANS) + i]:6);
      writeln;
   end;
   writeln('Post-trigger data acquired:');
   for i := 0 to CHANS - 1 do begin
      write('Channel ', chs[i]:2, ' Data:');
      for j := PRE_SCANS to BLOCK - 1 do write(' ', buf[(j * CHANS) + i]:6);
      writeln;
   end;
   (* Close and Exit *)
   wbkClose;
end.
```

ADCEX7.PAS

```
(* 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:                                                           *)
(*    wbkSetAcq(mode, preTrigCount, postTrigCount);                          *)
(*    wbkSetMux(startChan, endChan, gain, polarity);                         *)
(*    wbkSetFreq(preTrigFreq, postTrigFreq);                                 *)
(*    wbkSetTrigHardware(source, level);                                     *)
(*    wbkArm();                                                              *)
(*    wbkSoftTrig();                                                         *)
(*    wbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground,     *)
(*                      active, retCount);                                   *)
(*    wbkGetBackStat(active, retCount);                                      *)
(*    wbkInit(lptPort, lptIntr);                                             *)
(*    wbkClose();                                                            *)
program adcex7;
uses wbk;
const
   CHANS       = 8;
   SCANS       = 20000;
   BLOCK       = 1000;
   FREQ        = 5000.0;
var
   buf0:array[0..CHANS * BLOCK - 1] of integer;
   buf1:array[0..CHANS * BLOCK - 1] of integer;
   i, j:word;
   active:byte;
   retCount:longint;
   tmpActive:byte;
   tmpRetCount:longint;
   totals:array[0..CHANS - 1] of longint;
   whichBuf:integer;
procedure myhandler( error_code:integer );
begin
   writeln('Error! Program aborted');
   writeln('WaveBook Error: ', error_code);
   halt;
end;
begin
   writeln;
   writeln('ADCEX7.PAS');
   writeln;
   (* Set error handler and initialize WaveBook *)
   wbkSetErrHandler(myhandler);
   wbkInit(LPT1, 7);
   (* Set the acquisition to NShot on trigger and the post-trigger scan count *)
   wbkSetAcq(WamNShot, 0, SCANS);
   (* Set the scan configuration *)
   wbkSetMux(1, CHANS, WgcX1, 1);
   (* Set post-trigger scan rates *)
   wbkSetFreq(1.0, FREQ);
   (* Set the trigger source to a software trigger command *)
   wbkSetTrigHardware(WtsSoftware, 0.0);
   (* Arm the acquisition *)
   wbkArm;
   (* Issue a software trigger command to the hardware *)
   wbkSoftTrig;
   (* Start reading data into the first buffer *)
```

```
      wbkBufferTransfer(@buf0, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount);
      whichBuf := 0;
      repeat
         (* Swap the buffer selector, whichBuf selects the transfer buffer *)
         if whichBuf = 1 then whichBuf := 0 else whichBuf := 1;
         (* Wait for the acquisition to go inactive or the buffer to be filled *)
         repeat
            wbkGetBackStat(@active, @retCount);
         until (active = 0) or (retCount = BLOCK);
         (* If the previous acquisition is still active, start another transfer *)
         (* into the next buffer                                                 *)
         if (active  0) then begin
            if (whichBuf = 0) then
               wbkBufferTransfer(@buf0, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount)
            else
               wbkBufferTransfer(@buf1, BLOCK, 0, 0, 0, @tmpActive, @tmpRetCount);
         end;
         (* Process the data into the process buffer *)
         if (retCount  0) then begin
            (* Average the readings in the process buffer and print the results *)
            for j := 0 to CHANS - 1 do totals[j] := 0;
            for i := 0 to retCount - 1 do begin
               for j := 0 to CHANS - 1 do begin
                  if (whichBuf = 1) then
                     totals[j] := totals[j] + buf0[i * CHANS + j]
                  else
                     totals[j] := totals[j] + buf1[i * CHANS + j];
               end;
            end;
            write('Averages:');
            for j := 0 to CHANS - 1 do begin
               write(' ', ((5.0 * totals[j]) / (32768.0 * retCount)):6:3);
            end;
            writeln;
         end;
      until (active = 0);
      (* Close and Exit *)
      wbkClose;
   end.
```

Notes

Using WaveBook/512 with VB Subroutines Support 9

This chapter describes the use of the Visual Basic language with the **standard** API to develop a basic data acquisition program. For additional functions of the standard API, refer to chapter 10. **Note**: The WaveBook system includes full-featured DOS and Windows software drivers for C (chapter 6), QuickBASIC (chapter 7), Turbo Pascal (chapter 8), and Visual Basic. The enhanced API (for C, Visual Basic and Delphi) is described in chapters 11 and 12.

Accessing WaveBook 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 WaveBook. How these actions are combined and coordinated in response to Windows messages is up to the application designer.

Accessing WaveBook from a Visual Basic Program

WaveBook provides support for Microsoft's Visual Basic. Visual Basic includes a tool palette for designing your application's user interface, letting you use point-and-click operations to design and test your entire user interface. For example, to place a button in one of your application's windows, you simply select the button tool from the tool palette, then click and drag in the desired window to place and size the button.

To program the WaveBook from Visual Basic for Windows, the file WAVEBOOK.BAS must be included in the Visual Basic project along with any forms that make up your program.

To run the example program included in the WAVEBOOK\WIN\VB directory, create a new project by selecting New under the file menu. Remove the default form, FORM1, in the project window and add the files WBKEX.FRM and WAVEBOOK.BAS to the project. Finally, select form1 of WBKEX.FRM to be the startup form of the project by selecting PROJECT under the OPTIONS menu.

Simple Analog Input

The following subroutine, ADC1_CLICK within WBKEX.FRM, shows the usage of several high level analog input routines.

This subroutine will initialize the WaveBook hardware, then take readings from the analog input channels in the base unit (not the expansion cards).

Although not necessary, this subroutine includes an error handler which vectors program control to a user defined routine when a WaveBook error is detected. If no error handler is supplied, Visual Basic will receive and handle the error, posting the error on the screen and terminating the subroutine.

Visual Basic provides an integer variable, ERR, which contains the error code of the most recent error. This variable can be used in user defined error handlers to detect the error source and take the appropriate action. The function VBwbkSetErrHandler tells Visual Basic to assign ERR to a specific value when a WaveBook error is encountered.

For transporting data in and out of the WaveBook driver, arrays are dimensioned.

```
ReDim buf%(SCANS& * CHANS%)

Dim i%, j%
Dim sample%
Dim ret%
```

The following line tells Visual BASIC to set ERR to 100 when a WaveBook error is encountered.

```
ret% = VBwbkSetErrHandler%(100)
```

The ON ERROR GOTO command is part of the Visual BASIC command set. It 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 vector program control to the label ErrorHandler if an error is encountered.

```
On Error GoTo ErrorHandlerADC1
```

The next line tells the driver library which LPT port or base address and what interrupt line is being used, then initializes the WaveBook hardware.

```
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

The next command will retrieve 1 sample at a gain of 1. The polarity set to bipolar.

```
ret% = VBwbkRd%(1, sample%, GAIN%, BIPOLAR%)
```

The results of the are then printed.

```
Print "Result of Rd: "; sample%
Print
```

Next, 8 samples are acquired from channel 1 using software triggering. A 1000Hz sampling frequancy is used with unity gain for bipolar signals. The data returned data is stored in the integer array buffer.

```
ret% = VBwbkRdN%(1, buf%(0), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%, BIPOLAR%)
The results are then printed.
Print "Results of RdN:"
Print "Channel 1 Data: ";
For i% = 0 To SCANS&  - 1
   Print Tab(i% * 7 + 17); buf%(i%);
Next i%
Print
Print
```

The following command collects 1 sample from multiple channels. The following function arguments define a scan starting with channel 1 and ending with channel 8. This command sets all the channels to the same gain and unipolar/bipolar setting. The data is returned in the array buffer.

```
ret% = VBwbkRdScan%(1, CHANS%, buf%(0), GAIN%, BIPOLAR%)
```

The results are printed.

```
Print "Results of RdScan:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data: "; buf%(i%)
Next i%
Print
```

The following command collects multiple scans consisting of multiple channels. Like wbkRdScan, this command sets all the channels to the same gain and unipolar/bipolar setting. Since multiple scans are being collected, a trigger source and timebase is required. The arguments shown set the trigger source to Software causing an immediate trigger. The sample frequency of 1 Khz.

```
ret% = VBwbkRdScanN%(1, CHANS%, buf%(0), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%,
BIPOLAR%)
```

The results are printed.

```
Print "Results of RdScanN:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data: ";
   For j% = 0 To SCANS&    - 1
      Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
```

The subroutine is closed and exited.

```
ret% = VBwbkClose%()
Exit Sub
```

Low-Level Analog Input

The following subroutine, ADC2_CLICK within WBKEX.FRM, shows the usage of the lowest level WaveBook driver functions. These functions are more complex than the high level functions but allow the greatest flexibility.

Constants and arrays are defined.

```
Const CHANS% = 8            ' number of channels in scan
Const SCANS& = 10           ' number of scans
Const BLOCK% = 6            ' number scans reading for one time
Const FREQ# = 5#            ' postTrig scan rates in Hz

ReDim buf%(BLOCK% * CHANS%)

Dim i%, j%
Dim active%
Dim retCount&
Dim preTrigFreq#, postTrigFreq#
Dim preTrigPeriod#, postTrigPeriod#
Dim ret%
```

Set the error handler and initialize the WaveBook.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC2
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following line shows the usage of the wbkGetFreq command which returns the present settings for the pre- and post-trigger frequencies.

```
ret% = VBwbkGetFreq%(preTrigFreq#, postTrigFreq#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = VBwbkArm%()
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = VBwbkSoftTrig%()
```

The following lines perform a foreground transfer from the WaveBook of size BLOCK. The data is then printed on the screen. The transfers will continue until the acquisition is no longer active.

```
Do
   ' Read BLOCK scans from the hardware with cycle mode off,
   ' updateSingle on and foreground enabled
   ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 0, 1, 1, active%, retCount&)

   ' Print results
   Print "Result of BufferTransfer: retCount="; retCount&; " active="; active%
   For i% = 0 To retCount&    -1
      Print "Scan"; i% + 1; "Data:";
      For j% = 0 To CHANS%    -1
         Print Tab(j% * 7 + 17); buf%(i% * CHANS% + j%);
      Next j%
      Print
   Next i%
   Print
Loop While active%  0
```

Accessing the High-Speed Digital Input Port

The following subroutine, ADC3_CLICK within WBKEX.FRM, shows how to collect analog and high speed digital signals concurrently, in the same scan.

```
Constants and arrays are defined.
Const FREQ# = 5
Const SCANS& = 10
Const CHANS% = 3

ReDim buf%(SCANS& * CHANS%)

Dim i%, j%
Dim active%
Dim retCount&

Dim version%
ReDim chan%(CHANS%)
ReDim gains%(CHANS%), polarities%(CHANS%)
Dim ret%
```

Set scan sequence definitions

```
chan%(0) = 0        ' high speed digital channel
chan%(1) = 5        ' analog channel 5
chan%(2) = 8        ' analog channel 8
```

Set gains and polarities.

```
For i% = 0 To CHANS%  - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
Next i%
```

Get the driver version and print.

```
ret% = VBwbkGetDriverVersion%(version%)
Print "Using driver version: "; Format$(.01 * version, "0.00")
Print
```

Set error handler and initialize WaveBook.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC3
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a

specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

Setup the scan by passing the channel configuration arrays to the wbkSetScan command.

```
ret% = VBwbkSetScan%(chan%(), gains%(), polarities%(), CHANS%)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = VBwbkArm%()
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = VBwbkSoftTrig%()
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete.

```
ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 1, active%, retCount&)
```

The following lines print the transferred data.

```
Print "Results of BufferTransfer:"
Print "                Digital_ch_0  Analog_ch_5  Analog_ch_8"
For i% = 0 To retCount&  - 1
   ' shift the upper (valid) 8 bits of the digital input to the lower 8 bits
   buf%(i% * CHANS%) = ((buf%(i% * CHANS%) And &HFF00) \ 256) And &HFF
   Print "Scan"; i% + 1; "Data:";
   For j% = 0 To CHANS%    - 1
      Print Tab(j% * 14 + 17); buf%(i% * CHANS% + j%);
   Next j%
   Print
Next i%
```

Background Processing of Analog Input

The following subroutine, ADC4_CLICK within WBKEX.FRM, shows how to collect analog samples and transfer them into the PC's memory in the background. Once the background acquisition is configured and armed, your program can perform other operations concurrently with the background data collection. The foreground program can use the wbkGetBackStat command to periodically check on the status of the acquisition.

For performing acquisitions that are greater in size than the allocated buffer, the background operation can be set to Cycle mode which will wrap around in the allocated buffer as it becomes full. In this mode, your program must monitor the background and transfer the data out of the allocated buffer before the background operation overwrites it.

Constants and declarations are first defined.

```
Const CHANS% = 8
```

```
Const SCANS& = 9
Const FREQ# = 2

ReDim buf%(SCANS& * CHANS%)

Dim i%, j%
Dim active%
Dim retCount&
Dim ret%
```

Set error handller and initialize the WaveBook.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC4
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

To create a channel scan of non sequential channels with independent gain and unipolar/bipolar settings, the arrays of channel parameters must be created and passed to wbkSetScan.

```
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = VBwbkArm%()
```

The following line sets up a background transfer. Regardless of the state of the acquisition, the program will immediately return from this function call and proceed to the next line. As the data is collected by the WaveBook, it is automatically transferred to the buffer.

```
ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 0, active%, retCount&)
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = VBwbkSoftTrig%()
```

Although your program can begin processing other tasks at this point, our example program simply monitors the background until the user hits a key or the acquisition is complete.

```
Print "Waiting for trigger."
While retCount& = 0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Triggered.  Transfer in progress."
While active%  0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
```

```
Print "Acquisition complete:"; retCount&; "scans acquired."
Print
```

The following lines print the collected data.

```
Print "Data acquired:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = 0 To SCANS&    - 1
      Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
```

Complex Triggering

The following subroutine, ADC5_CLICK within WBKEX.FRM, shows how to setup a complex trigger where more than one channel can be combined in a logical trigger equation.

Define constants and arrays.

```
Const FREQ# = 1000
Const SCANS& = 9
Const CHANS% = 3
Const NUM_TRIG% = 2

ReDim buf%(SCANS& * CHANS%)

Dim i%, j%
Dim active%
Dim retCount&
ReDim chan_tr%(NUM_TRIG%)
ReDim gains_tr%(NUM_TRIG%), polarity_tr%(NUM_TRIG%)
ReDim rising%(NUM_TRIG%)
ReDim levels!(NUM_TRIG%), hysteresis!(NUM_TRIG%)
Dim opstr$
Dim ret%
```

The following lines are definitions and initialization for the variables used for setting up the trigger equation. The variable chans_tr is an array of channels used in the trigger equation. Channels 1 and 2 are specified. The variable gain_tr is an array that holds the gains for channels 1 and 2. In this case they are both set to X1. Channels 1 and 2 can also be a part of the scan group with the same or different gain assignments. The variable polarity_tr is an array that holds the unipolar/bipolar settings for channels 1 and 2. The variable rising is an array that holds the edge settings for channels 1 and 2. In this case, channel 1 triggers on the rising edge, while channel 2 triggers on the falling edge. The variables levels and hysteresis are arrays that hold the voltage thresholds and hysteresis settings for channels 1 and 2, respectively. The variable opstr holds the boolean operator for the trigger equation. The + sign indicates an OR operator between channels 1 and 2.

```
chan_tr%(0) = 1
gains_tr%(0) = WgcX1%
polarity_tr%(0) = 1
rising%(0) = WctRisingEdge%
levels!(0) = 2
hysteresis!(0) = .1

chan_tr%(1) = 2
gains_tr%(1) = WgcX1%
polarity_tr%(1) = 1
rising%(1) = WctFallingEdge%
levels!(1) = 3
hysteresis!(1) = .1

opstr$ = "+"
```

The previous definitions create the following trigger setup:

System Trigger = (CH1 @ X1, bipolar, rising edge through 2.0V with 0.1V hyst) OR (CH2 @ X1, bipolar, falling edge through 3.0V with 0.1V hyst)

Next, the error handler is set and the WaveBook is initialized.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC5
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

Set the scan configuration.

```
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following lines notify the user of the system's status then setup the complex trigger.

```
Print "Waiting for complex trigger of channels 1 or 2..."
Print
ret% = VBwbkSetTrigComplex%(chan_tr%(), gains_tr%(), polarity_tr%(), rising%(),
levels!(), hysteresis!(), NUM_TRIG%, opstr$)
```

This command arms the system to acquire data. Since no pre-trigger scans were configured, no data will be available until the trigger is satisfied.

```
ret% = VBwbkArm%()
```

The next line performs a foreground data transfer from the WaveBook's internal buffer to the PC's memory. The foreground transfer will continue until its buffer is full or the acquisition is complete. If the trigger is not satisfied within the programmed timeout, the driver will return control to the program. If you do not want your program to "hang" until the trigger is satisfied, it is recommended that a background transfer be used. Once your program initiates a background transfer, control is passed back to your program to perform other tasks while waiting for a trigger or collecting data.

```
ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 1, active%, retCount&)
```

Print the transferred data.

```
Print "Results of BufferTransfer:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = 0 To SCANS&    - 1
      Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
```

Pre- and Post-Trigger Acquisitions

The following subroutine, ADC6_CLICK within WBKEX.FRM, shows how to setup and process acquisitions with both pre- and post-trigger scans.

Define constants and arrays.

```
Const CHANS% = 4
Const PRE_SCANS& = 5
Const POST_SCANS& = 9
Const PRE_FREQ# = 100#
Const POST_FREQ# = 200#
Const BLOCK% = (PRE_SCANS& + POST_SCANS&)
```

```
ReDim buf%(BLOCK% * CHANS%)

Dim i%, j%
Dim active%
Dim retCount&
ReDim chan%(CHANS%)
ReDim gains%(CHANS%), polarities%(CHANS%)
Dim ret%
```

Channels, gains, and polarity are defined for the scan.

```
chan%(0) = 1        ' channel numbers
chan%(1) = 3
chan%(2) = 5
chan%(3) = 7
For i% = 0 To CHANS%  - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
Next i%
```

Set error handler and initialize WaveBook.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC6
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Enable data packing.

```
ret% = VBwbkSetDataPacking%(1)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamPrePost-specifies that both pre- and post-trigger scans are to be collected. An acquisition is defined as a specified number of pre- and post-trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamPrePost%, PRE_SCANS&, POST_SCANS&)
```

To create a channel scan of non sequential channels with independent gain and unipolar/bipolar settings, channel parameters must be passed to wbkSetScan.

```
ret% = VBwbkSetScan%(chan%(), gains%(), polarities%(), CHANS%)
```

This command sets the pre- and post-trigger sample frequencies.

```
ret% = VBwbkSetFreq%(PRE_FREQ#, POST_FREQ#)
```

The following line sets up a simple analog trigger using channel 1 as the trigger source. The scan is set to trigger at 2 V.

```
ret% = VBwbkSetTrigAnalog%(1, WgcX1%, 1, WctRisingEdge%, 2!, .1)
```

This command arms the system to acquire data. Since pre-trigger scans are to be collected, scans will be immediately available for transfer into the PC's memory.

```
ret% = VBwbkArm%()
```

When pre-trigger scans are included in the acquisition, scans begin to be acquired the moment the system is armed. Scans will continue to be acquired until the trigger is satisfied and the post-trigger is complete. Your application program must transfer the acquired data into a buffer in the PC as it is collected. Until the trigger occurs, your application must be prepared to accept data continuously, potentially far in excess of the sum of the specified pre-trigger and post-trigger scan counts. This is best accomplished by setting up a background transfer in cycle mode which will automatically transfer the scans as they are collected and wrap the buffer as it becomes full. The following line sets up a background transfer of the acquired scans into buf. Cycle mode is turned on, allowing the buffer to wrap around as it becomes full.

```
ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 1, 0, 0, active%, retCount&)
```

The following lines monitor the background operation, waiting for the acquisition to be complete.

```
Print "Waiting for trigger."
```

```
While retCount& = 0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Triggered.  Transfer in progress."
While active%  0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Acquisition complete:"; retCount&; "scans acquired."
Print
```

The following line unpacks the data so that each sample occupies an integer.

```
ret% = VBwbkBufferUnpack%(buf%(0), buf%(0), BLOCK%, CHANS%, retCount)
```

Since the buffer has potentially wrapped around, the earliest data is not at the beginning of the buffer. The following line reorganizes the buffer so that the 1st pre-trigger scan occupies the 1st buffer location and the last post-trigger scan occupies the last buffer location.

```
ret% = VBwbkBufferRotate%(buf%(0), BLOCK%, CHANS%, retCount)
```

The following lines print the acquired data.

```
Print "Pre-trigger data acquired:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = 0 To PRE_SCANS&    - 1
      Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
Print
Print "Post-trigger data acquired:"
For i% = 0 To CHANS%  - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = PRE_SCANS& To BLOCK%    - 1
      Print Tab((j%   PRE_SCANS&) * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
```

Buffer Management

The following excerpts are from the example program ADC7_CLICK found in the WaveBook directory of your hard drive. 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.

Constants and buffers are defined.

```
Const CHANS% = 8
Const SCANS& = 20000
Const BLOCK% = 1000
Const FREQ# = 5000#

ReDim buf0%(CHANS% * BLOCK%)
ReDim buf1%(CHANS% * BLOCK%)

Dim i%, j%
Dim active%
Dim retCount&
Dim tmpActive%
Dim tmpRetCount&
Dim ret%
ReDim totals&(CHANS%)
Dim whichBuf%
```

The error handler is set up.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC7
```

The following command initializes the WaveBook and puts it online. LPT1 specifies the port number which the WaveBook is connected to and 7 is the interrupt level used.

```
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = VBwbkArm%()
```

The next line triggers the system for data collection. When the trigger is satisfied, data immediately starts flowing into the WaveBook's internal buffer. This data must be transferred to the PC before the internal buffer overflows. If a background acquisition is configured, the data will automatically be transferred into the allocated PC buffer. If a foreground data transfer is desired, this transfer to PC memory must keep up with the acquisition rate to avoid a WaveBook buffer overrun.

```
ret% = VBwbkSoftTrig%()
```

Start the data transfer into the first buffer

```
ret% = VBwbkBufferTransfer%(buf0%(0), BLOCK%, 0, 0, 0, tmpActive%, tmpRetCount&)
whichBuf% = 0
```

The contents of the two buffers are swapped. The loop continues until the acquisition completes or the buffer fills.

```
Do
   ' Swap the buffer selector, whichBuf selects the transfer 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% = VBwbkGetBackStat(active%, retCount&)
   Loop While ((active%  0) And (retCount&  BLOCK%))
```

If the previous acquisition is still active, another transfer into the next buffer begins.

```
   If (active%  0) Then
      If whichBuf% = 0 Then
         ret% = VBwbkBufferTransfer(buf0%(0), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      Else
```

```
         ret% = VBwbkBufferTransfer(buf1%(0), BLOCK%, 0, 0, 0, tmpActive%,
tmpRetCount&)
      End If
   End If
```

The following commands average the data in the process buffer and print the results.

```
   If (retCount&  0) Then
      ' Average the readings in the process buffer and print the results
      For j% = 0 To CHANS%        - 1
         totals&(j%) = 0
      Next j%
      For i% = 0 To retCount&      - 1
         For j% = 0 To CHANS%        - 1
            If whichBuf% = 0 Then
               totals&(j%) = totals&(j%) + buf1%(i% * CHANS% + j%)
            Else
               totals&(j%) = totals&(j%) + buf0%(i% * CHANS% + j%)
            End If
         Next j%
      Next i%
      Print "Averages:";
      For j% = 0 To CHANS%  -      1
         Print Tab(j% * 7 + 17); Format$((5# / 32768#) * totals&(j%) / retCount&,
"#0.000");
      Next j%
      Print
   End If
Loop While (active%  0)
```

The WaveBook is taken offline and reset.

```
'Close and exit
ret% = VBwbkClose()
```

Direct-to-Disk

The following excerpts are from the example program ADC8_CLICK found in the WaveBook directory of your hard drive. This example reads multiple scans from multiple channels and writes the data directly to a disk file.

```
Constants and buffers are defined.
Const CHANS% = 2
Const SCANS& = 100000
Const FREQ# = 10000#
Const BLOCK% = 2000     ' CHANS% * BLOCK% must be a multiple of 4

ReDim buf%(CHANS% * BLOCK%)

Dim i%, j%
Dim active%
Dim retCount&
Dim ret%
Dim fileHandle%
Dim byteCount%, wordCount%, sampleCount%, scanCount%
Dim termChar$
Dim voltage!
```

Set error handler.

```
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC8
```

The following command initializes the WaveBook and puts it online. LPT1 specifies the port number which the WaveBook is connected to and 7 is the interrupt level used.

```
ret% = VBwbkInit%(LPT1%, IRQ7%)
```

Enable data packing

```
ret% = VBwbkSetDataPacking%(1)
```

Mode, the 1st argument, defines the way in which the WaveBook reacts to a trigger. The definition WamNShot specifies that an entire acquisition be performed on every trigger until the system is disarmed. An acquisition is defined as a specified number of pre- and post- trigger scans sampled at a specified timebase. The 2nd and 3rd arguments define the number of pre- and post-trigger scans, respectively.

```
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
```

The following command defines the channels in a scan. The 1st and 2nd arguments define the start and end channels of the scan. Unlike the command wbkSetScan, this command does not allow a separate gain and unipolar/bipolar setting per channel, nor does it allow channels to be added to the scan in a random order.

```
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
```

This command sets the pre- and post-trigger sample frequencies. Since this application does not collect pre-trigger scans, the 1st argument is ignored.

```
ret% = VBwbkSetFreq%(1#, FREQ#)
```

The following line sets the trigger source to Software. This trigger is satisfied by the execution of the command wbkSoftTrig. The 2nd argument is a voltage level used when the trigger source is an analog channel. In this case, the voltage level argument is ignored.

```
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
```

This command arms the system to acquire data. For the present configuration, the data will not be collected until the software trigger has been satisfied by the execution of the wbkSoftTrig command.

```
ret% = VBwbkArm%()
```

The following command creates a filename ADCEX8.BIN for the file that will hold the data. No pre-write is used.

```
ret% = VBwbkSetDiskFile%("adcex8.bin", WdfWriteFile%, 0)
```

Start reading data in the background mode with cycle mode on and updateSingle off

```
ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 1, 0, 0, active%, retCount&)
```

The next line triggers the system for data collection.

```
ret% = VBwbkSoftTrig%()
```

The following commands monitor the progress of the background transfer. The program prints a running total of the number of scans acquired.

```
Print "Waiting for trigger."
While retCount& = 0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Triggered.  Transfer in progress."
While active%  0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
```

Once the transfer has finished, a completion message is printed.

```
Print "Acquisition complete:"; retCount&; "scans acquired."
```

The WaveBook is taken off line and reset.

```
ret% = VBwbkClose%()
```

After the acquisition has finished, the collected binary data will be converted to ascii format for display as a text file. The process starts with reading binary data from the ADCEX8.BIN file. If the file cannot be opened, an error is issued

```
Print "Converting adcex8.bin to adcex8.txt"

' Open the binary input file
```

```
Open "adcex8.bin" For Input As 1
fileHandle% = FileAttr(1, 2)
Next, a text file is created to hold the converted data.
 ' Open the text output file
Open "adcex8.txt" For Output As 2
```

The binary data is converted to ascii and transferred to the text file. Errors are generated if the program cannot read from ADCEX8.BIN or write to ADCEX8.TXT.

```
Do
   ' Convert BLOCK unpacked scans to packed bytes
   scanCount% = BLOCK%
   sampleCount% = scanCount% * CHANS%
   wordCount% = sampleCount% * 3 / 4
   byteCount% = 2 * wordCount%
```

Read the packed bytes from the input file and get the number of bytes actually read

```
   byteCount% = fRead%(fileHandle%, buf%(0), byteCount%)
```

Convert the number of bytes read from packed bytes to unpacked scans

```
   wordCount% = byteCount% / 2
   sampleCount% = wordCount% * 4 / 3
   scanCount% = sampleCount% / CHANS%
```

Unpack the packed data using the same buffer. This command can be called even if the WaveBook if not online or connected.

```
       ret% = VBwbkBufferUnpack%(buf%(0), buf%(0), BLOCK%, CHANS%, scanCount%)
```

The unpacked data is next written to the text file. The voltage values are then calculated and printed. The program prints the "." character to indicate that it is still active.

```
   For i% = 0 To scanCount%   - 1
      For j% = 0 To CHANS%   - 1
         ' Send a tab between channels and a newline after each scan
         If (j%  CHANS%   1) Then
            termChar$ = Chr$(9)
         Else
            termChar$ = Chr$(13) + Chr$(10)
         End If
         ' calculate and write out the voltage value
         voltage! = buf%(i% * CHANS% + j%) * 5! / 32768!
         Print #2, Format$(voltage!, ".000") + termChar$;
      Next j%
   Next i%

   ' Print something so that the program doesn't appear locked
   Print ".";
Loop While (byteCount%  0)   ' A byteCount of 0 indicates end of file
```

Finally, the files are closed and a completion message printed.

```
Close 1
Close 2
Print "complete."
```

Sample Programs

Sub ADC1_Click ()

```
Sub ADC1_Click ()
  ' This example demonstrates the use of the WaveBook's one-step
  ' acquisition functions and user error handling.
  ' Function used:
  '    VBwbkRd(chan, sample, gain, polarity)
  '    VBwbkRdN(chan, buf, count, trigger, level, freq, gain, polarity)
  '    VBwbkRdScan(startChan, endChan, buf, gain, polarity)
  '    VBwbkRdScanN(startChan, endChan, buf, count, trigger, level, freq, gain,
       polarity)
  '    VBwbkSetErrHandler(errNum)
```

```
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const FREQ# = 1000#
  Const GAIN% = WgcX1%
  Const BIPOLAR% = 1
  Const SCANS& = 9
  Const CHANS% = 8
  ReDim buf%(SCANS& * CHANS%)
  Dim i%, j%
  Dim sample%
  Dim ret%
  Cls
  Print "ADC1"
  Print
  ' Set error handler and initialize WaveBook
  ret% = VBwbkSetErrHandler%(100)
  On Error GoTo ErrorHandlerADC1
  ret% = VBwbkInit%(LPT1%, IRQ7%)
  ' Get a single sample from a single channel
  ret% = VBwbkRd%(1, sample%, GAIN%, BIPOLAR%)
  ' Print result
  Print "Result of Rd: "; sample%
  Print
  ' Get multiple samples from a single channel, triggered by a software trigger
  ret% = VBwbkRdN%(1, buf%(0), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%, BIPOLAR%)
  ' Print results
  Print "Results of RdN:"
  Print "Channel 1 Data: ";
  For i% = 0 To SCANS& - 1
     Print Tab(i% * 7 + 17); buf%(i%);
  Next i%
  Print
  Print
  ' Get a single sample from multiple channels
  ret% = VBwbkRdScan%(1, CHANS%, buf%(0), GAIN%, BIPOLAR%)
  ' Print results
  Print "Results of RdScan:"
  For i% = 0 To CHANS% - 1
     Print "Channel"; i% + 1; "Data: "; buf%(i%)
  Next i%
  Print
  ' Get multiple samples from multiple channels, triggered by a software trigger
  ret% = VBwbkRdScanN%(1, CHANS%, buf%(0), SCANS&, WtsSoftware%, 0!, FREQ#, GAIN%,
BIPOLAR%)
  ' Print results
  Print "Results of RdScanN:"
  For i% = 0 To CHANS% - 1
     Print "Channel"; i% + 1; "Data: ";
     For j% = 0 To SCANS& - 1
        Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
     Next j%
     Print
  Next i%
  'Close and exit
  ret% = VBwbkClose%()
Exit Sub
ErrorHandlerADC1:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC1"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC2_Click ()

```
Sub ADC2_Click ()
  ' This example demonstrates the use of WaveBook's custom acquistion functions.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetMux(startChan, endChan, gain, polarity)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkGetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkGetPeriod(preTrigPeriod, postTrigPeriod)
  '    VBwbkSetTrigHardware(source, level)
  '    VBwbkArm()
  '    VBwbkSoftTrig()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
       retCount)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const CHANS% = 8             ' number of channels in scan
  Const SCANS& = 10            ' number of scans
  Const BLOCK% = 6             ' number scans reading for one time
  Const FREQ# = 5#             ' postTrig scan rates in Hz
  ReDim buf%(BLOCK% * CHANS%)
  Dim i%, j%
  Dim active%
  Dim retCount&
  Dim preTrigFreq#, postTrigFreq#
  Dim preTrigPeriod#, postTrigPeriod#
  Dim ret%
  Cls
  Print "ADC2"
  Print
  ' Set error handler and initialize WaveBook
  ret% = VBwbkSetErrHandler%(100)
  On Error GoTo ErrorHandlerADC2
  ret% = VBwbkInit%(LPT1%, IRQ7%)
  ' Set the acquisition to NShot on trigger and the post-trigger scan count
  ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
  ' Set the scan configuration
  ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
  ' Set the post-trigger scan rates
  ret% = VBwbkSetFreq%(1#, FREQ#)
  ' Get the pre-trigger and post-trigger scan rates in frequency and period
  ret% = VBwbkGetFreq%(preTrigFreq#, postTrigFreq#)
  Print "Result of GetFreq: pre-trigger="; preTrigFreq#; "Hz, post-trigger=";
postTrigFreq#; "Hz"
  ret% = VBwbkGetPeriod%(preTrigPeriod#, postTrigPeriod#)
  Print "Result of GetPeriod: pre-trigger="; preTrigPeriod#; "ns, post-trigger=";
postTrigPeriod#; "ns"
  Print
  ' Set the trigger source to a software trigger command
  ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
  ' Arm the acquisition
  ret% = VBwbkArm%()
  ' Issue a software trigger command to the hardware
  ret% = VBwbkSoftTrig%()
  Do
     ' Read BLOCK scans from the hardware with cycle mode off,
     ' updateSingle on and foreground enabled
     ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 0, 1, 1, active%, retCount&)
     ' Print results
     Print "Result of BufferTransfer: retCount="; retCount&; " active="; active%
     For i% = 0 To retCount& - 1
        Print "Scan"; i% + 1; "Data:";
        For j% = 0 To CHANS% - 1
           Print Tab(j% * 7 + 17); buf%(i% * CHANS% + j%);
        Next j%
        Print
     Next i%
     Print
  Loop While active%  0
  'Close and exit
  ret% = VBwbkClose%()
```

```
Exit Sub
ErrorHandlerADC2:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC2"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC3_Click ()

```
Sub ADC3_Click ()
  ' This example takes multiple scans from hardware using a software trigger.
  ' Each scan includes the high speed digital I/O port (channel 0) and
  ' two analog channels: 5 and 8.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetScan(chans, gains, polarities, count)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkSetTrigHardware(source, level)
  '    VBwbkArm()
  '    VBwbkSoftTrig()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
       retCount)
  '    VBwbkGetDriverVersion(version)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const FREQ# = 5
  Const SCANS& = 10
  Const CHANS% = 3
  ReDim buf%(SCANS& * CHANS%)
  Dim i%, j%
  Dim active%
  Dim retCount&
  Dim version%
  ReDim chan%(CHANS%)
  ReDim gains%(CHANS%), polarities%(CHANS%)
  Dim ret%
  Cls
  Print "ADC3"
  Print
  ' Scan sequence definition
  chan%(0) = 0        ' high speed digital channel
  chan%(1) = 5        ' analog channel 5
  chan%(2) = 8        ' analog channel 8
  ' Channel gains and polarities setting
  For i% = 0 To CHANS% - 1
     gains%(i%) = WgcX1%  ' unity gain
     polarities%(i%) = 1  ' bipolar
  Next i%
  ' Get driver version
  ret% = VBwbkGetDriverVersion%(version%)
  Print "Using driver version: "; Format$(.01 * version, "0.00")
  Print
  ' Set error handler and initialize WaveBook
  ret% = VBwbkSetErrHandler%(100)
  On Error GoTo ErrorHandlerADC3
  ret% = VBwbkInit%(LPT1%, IRQ7%)
  ' Set the acquisition to NShot on trigger and the post-trigger scan count
  ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
  ' Set scan configuration
  ret% = VBwbkSetScan%(chan%(), gains%(), polarities%(), CHANS%)
  ' Set the post-trigger scan rates
  ret% = VBwbkSetFreq%(1#, FREQ#)
  ' Set the trigger source to a software trigger command
  ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
  ' Arm the acquisition
  ret% = VBwbkArm%()
  ' Issue a software trigger command to the hardware
  ret% = VBwbkSoftTrig%()
  ' Read SCANS& number of scans from the hardware
```

```
  ' with cycle mode off, updateSingle on and foreground enabled
  ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 1, active%, retCount&)
  ' Print results
  Print "Results of BufferTransfer:"
  Print "                Digital_ch_0  Analog_ch_5  Analog_ch_8"
  For i% = 0 To retCount& - 1
     ' shift the upper (valid) 8 bits of the digital input to the lower 8 bits
     buf%(i% * CHANS%) = ((buf%(i% * CHANS%) And &HFF00) \ 256) And &HFF
     Print "Scan"; i% + 1; "Data:";
     For j% = 0 To CHANS% - 1
        Print Tab(j% * 14 + 17); buf%(i% * CHANS% + j%);
     Next j%
     Print
  Next i%

  'Close and exit
  ret% = VBwbkClose%()
Exit Sub
ErrorHandlerADC3:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC3"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC4_Click ()

```
Sub ADC4_Click ()
  ' This example reads scans of multiple channels in the background mode
  ' and uses a software trigger to start the acquisition.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkSetMux(startChan, endChan, gain, polarity)
  '    VBwbkSetTrigHardware(source, level)
  '    VBwbkArm()
  '    VBwbkSoftTrig()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
        retCount)
  '    VBwbkGetBackStat(active, retCount)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const CHANS% = 8
  Const SCANS& = 9
  Const FREQ# = 2
  ReDim buf%(SCANS& * CHANS%)
  Dim i%, j%
  Dim active%
  Dim retCount&
  Dim ret%
  Cls
  Print "ADC4"
  Print
  ' Set error handler and initialize WaveBook
  ret% = VBwbkSetErrHandler%(100)
  On Error GoTo ErrorHandlerADC4
  ret% = VBwbkInit%(LPT1%, IRQ7%)
  ' Set the acquisition to NShot on trigger and the post-trigger scan count
  ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
  ' Set scan's configuration
  ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
  ' Set the post-trigger scan rates
  ret% = VBwbkSetFreq%(1#, FREQ#)
  ' Set the trigger source to a software trigger command
  ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
  ' Arm the acquisition
  ret% = VBwbkArm%()
  ' Start reading data in the background mode with cycle mode off
  ' and updateSingle on
  ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 0, active%, retCount&)
```

```
  ' Issue a software trigger command to the hardware
  ret% = VBwbkSoftTrig%()
  ' Monitor the progress of the background transfer
  Print "Waiting for trigger."
  While retCount& = 0
     ret% = VBwbkGetBackStat%(active%, retCount&)
  Wend
  Print "Triggered.  Transfer in progress."
  While active%  0
     ret% = VBwbkGetBackStat%(active%, retCount&)
  Wend
  Print "Acquisition complete:"; retCount&; "scans acquired."
  Print
  ' Print results
  Print "Data acquired:"
  For i% = 0 To CHANS% - 1
     Print "Channel"; i% + 1; "Data:";
     For j% = 0 To SCANS& - 1
        Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
     Next j%
     Print
  Next i%
  ' Close and Exit
  ret% = VBwbkClose%()
Exit Sub
ErrorHandlerADC4:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC4"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC5_Click ()

```
Sub ADC5_Click ()
  ' This example takes multiple scans from hardware using a complex analog
  ' trigger.  The acquisition will start on a rising-edge of channel 1 at
  ' 2 volts OR a falling edge on channel 2 at 3 volts.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetMux(startChan, endChan, gain, polarity)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkSetTrigComplex(chans, gains, polarities, rising, levels, hysteresis,
        count, opstr)
  '    VBwbkArm()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
        retCount)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const FREQ# = 1000
  Const SCANS& = 9
  Const CHANS% = 3
  Const NUM_TRIG% = 2
  ReDim buf%(SCANS& * CHANS%)
  Dim i%, j%
  Dim active%
  Dim retCount&
  ReDim chan_tr%(NUM_TRIG%)
  ReDim gains_tr%(NUM_TRIG%), polarity_tr%(NUM_TRIG%)
  ReDim rising%(NUM_TRIG%)
  ReDim levels!(NUM_TRIG%), hysteresis!(NUM_TRIG%)
  Dim opstr$
  Dim ret%
  ' Initialize the complex trigger arrays for a rising-edge on channel 1
  ' at 2 volts OR a falling-edge on channel 2 at 3 volts
  chan_tr%(0) = 1
  gains_tr%(0) = WgcX1%
  polarity_tr%(0) = 1
  rising%(0) = WctRisingEdge%
  levels!(0) = 2
  hysteresis!(0) = .1
```

```
  chan_tr%(1) = 2
  gains_tr%(1) = WgcX1%
  polarity_tr%(1) = 1
  rising%(1) = WctFallingEdge%
  levels!(1) = 3
  hysteresis!(1) = .1
  opstr$ = "+"
  Cls
  Print "ADC4"
  Print
  ' Set error handler and initialize WaveBook
  ret% = VBwbkSetErrHandler%(100)
  On Error GoTo ErrorHandlerADC5
  ret% = VBwbkInit%(LPT1%, IRQ7%)
  ' Set the acquisition to NShot on trigger and the post-trigger scan count
  ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
  ' Set the scan configuration
  ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
  ' Set the post-trigger scan rates
  ret% = VBwbkSetFreq%(1#, FREQ#)
  ' Set a complex trigger at channels 1 and 2
  Print "Waiting for complex trigger of channels 1 or 2..."
  Print
  ret% = VBwbkSetTrigComplex%(chan_tr%(), gains_tr%(), polarity_tr%(), rising%(),
levels!(), hysteresis!(), NUM_TRIG%, opstr$)
  ' Arm the acquisition
  ret% = VBwbkArm%()
  ' Read SCANS& number of scans from the hardware
  ' with cycle mode off, updateSingle on and foreground enabled
  ret% = VBwbkBufferTransfer%(buf%(0), SCANS&, 0, 1, 1, active%, retCount&)
  ' Print results
  Print "Results of BufferTransfer:"
  For i% = 0 To CHANS% - 1
     Print "Channel"; i% + 1; "Data:";
     For j% = 0 To SCANS& - 1
        Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
     Next j%
     Print
  Next i%
  'Close and exit
  ret% = VBwbkClose%()
Exit Sub
ErrorHandlerADC5:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC5"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC6_Click ()

```
Sub ADC6_Click ()
  ' This example demonstrates an acquisition made up of pre-trigger and
  ' post-trigger scans from multiple channels using a DSP-based analog
  ' trigger.  It also uses data packing and rotating.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkSetScan(chans, gains, polarities, chanCount)
  '    VBwbkSetTrigAnalog(chan, gain, polarity, rising, level, opstr)
  '    VBwbkArm()
  '    VBwbkSoftTrig()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
        retCount)
  '    VBwbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount, retCount)
  '    VBwbkBufferRotate(buf, scanCount, chanCount, retCount)
  '    VBwbkGetBackStat(active, retCount)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
```

```
Const CHANS% = 4
Const PRE_SCANS& = 5
Const POST_SCANS& = 9
Const PRE_FREQ# = 100#
Const POST_FREQ# = 200#
Const BLOCK% = (PRE_SCANS& + POST_SCANS&)
ReDim buf%(BLOCK% * CHANS%)
Dim i%, j%
Dim active%
Dim retCount&
ReDim chan%(CHANS%)
ReDim gains%(CHANS%), polarities%(CHANS%)
Dim ret%
Cls
Print "ADC6"
Print
' Scan definition
chan%(0) = 1        ' channel numbers
chan%(1) = 3
chan%(2) = 5
chan%(3) = 7
For i% = 0 To CHANS% - 1
   gains%(i%) = WgcX1%  ' unity gain
   polarities%(i%) = 1  ' bipolar
Next i%
' Set error handler and initialize WaveBook
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC6
ret% = VBwbkInit%(LPT1%, IRQ7%)
' Enable data packing
ret% = VBwbkSetDataPacking%(1)
' Set the acquisition for pre/post-trigger mode and the scan counts
ret% = VBwbkSetAcq%(WamPrePost%, PRE_SCANS&, POST_SCANS&)
' Set the scan configuration
ret% = VBwbkSetScan%(chan%(), gains%(), polarities%(), CHANS%)
' Set the pre-trigger and post-trigger scan rates
ret% = VBwbkSetFreq%(PRE_FREQ#, POST_FREQ#)
' Set the trigger source to an analog trigger on channel 1 at 2 volts
ret% = VBwbkSetTrigAnalog%(1, WgcX1%, 1, WctRisingEdge%, 2!, .1)
' Arm the acquisition
ret% = VBwbkArm%()
' Start reading data in the background mode with cycle mode on
' and updateSingle off
ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 1, 0, 0, active%, retCount&)
' Monitor the progress of the background transfer
Print "Waiting for trigger."
While retCount& = 0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Triggered.  Transfer in progress."
While active%  0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Acquisition complete:"; retCount&; "scans acquired."
Print
' Unpack the packed data using the same buffer
ret% = VBwbkBufferUnpack%(buf%(0), buf%(0), BLOCK%, CHANS%, retCount)
' Rotate the unpacked data so that the earliest data starts at the
' beginning of the buffer and the latest is at the end
ret% = VBwbkBufferRotate%(buf%(0), BLOCK%, CHANS%, retCount)
' Print results
Print "Pre-trigger data acquired:"
For i% = 0 To CHANS% - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = 0 To PRE_SCANS& - 1
      Print Tab(j% * 7 + 17); buf%(j% * CHANS% + i%);
   Next j%
   Print
Next i%
Print
Print "Post-trigger data acquired:"
For i% = 0 To CHANS% - 1
   Print "Channel"; i% + 1; "Data:";
   For j% = PRE_SCANS& To BLOCK% - 1
      Print Tab((j% - PRE_SCANS&) * 7 + 17); buf%(j% * CHANS% + i%);
```

```
      Next j%
      Print
   Next i%
   ' Close and Exit
   ret% = VBwbkClose%()
Exit Sub
ErrorHandlerADC6:
   Dim ErrorString$
   ErrorString$ = "ERROR in ADC6"
   ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
   If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
   MsgBox ErrorString$, , "Error!"
   End
End Sub
```

Sub ADC7_Click ()

```
Sub ADC7_Click ()
   ' 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:
   '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
   '    VBwbkSetMux(startChan, endChan, gain, polarity)
   '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
   '    VBwbkSetTrigHardware(source, level)
   '    VBwbkArm()
   '    VBwbkSoftTrig()
   '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
        retCount)
   '    VBwbkGetBackStat(active, retCount)
   '    VBwbkSetErrHandler(errNum)
   '    VBwbkInit(lptPort, lptIntr)
   '    VBwbkClose()
   Const CHANS% = 8
   Const SCANS& = 20000
   Const BLOCK% = 1000
   Const FREQ# = 5000#
   ReDim buf0%(CHANS% * BLOCK%)
   ReDim buf1%(CHANS% * BLOCK%)
   Dim i%, j%
   Dim active%
   Dim retCount&
   Dim tmpActive%
   Dim tmpRetCount&
   Dim ret%
   ReDim totals&(CHANS%)
   Dim whichBuf%
   Cls
   Print "ADC7"
   Print
   ' Set error handler and initialize WaveBook
   ret% = VBwbkSetErrHandler%(100)
   On Error GoTo ErrorHandlerADC7
   ret% = VBwbkInit%(LPT1%, IRQ7%)
   ' Set the acquisition to NShot on trigger and the post-trigger scan count
   ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
   ' Set the scan configuration
   ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
   ' Set the post-trigger scan rates
   ret% = VBwbkSetFreq%(1#, FREQ#)
   ' Set the trigger source to a software trigger command
   ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
   ' Arm the acquisition
   ret% = VBwbkArm%()
   ' Issue a software trigger command to the hardware
   ret% = VBwbkSoftTrig%()
   ' Start reading data into the first buffer
   ret% = VBwbkBufferTransfer%(buf0%(0), BLOCK%, 0, 0, 0, tmpActive%, tmpRetCount&)
   whichBuf% = 0
   Do
      ' Swap the buffer selector, whichBuf selects the transfer 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% = VBwbkGetBackStat(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% = VBwbkBufferTransfer(buf0%(0), BLOCK%, 0, 0, 0, tmpActive%,
                     tmpRetCount&)
           Else
              ret% = VBwbkBufferTransfer(buf1%(0), BLOCK%, 0, 0, 0, tmpActive%,
                      tmpRetCount&)
           End If
        End If
        ' Process the data into the process buffer
        If (retCount&  0) Then
           ' Average the readings in the process buffer and print the results
           For j% = 0 To CHANS% - 1
              totals&(j%) = 0
           Next j%
           For i% = 0 To retCount& - 1
              For j% = 0 To CHANS% - 1
                 If whichBuf% = 0 Then
                    totals&(j%) = totals&(j%) + buf1%(i% * CHANS% + j%)
                 Else
                    totals&(j%) = totals&(j%) + buf0%(i% * CHANS% + j%)
                 End If
              Next j%
           Next i%
           Print "Averages:";
           For j% = 0 To CHANS% - 1
              Print Tab(j% * 7 + 17); Format$((5# / 32768#) * totals&(j%) / retCount&,
"#0.000");
           Next j%
           Print
        End If
  Loop While (active%  0)

  'Close and exit
  ret% = VBwbkClose()
Exit Sub
ErrorHandlerADC7:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC7"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)
  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Sub ADC8_Click ()

```
Sub ADC8_Click ()
  ' This example reads multiple scans from multiple channels and writes the
  ' data directly a disk file in a packed format.
  ' Function used:
  '    VBwbkSetAcq(mode, preTrigCount, postTrigCount)
  '    VBwbkSetFreq(preTrigFreq, postTrigFreq)
  '    VBwbkSetMux(chans, gains, polarities, chanCount)
  '    VBwbkSetTrigHardware(source, level)
  '    VBwbkArm()
  '    VBwbkSoftTrig()
  '    VBwbkBufferTransfer(buf, scanCount, cycle, updateSingle, foreground, active,
       retCount)
  '    VBwbkGetBackStat(active, retCount)
  '    VBwbkBufferUnpack(packedBuf, unpackedBuf, scanCount, chanCount, retCount)
  '    VBwbkSetErrHandler(errNum)
  '    VBwbkInit(lptPort, lptIntr)
  '    VBwbkClose()
  Const CHANS% = 2
  Const SCANS& = 100000
  Const FREQ# = 10000#
```

```
Const BLOCK% = 2000     ' CHANS% * BLOCK% must be a multiple of 4
ReDim buf%(CHANS% * BLOCK%)
Dim i%, j%
Dim active%
Dim retCount&
Dim ret%
Dim fileHandle%
Dim byteCount%, wordCount%, sampleCount%, scanCount%
Dim termChar$
Dim voltage!
Cls
Print "ADC8"
Print
' Set error handler and initialize WaveBook
ret% = VBwbkSetErrHandler%(100)
On Error GoTo ErrorHandlerADC8
ret% = VBwbkInit%(LPT1%, IRQ7%)
' Enable data packing
ret% = VBwbkSetDataPacking%(1)
' Set the acquisition to NShot on trigger and the post-trigger scan count
ret% = VBwbkSetAcq%(WamNShot%, 0, SCANS&)
' Set the scan configuration
ret% = VBwbkSetMux%(1, CHANS%, WgcX1%, 1)
' Set the post-trigger scan rate
ret% = VBwbkSetFreq%(1#, FREQ#)
' Set the trigger source to a software trigger command
ret% = VBwbkSetTrigHardware%(WtsSoftware%, 0!)
' Arm the acquisition
ret% = VBwbkArm%()
' Set the direct-to-disk filename with no pre-write
ret% = VBwbkSetDiskFile%("adcex8.bin", WdfWriteFile%, 0)
' Start reading data in the background mode with cycle mode on
' and updateSingle off
ret% = VBwbkBufferTransfer%(buf%(0), BLOCK%, 1, 0, 0, active%, retCount&)
' Issue a software trigger command to the hardware
ret% = VBwbkSoftTrig%()
' Monitor the progress of the background transfer
Print "Waiting for trigger."
While retCount& = 0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Triggered.  Transfer in progress."
While active%  0
   ret% = VBwbkGetBackStat%(active%, retCount&)
Wend
Print "Acquisition complete:"; retCount&; "scans acquired."
Print
' Close and Exit
ret% = VBwbkClose%()
' Convert the binary file just read to a text file
Print "Converting adcex8.bin to adcex8.txt"
' Open the binary input file
Open "adcex8.bin" For Input As 1
fileHandle% = FileAttr(1, 2)
' Open the text output file
Open "adcex8.txt" For Output As 2
Do
   ' Convert BLOCK unpacked scans to packed bytes
   scanCount% = BLOCK%
   sampleCount% = scanCount% * CHANS%
   wordCount% = sampleCount% * 3 / 4
   byteCount% = 2 * wordCount%
   ' Read the packed bytes from the input file and get the number
   ' of bytes actually read
   byteCount% = fRead%(fileHandle%, buf%(0), byteCount%)
   ' Convert the number of bytes read from packed bytes to unpacked scans
   wordCount% = byteCount% / 2
   sampleCount% = wordCount% * 4 / 3
   scanCount% = sampleCount% / CHANS%
   ' Unpack the packed data using the same buffer.  This command
   ' can be called even if the WaveBook if not online or connected.
   ret% = VBwbkBufferUnpack%(buf%(0), buf%(0), BLOCK%, CHANS%, scanCount%)
   ' Write the scans read and unpacked to the text file
   For i% = 0 To scanCount% - 1
      For j% = 0 To CHANS% - 1
```

```
           ' Send a tab between channels and a newline after each scan
           If (j%  CHANS% - 1) Then
              termChar$ = Chr$(9)
           Else
              termChar$ = Chr$(13) + Chr$(10)
           End If
           ' calculate and write out the voltage value
           voltage! = buf%(i% * CHANS% + j%) * 5! / 32768!
           Print #2, Format$(voltage!, ".000") + termChar$;
        Next j%
     Next i%
     ' Print something so that the program doesn't appear 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."
Exit Sub
ErrorHandlerADC8:
  Dim ErrorString$
  ErrorString$ = "ERROR in ADC8"
  ErrorString$ = ErrorString$ & Chr(10) & "BASIC Error :" + Str$(Err)
  If Err = 100 Then ErrorString$ = ErrorString$ & Chr(10) & "WaveBook Error : " +
Hex$(wbkErrno%)

  MsgBox ErrorString$, , "Error!"
  End
End Sub
```

Notes

Command Reference (Standard API) 10

Overview

The first part of this chapter describes the WaveBook/512 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 24), several reference tables define parameters for: Parallel Port Protocols, LPT channels, IRQ Settings, Trigger Sources/Types, Gain Codes, the API Error Codes, etc.

The WaveBook software commands are described on the following pages.

Function	Description	Page
Initialization and LPT Port Control		
`wbkInit`	Starts the session with the WaveBook by putting it on-line and initializing its internal hardware.	10-12
`wbkSelectPort`	Selects an initialized WaveBook/512.	10-14
`wbkClose`	Ends the session with the currently selected WaveBook and takes it off line.	10-5
`wbkSetProtocol`	Sets the printer port communication protocol.	10-19
`wbkSetDefaultProtocol`	Forces the use of a specific printer port protocol.	10-16
`wbkGetProtocol`	Retrieves the current printer port protocol.	10-11
`wbkGetDriverVersion`	Gets the version of the WaveBook software driver.	10-9
Hardware Interrogation		
`wbkGetChannelType`	Retrieves the type of chassis or option card attached to the given channel.	10-8
`wbkGetChannelCount`	Retrieves the number of channels supplied by the chassis that includes the given channel.	10-7
`wbkGetChannelInfo`	Retrieves various information stored in non-volatile memory of a chassis.	10-8
`wbkGetGainVal`	Retrieves the gain setting of the selected channel.	10-9
Calibration		
`wbkSetCalInput`	Sets the WaveBook to sample either the front-panel analog inputs, or the 0, 5, or 0.5 volt calibration sources.	10-15
`wbkSetCalTable`	Selects between the factory or user calibration tables.	10-15
`wbkGetCalConstants`	Retrieves factory or user calibration constants.	10-7
`wbkSetUserCalConstants`	Sets the user calibration table entry for the specified channel, gain, and range.	10-22
`wbkWriteUserCalConstants`	Writes the user calibrations entries for the chassis containing the specified channel into the non-volatile memory.	10-23
One-Step Acquisition		
`wbkRd`	Takes one sample of a channel.	10-12
`wbkRdScan`	Takes one sample of a range of channels.	10-13
`wbkRdN`	Takes a number of samples of a channel.	10-12
`wbkRdScanN`	Takes a number of samples of a range of channels.	10-13
Custom Acquisition Scan Sequence		
`wbkSetScan`	Sets the scan sequence, with arbitrary channel number, gain, and range settings.	10-19
`wbkSetMux`	Sets the scan sequence to all the channels in a range, all with the same gain and range settings.	10-18
`wbkGetScan`	Retrieves the current scan sequence: the number of channels and their channel numbers, gains, and ranges.	10-11
Custom Acquisition Trigger		
`wbkSetTrigHardware`	Sets the trigger source to one of the high-speed trigger sources.	10-22
`wbkSetTrigAnalog`	Sets the trigger source to any one of the analog input channels in the system.	10-20
`wbkSetTrigComplex`	Sets the trigger source to a combination of analog input channels.	10-21
`wbkSoftTrig`	Issues a trigger, regardless of the selected trigger source.	10-22
Custom Acquisition Scan Count and Rate		
`wbkSetFreq`	Sets the scan frequency (in Hertz).	10-17

`wbkSetPeriod`	Sets the scan period (in nanoseconds).	10-18
`wbkSetAcq`	Sets the acquisition mode and the number of pre-and post-trigger scans.	10-14
`wbkGetMaxFreq`	Retrieves the maximum valid scan frequency (in Hertz).	10-10
`wbkGetMinPeriod`	Retrieves the minimum valid scan period (in nanoseconds).	10-10
`wbkGetFreq`	Retrieves the current scan frequency (in Hertz).	10-9
`wbkGetPeriod`	Retrieves the current scan period (in nanoseconds).	10-11
Custom Acquisition Acquire Samples and Analyze		
`wbkArm`	Enable an acquisition.	10-2
`wbkDisarm`	Disable the current acquisition.	10-6
`wbkBufferTransfer`	Transfer acquired samples into a user-supplied buffer.	10-4
`wbkGetBackStat`	Retrieves the state of a background transfer.	10-7
`wbkStopBack`	Stops the current background data transfer.	10-23
`wbkBufferUnpack`	Unpacks packed samples.	10-5
`wbkBufferRotate`	Rearranges a circular buffer of scans into chronological order.	10-3
`wbkSetAdcProcess`	Sets the handler for on-the-fly sample processing.	10-15
`wbkSetDiskFile`	Requests that an acquisition be copied to disk.	10-17
`wbkSetDataPacking`	Enables/disables packing of acquired samples	10-16
`wbkSetTimeout`	Sets the maximum amount of time a foreground transfer should wait.	10-20
Custom Acquisition Hardware Configuration		
`wbkSetChanOption`	Sets the configuration of channel parameters.	10-16
`wbkSetModuleOption`	Sets the configuration of parameters for the entire module.	10-18
`wbkGetChanOption`	Retrieves the channel configuration parameters.	10-16
`wbkGetModuleOption`	Retrieves the module configuration parameters.	10-10
Automatic Error Processing		
`wbkSetErrorHandler`	Sets the handler that will be executed upon an error condition.	10-17
`wbkDefaultHandler`	Displays an error message and exits the application.	10-5
Digital I/O		
`wbkDigWrite`	Writes a byte to the digital I/O port.	10-6
`wbkDigRead`	Reads a byte from the digital I/O port.	10-6

Commands in Alphabetical Order

The following pages give the details for each WaveBook/512 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).

wbkArm

Prototype	`wbkArm(void);`
Parameters	None
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkDisarm`
Program References	ADCEX2, ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

wbkArm arms an acquisition using the current acquisition configuration. **wbkArm** downloads all the acquisition parameters to the hardware and arms the trigger. All of the acquisition parameters must be set before **wbkArm** is invoked. If an acquisition has already been performed, then the acquisition parameters are maintained, and **wbkArm** will re-use those settings. Otherwise, each of the following parameters must be set:

Scan Sequence	**wbkSetScan or wbkSetMux**
Trigger Source	**wbkSetTrigHardware or wbkSetTrigAnalog or wbkSetTrigComplex**
Scan Rate	**wbkSetFreq or wbkSetPeriod**
Acquisition Type / Scan Count	**wbkSetAcq**

wbkBufferRotate

Prototype	`int wbkBufferRotate(int *buffer, ulong scanCount, uint chanCount, ulong acqCount)`
Parameters	
`int _huge *buffer`	The buffer, containing up to **`scanCount`** scans of **`chanCount`** samples per scan, that is to be arranged.
`ulong scanCount`	The number of scans the buffer has room for.
`uint chanCount`	The number of samples (channels) in each scan.
`ulong acqCount`	The total number of scans acquired into the buffer. Retrieved with **`wbkGetBackStat`**. May be much larger than **`scanCount`**. If **`acqCount`** is not greater than **`scanCount`**, then **`wbkRotateBuffer`** does nothing.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkBufferTransfer, wbkGetBackStat, wbkBufferUnpack`
Program References	ADCEX6

`wbkBufferRotate` rotates a circular buffer of scans into chronological order. When scans are acquired using **`wbkBufferTransfer`**, with the cycle parameter non-zero, 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 anyplace 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	...	5	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. wbkBufferRotate 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 **`wbkBufferRotate`** does not modify the buffer.

`wbkBufferRotate` only works on unpacked samples. If the scans were packed when they were transferred, then they must be unpacked with **`wbkUnpackBuffer`** before they are re-arranged.

wbkBufferTransfer

Prototype	`wbkBufferTransfer(int_huge *buf, ulong scanCount, uchar cycle, uchar updateSingle, uchar foreground, uchar *active, ulong *retCount)`
Parameters	
`int _huge *buf`	Pointer to buffer holding the words of data collected from the WaveBook. Can be as large as memory allows.
`ulong scanCount`	The number of scans the buffer has room for. For packed data, the scan count times the scan length must be a multiple of 4. Also if packed data is used and will not be unpacked in the same buffer, the buffer size may be reduced by 25%.
`uchar cycle`	Specifies what happens if the amount of data being acquired exceeds the size of the buffer. If the cycle flag=0 for no cycle (false), the data transfer stops at the end of the buffer. The program should execute **wbkBufferTransfer** again if the acquisition is still active (see active) to store the next set of data. If cycle mode is non-zero for cycle (true) and the buffer is full, the transfer continues transferring data starting again overwriting the beginning of the buffer.
`uchar updateSingle`	Controls whether data is transferred whenever data is available or in 2K blocks. If **updatesingle**=0 (false), data will be transferred in blocks of 2K words, which improves efficiency. This may be useful for high speed transfers. If **updatesingle** is non-zero (true), data will be transferred whenever data is available one word at a time. These single word transfers may be useful for slower transfers to get immediate recognition of data.
`uchar foreground`	If the foreground flag=0, the buffer transfer command returns to the application program as soon as practical and the status of the transfer can be monitored using the **wbkGetBackStat** function. If non-zero, the buffer transfer command does not return to the application program until the transfer is complete. If cycle mode is true and foreground is true, the buffer transfer command will not return to the application program until the acquisition is complete. If cycle is false and foreground is true, the acquisition complete or buffer full will cause a return to the program.
`uchar *active`	A value returned by the buffer transfer command upon completion. If **active**=0, the acquisition is finished and all data has been transferred. If non-zero, the acquisition is still active when the buffer transfer command has completed. This does not necessarily mean that more data will be transferred.
`ulong *retCount`	The number of scans that have been transferred into the buffer prior to the function returning.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	**wbkGetBackStat, wbkStopBack, wbkSetTimeout**
Program References	ADCEX2, ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

wbkBufferTransfer transfers data that is being acquired from the WaveBook into memory. For background data transfers, **wbkBufferTransfer** may be issued before the arm command so that the buffer is ready to hold data when the acquisition starts. Typically, **wbkBufferTransfer** is performed immediately after the **wbkArm** command.

The cycle flag in the **wbkBufferTransfer** command controls the overwriting of the acquisition buffer. If the cycle flag is true, the background data collection will wrap around in the acquisition buffer, continually overwriting the oldest data with new data. It is the responsibility of the application program to process or to transfer the scans to another location, typically a larger buffer or a disk file, before the data is overwritten. The application should maintain a variable which holds the number of scans that have been transferred from the acquisition buffer. The difference between the scans transferred and the total number of scans collected, which is returned by the **wbkGetBackStat** command, is the number of new scans available for transfer.

A hardware first in/first out (FIFO) buffer in the WaveBook collects data momentarily so that blocks of data can be transferred to the PC all at once. The return error code **WerrFIFOFull** means that the FIFO buffer in the WaveBook overflowed before the driver was able to empty it. This usually indicates that the PC or the parallel port is too slow for the acquisition rate. Under Windows, if the operating system is busy servicing another high priority task, the interrupt latency may cause a buffer overrun during fast sampling rates. The error code **WerrOverrun** can occur when in the cycle mode if the acquisition rate is very close to the maximum transfer rate of the computer's printer port. This error occurs when the computer can read data fast enough so that a **WerrFIFOFull** error doesn't occur, but so fast that the foreground task has no time to execute, potentially locking the system. Packed data transfers can reduce the occurrence of these errors.

wbkBufferUnpack

Prototype	`int wbkBufferUnpack ( int *packed, int *unpacked, ulong scanCount, uint chanCount, ulong acqCount);`
Parameters	
`int *packed`	The array of packed readings.
`int *unpacked`	The array of unpacked readings. The packed data can be unpacked in place by setting unpacked the same as packed.
`ulong scanCount`	The number of scans the buffer has room for.
`uint chanCount`	The number of samples (channels) in each scan.
`ulong acqCount`	The total number of scans in the buffer. If `acqCount` is less than `scanCount`, then only `acqCount` scans will be unpacked. The `retCount` parameter of the `wbkBufferTransfer` or `wbkGetBackStat` function can be passed as the `acqCount` parameter once the transfer started by `wbkBufferTransfer` is completed.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetDataPacking, wbkBufferTransfer, wbkBufferRotate, wbkGetBackStat`
Program References	ADCEX6, ADCEX8

wbkBufferUnpack unpacks packed samples. To maximize the sample acquisition rate through the parallel port, the WaveBook/512 can pack 4 12-bit samples into 3 16-bit words before they are transferred to the PC. **wbkSetDataPacking** controls whether or not packing is enabled. If packing is enabled, then once the data has been transferred with **wbkBufferTransfer**, it should be unpacked with **wbkBufferUnpack** before any further processing can be performed.

If the buffer is large enough to hold the unpacked samples, then they may be unpacked in place with the unpacked samples overwriting the packed samples. This reduces the memory requirements by eliminating the need for a separate buffer.

wbkClose

Prototype	`wbkClose(void);`
Parameters	None
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkInit, wbkSelectPort`
Program References	ADCEX1, ADCEX2, ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

wbkClose takes the currently selected WaveBook off line. Should be used at the end of an application to assure that the WaveBook is properly reset. If more than one WaveBook is in use then each should be individually selected with **wbkSelectPort** and then closed.

wbkDefaultHandler

Prototype	`void wbkDefaultHandler(int wbkErrnum);`
Parameters	
`int wbkErrnum`	The error code of the detected error.
Returns	Nothing
See Also	`wbkSetErrHandler`
Program References	None

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

wbkDigRead

Prototype	`int wbkDigRead(uchar address, uchar *byteVal);`
Parameters	
`uchar address`	The address, between 0 and 31, that is presented on the 5 digital I/O port address outputs during the read.
`uchar *byteVal`	The read data.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkDigWrite, wbkSetScan, wbkSetMux`
Program References	None

wbkDigRead reads a byte from the digital I/O port data lines after setting the digital I/O port address lines as specified. **wbkDigRead** may not be used during analog acquisition. It may only be used when the acquisition is inactive. To read the digital I/O port during an acquisition, set the first channel of the scan to channel 0 so that the digital inputs are read as the first sample of the scan.

wbkDigRead sets the 5 address outputs on the digital I/O connector, thus addressing up to 32 8-bit input sources.

wbkDigWrite

Prototype	`int wbkDigWrite(uchar address, uchar byteVal);`
Parameters	
`uchar address`	The address, between 0 and 31, that is presented on the 5 digital I/O port address outputs during the write.
`uchar byteVal`	The data to write.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkDigRead`
Program References	None

wbkDigWrite writes a byte to the digital I/O port data lines after setting the digital I/O port address lines as specified. **wbkDigWrite** may not be used during analog acquisition. It may only be used when the acquisition is inactive. **wbkDigWrite** sets the 5 address outputs on the digital I/O connector, thus addressing up to 32 8-bit output destinations.

wbkDisarm

Prototype	`wbkDisarm(void);`
Parameters	None
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkArm`
Program References	None

wbkDisarm disarms the acquisition, stops any ongoing sample acquisition, prevents any further triggers from being recognized, and finishes transferring acquired samples. This command forces the current acquisition, if any, to stop. A new acquisition may then be started with **wbkArm**.

wbkGetBackStat

Prototype	`int wbkGetBackStat(uchar *active, ulong *count);`
Parameters	
`uchar *active`	uchar *active, 0 if the acquisition is complete and all data has been read; non zero otherwise.
`ulong *count`	The total number of scans read.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkBufferTransfer, wbkSetDataPacking`
Program References	ADCEX4, ADCEX6, ADCEX7, ADCEX8

`wbkGetBackStat` retrieves the current state (active or inactive) and the total number of scans read so far by a background buffer transfer. If the background operation is active, the acquisition is continuing; otherwise, it is complete. The total number of scans can be used to calculate the number of new scans available so that they can be processed and/or transferred out of the background acquisition buffer.

wbkGetCalConstants

Prototype	`wbkGetCalConstants (uint chan, uchar gain, uchar bipolar,uint *gainConstant, int *offsetConstant);`
Parameters	
`uint chan`	Channel to change cal constants.
`uchar gain`	Gain to change cal constants.
`uchar bipolar`	Range to change cal constants.
`uint *gainConstant`	Variable to hold the gain calibration constant (gain times 0x8000).
`int *offsetConstant`	Variable to hold the offset calibration constant (1 LSB = 0x0010).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetUserCalConstants, wbkSetCalTable`
Program References	None

`wbkGetCalConstants` gets the calibration constants from the currently selected calibration table chosen by the **`wbkSetCalTable`** command.

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

wbkGetChannelCount

Prototype	`wbkGetChannelCount(uint chan, uchar *chanCount);`
Parameters	
`uint chan`	Channel number between 1 and 72.
`uchar *chanCount`	Number of channels in the chassis.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program Reference	None

`wbkGetChannelCount` - Each chassis in the WaveBook product line contain certain number of channels. Presently, the WaveBook/512 and the WBK10 expansion chassis contain 8 channels each. As future products are released, the number of channels per chassis may vary depending on the application. The **`wbkGetChannelCount`** retrieves the number of analog channels that are physically present in the chassis.

wbkGetChannelInfo

Prototype	`wbkGetChannelInfo(uint chan, uchar option, uchar whichInfo, char*info);`
Parameters	
`uint chan`	The channel number from 1-72 for which to read the info.
`uchar option`	If option=0, the main unit channel info will be retrieved, otherwise the option card info will be retrieved.
`uchar whichInfo`	Choose which information is to be retrieved from the specified unit/option card. Can be one of the following: `WinfoLastChangedDate`, or `WinfoLastChangedTime`.
`char *info`	If whichInfo is set to `WinfoLastChangedTime`, the time of the last write to NVRAM will be returned in the format HH:MM:SS. If whichInfo is set to `WinfoLastChangedDate`, the date of the last write to the NVRAM will be returned in the format MM/DD/YYYY
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkGetChannelInfo` returns various channel information from the non-volatile memory of the main unit or option card, of the selected channel. Currently the Date or Time of the last write to the NVRAM can be retrieved as an ASCII string.

wbkGetChannelType

Prototype	`wbkGetChannelType(uint chan, uchar option, uchar *type);`
Parameters	
`uint chan`	The channel number from 1-72 for which to read the type.
`uchar option`	If option=0, the main unit channel type will be retrieved, otherwise the option card type will be retrieved.
`uchar *type`	The channel type is one of the following: `WmctNone,WoctNone, WctStd, WctSSH`. Type `WctNone` is the channel (expansion) if not physically connected. Type is `WmctWbk512` or `WmctWbk10` if the channel exists, but channel does not use an option card. Finally, type is `WoctWbk11` if the channel uses an SSH option card. If option=0, type is either `WmctWbk512, WmctWbk10`, or `WmctNone`. If option is non-zero, type is either `WoctWbk10` or `WoctNone`.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkGetChannelType` returns the channel characteristics, main unit or option card, of the selected channel. A returned value equal to **`WmctNone`** indicates that the channel or option card does not exist. The only time such a type can be returned is if the selected channel is an expansion channel or option card.

wbkGetChanOption

Prototype	`int wbkGetChanOption(unsigned int chan, unsigned char optionCard, unsigned int optionType, double *optionValue)`
Parameters	
`unsigned int chan`	The number of the channel that data is being requested for.
`unsigned char optionCard`	Whether the value is to be retrieved from an option card or a main unit.
`unsigned int optionType`	The option value to be retrieved (see table WBK Channel Option Type Definitions).
`double *optionValue`	The return value.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkGetChanOption` allows the user to retrieve channel configuration parameters. The current value is returned in the **`optionValue`** parameter when the appropriate defined constant (from the WBK Channel Option Type Definitions table) is passed to the function in the **`optionType`** parameter.

wbkGetDriverVersion

Prototype	`wbkGetDriverVersion(uint *version);`
Parameters	
`uint *version`	A decimal version number which is 100 times the actual version. For example, 110 would indicate a version of '1.10'.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX3

`wbkGetDriverVersion` retrieves the version of the WaveBook software driver.

wbkGetFreq

Prototype	`wbkGetFreq(double *preTrigFreq, double *postTrigFreq);`
Parameters	
`double *preTrigFreq`	The currently defined pre-trigger sampling frequency from 1000000.0 to 0.01 Hz.
`double *postTrigFreq`	The currently defined post-trigger sampling frequency from 1000000.0 to 0.01 Hz.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetFreq, wbkSetPeriod, wbkGetPeriod, wbkGetMaxFreq, wbkGetMinPeriod`
Program References	ADCEX2

`wbkGetFreq` reads the current setting of both the pre-trigger and post-trigger scan frequencies set by the **`wbkSetFreq`** or **`wbkSetPeriod`** command.

wbkGetGainVal

Prototype	`wbkGetGainVal(uint *chan, uchar *gain, float *gainVal);`
Parameters	
`uint *chan`	Specifies the channel number from 1 - 72 for which to read the gain setting.
`uchar *gains`	The gain code for which the gain value will be calculated. Gain can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100(SSH).
`float *gainVal`	The actual gain value. Valid values are: 1.0, 2.0, 5.0, 10.0, 20.0 (SSH), 50.0 (SSH), 100.0 (SSH).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkGetGainVal` returns the gain setting of the selected channel through the argument **`gainVal`**. The gain setting is the actual hardware gain value for the specified channel and gain code. The gain setting is useful for converting the data read from A/D count values to a usable format such as volts.

wbkGetMaxFreq

Prototype	`wbkGetMaxFreq(double *preTrigFreq, double *postTrigFreq);`
Parameters	
`double *preTrigFreq`	Maximum frequency at which pre-trigger scans may be read. `preTrigFreq` is between 1,000,000.00 and 0.01 Hz (scans per second).
`double *postTrigFreq`	Maximum frequency at which post-trigger scans may be read. `postTrigFreq` is between 1,000,000.00 and 0.01 Hz (scans per second).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkGetMinPeriod, wbkSetFreq, wbkGetFreq, wbkSetPeriod,wbkGetPeriod`
Program References	None

wbkGetMaxFreq - The time required to complete a scan depends on a number of different parameters. The **wbkGetMaxFreq** command provides an easy method for determining this time. The value returned by this function is the maximum frequency at which a scan can be completed. During the post-trigger time, this depends on the actual number of samples you are acquiring per scan and whether or not the first channel might be a sample & hold channel. If the first channel is a sample & hold channel, extra dummy acquisition time must be added. During the pre-trigger period the situation becomes even more complicated as trigger channels become involved. In the case of complex triggering, several trigger channels may be involved in determining the trigger condition. Additional time is required as each channel is examined for a trigger condition. Therefore, even though the scan composition is always the same pre-trigger and post-trigger, the additional burden of triggering can make the pre-trigger maximum frequency lower than the post-trigger maximum frequency.

wbkGetMinPeriod

Prototype	`wbkGetMinPeriod(double *preTrigPeriod, double *postTrigPeriod);`
Parameters	
`double *preTrigPeriod`	Minimum period at which pre-trigger scans can be read. `preTrigPeriod` is between 1000.0 and 100,000,000,000 nanoseconds (100 s).
`double *postTrigPeriod`	Minimum period at which post-trigger scans can be read. `postTrigPeriod` is between 1000.0 and 100,000,000,000 nanoseconds (100 s).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkGetMaxFreq, wbkGetPeriod, wbkGetFreq, wbkSetPeriod, wbkSetFreq`
Program References	None

wbkGetMinPeriod returns the minimum pre- and post-trigger periods of a scan. The values returned are the reciprocals of the **wbkGetMaxFreq** command values which are the pre- and post-trigger periods and are measured in nanoseconds.

wbkGetModuleOption

Prototype	`int wbkGetModuleOption(unsigned int chan, unsigned char optionCard, unsigned int optionType, double *optionValue)`
Parameters	
`unsigned int chan`	Any channel on the module (expansion chassis) that has the correct type of option.
`unsigned char optionCard`	Whether the value is to be retrieve from an option card or a main unit.
`unsigned int optionType`	The option value to be retrieved (see table WBK Module Option Type Definitions).
`double *optionValue`	The return value.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

wbkGetModuleOption allows the user to retrieve module configuration parameters. The current value is returned in the **optionValue** parameter when the appropriate defined constant (from the WBK Module Option Type Definitions table) is passed to the function in the **optionType** parameter.

wbkGetPeriod

Prototype	`wbkGetPeriod(double *preTrigPeriod, double *postTrigPeriod);`
Parameters	
`double *preTrigPeriod`	The pre-trigger scan period between 1000 and 10,00,000 nanoseconds
`double *postTrigPeriod`	The post-trigger scan period between 1000 and 10,00,000 nanoseconds
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX2

`wbkGetPeriod` reads the current setting of both the pre-trigger and post-trigger sample periods set by the **`wbkSetPeriod`** or **`wbkSetFreq`** command.

wbkGetProtocol

Prototype	`wbkGetProtocol(int *protocol);`
Parameters	
`int *protocol`	Set to one of the protocol codes described under wbkSetProtocol.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkGetProtocol` - Depending on your printer port, one of several protocols might be selected. This command returns the presently selected protocol.

wbkGetScan

Prototype	`wbkGetScan(uint *chans, uchar *gains, uchar *bipolar, uint *count);`
Parameters	
`uint *chans`	An array to hold up to 128 channel numbers. The first location can be 0 for the high speed digital input. Channel numbers 1 through 8 are local channels and 9 through 72 are expansion channels.
`uchar *gains`	An array to hold up to 128 gain codes. Each gain code can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar *bipolar`	An array to hold up to 128 bipolar flags. Each bipolar flag can be 0 for unipolar mode and non-zero for bipolar mode.
`uint count`	The number of elements in each of the previous arrays and can be between 1 and 128.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetScan, wbkSetMux`
Program References	None

`wbkGetScan` returns 3 arrays describing the current scan configuration. The count argument indicates how many channels are in the scan. The following table shows the configuration of the returned array.

Array Element	Channel Number	Gain Setting	Bi/Uni-Polar Setting
0	chans(0)	gains(0)	bipolar(0)
1	chans(1)	gains(1)	bipolar(1)
2	chans(2)	gains(2)	bipolar(2)
3	chans(3)	gains(3)	bipolar(3)
etc.	etc.	etc.	etc.
count-1	chans(count-1)	gains(count-1)	bipolar(count-1)

wbkInit

Prototype	`wbkInit(uchar lptPort, uchar lptIntr);`
Parameters	
`uchar lptPort`	The port number which the WaveBook is connected to (LPT1, LPT2, LPT3, LPT4)
`uchar lptIntr`	The interrupt level (usually 7 for LPT1 and 5 for LPT2).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkClose, wbkSetDefaultProtocol`
Program References	ADCEX1, ADCEX2, ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

`wbkInit` starts the session with the WaveBook by putting it on-line and initializing its internal hardware. An error code, **`WerrNotOnLine`**, is returned if a WaveBook is not found at the specified location.

wbkRd

Prototype	`wbkRd(uint chan, int _huge *sample, uchar gain,uchar bipolar);`
Parameters	
`uint chan`	A single channel number to sample. A channel number of 0 will cause the high speed digital input to be read. Channel numbers 1 through 8 are local channels and 9 through 72 are expansion channels.
`uint _huge *sample`	A pointer to the word where the sample will be stored.
`uchar gain`	The channel gain which can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	The bipolar flag, which can be 0 for unipolar mode, and non-zero for bipolar mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkRdN, wbkRdScan, wbkRdScanN`
Program References	ADCEX1

`wbkRd` is used to take a single reading from a single channel using the specified gain and offset. It takes the channel to use as the trigger, the pointer to the word to hold the result, the gain code and the range code and takes a reading based on these parameters. This function will use a software trigger to immediately trigger and acquire one sample from the specified channel.

wbkRdN

Prototype	`wbkRdN(uint chan, int _huge *buf, ulong scanCount, uchar trigger, float level, double freq, uchar gain, uchar bipolar);`
Parameters	
`uint chan`	A single channel number to sample. A channel number of 0 will cause the high speed digital input to be read. Channel numbers 1 through 8 are local channels and 9 through 72 are expansion channels.
`uint _huge *buf`	Pointer to buffer holding the words of data collected from the WaveBook. Can be as large as memory allows.
`ulong scanCount`	Specifies the number of scans to read after a trigger occurs.
`uchar trigger`	The trigger source. Can be one of the following types: **`WtsSoftware, WtsTTLRise, WtsTTLFall, WtsAnalogRise`** and **`WtsAnalogFall`**.
`float level`	The trigger level if an analog trigger is specified.
`double freq`	The post-trigger sampling frequency from 1000000.0 to 0.01 Hertz.
`uchar gain`	The channel gain which can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	The bipolar flag, which can be 0 for unipolar mode, and non-zero for bipolar mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkRd, wbkRdScan, wbkRdScanN`
Program References	ADCEX1

`wbkRdN` is used to take multiple scans from a single channel. This function takes a channel, a pointer to an array to hold scan readings, a scan count (sample count), trigger type, trigger level, the sampling frequency in Hz, gain, and bipolar settings and uses these parameters to take a scan. If the trigger is set to software, then **`wbkRdN`** will trigger immediately. These are foreground commands (they do not return to the application program until the specified acquisition is complete).

wbkRdScan

Prototype	`wbkRdScan(uint startChan,uint endChan, int _huge *buf, uchar gain, uchar bipolar);`
Parameters	
`uint startChan`	The starting channel of the scan sequence range. **startChan** can be 0 for the high speed digital input, 1 through 8 for local channels and 9 through 72 for expansion channels.
`uint endChan`	The ending channel of the scan sequence range. **endChan** cannot be less than **startChan**. 1 through 8 are local channels and 9 through 72 are expansion channels.
`int _huge *buf`	Pointer to buffer holding the words of data collected from the WaveBook. Can be as large as memory allows.
`uchar gain`	The channel gain which can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	The bipolar flag, which can be 0 for unipolar mode, and non-zero for bipolar mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	**wbkRd, wbkRdN, wbkRdScanN**
Program References	ADCEX1

wbkRdScan reads a single sample from multiple channels using the specified gain and offset for all 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**.

wbkRdScanN

Prototype	`wbkRdScanN(uint startChan, uint endChan, int _huge *buf, ulong scanCount, uchar trigger, float level, double freq, uchar gain, uchar bipolar );`
Parameters	
`uint startChan`	The starting channel of the scan sequence range. **startChan** can be 0 for the high speed digital input, 1 through 8 for local channels and 9 through 72 for expansion channels.
`uint endChan`	The ending channel of the scan sequence range. **endChan** cannot be less than **startChan**. 1 through 8 are local channels and 9 through 72 are expansion channels.
`int _huge *buf`	Pointer to buffer holding the words of data collected from the WaveBook. Can be as large as memory allows.
`ulong scanCount`	Specifies the number of scans to read after a trigger occurs.
`uchar trigger`	The trigger source. Can be one of the following types: **WtsSoftware, WtsTTLRise, WtsTTLFall, WtsAnalogRise** and **WtsAnalogFall**.
`float level`	The trigger level (in volts) if an analog trigger is specified.
`double freq`	The sampling frequency from 1000000.0 to 0.01 Hertz.
`uchar gain`	The channel gain which can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	The bipolar flag, which can be 0 for unipolar mode, and non-zero for bipolar mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	**wbkRd, wbkRdN, wbkRdScan**
Program References	ADCEX1

wbkRdScanN reads multiple scans from multiple channels using the specified gain and offset for all channels. This function will arm the trigger and acquire scans consisting of each channel starting with **startChan** and ending with **endChan**.

wbkSelectPort

Prototype	`int wbkSelectPort(uchar lptPort);`
Parameters	
`uchar lptPort`	The port number to which the WaveBook is connected (one of LPT1, LPT2, LPT3, LPT4).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

wbkSelectPort selects an initialized WaveBook/512. This function causes any subsequent function calls to be performed on this WaveBook. Because **wbkInit** initializes then selects a WaveBook/512, **wbkSelectPort** is only needed when using multiple WaveBooks.

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

wbkSetAcq

Prototype	`wbkSetAcq(uchar mode, ulong preTrigCount,ulong postTrigCount);`
Parameters	
`uchar mode`	Sets the acquisition mode. Valid modes are: **WamNShot, WamNShotRearm, WamPrePost, WamInfinitePost**.
`ulong preTrigCount`	Specifies the minimum number of scans to read before arming the trigger when in PrePost mode. **preTrigCount** is ignored in all other modes.
`ulong postTrigCount`	Specifies the number of scans to read after a trigger occurs. **postTrigCount** is ignored in the **InfinitePost** mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

wbkSetAcq sets the acquisition mode and the number of pre-trigger and post-trigger scans to collect. There are four acquisition modes:

`NShot mode`	This mode sets up the system to collect only post-trigger scans, the argument **preTrigCount** is ignored.
`NShotRearm mode`	Like **NShot** mode, this mode sets up the system to collect only post-trigger scans, but then automatically rearms the system for another, identical acquisition.
`NShotPrePost mode`	This mode sets up the system to collect both pre-trigger and post-trigger scans. The pre-trigger scan collection is initiated by this command. Pre-trigger data collection continues until the trigger is satisfied, after which the post-trigger scans are collected.
`InfinitePost mode`	This mode sets up the system to collect an infinite amount of post-trigger scans. In this mode, data collection will continue until the **wbkDisarm** command is called.

wbkSetAdcProcess

Prototype	`wbkSetAdcProcess(wbkAdcProcessFPT wbkAdcProcess);`
Parameters	
`wbkSetAdcProcess`	Pointer to the data handler function or NULL to disable.
Prototype for Data Handler	`wbkAdcProcessFT(int _huge *buf, ulong wordCount) ;`
Parameters	
`int _huge *buf`	Pointer to data storage buffer.
`ulong wordCount`	Total number of words transferred
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkSetAdcProcess` provides access to the interrupt-time data transfer. **`wbkSetAdcProcess`** sets up a data handler for the WaveBook. Any time data is transferred by the WaveBook, it will call this handler specifying to the handler the address into which the data was transferred and the number of words that were transferred. Packing and scan boundaries are ignored. You must pass to the setProcess command a pointer to a function (in C) that is designed to process the data. This function requires two parameters, a buffer to the data that is being passed and the count of words that were transferred to the buffer. Use of this function requires caution as it operates at interrupt time.

wbkSetCalInput

Prototype	`int wbkSetCalInput(uchar calInput);`
Parameters	
`uchar Input`	Valid values: `WciNormal, WciCalGnd, WciCalGain`
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkSetCalInput` sets the WaveBook to sample either the normal analog inputs (BNCs), a calibration ground input or a calibration gain input (5.0 V or 0.5 V).

wbkSetCalTable

Prototype	`wbkSetCalTable(uchar user);`
Parameters	
`uchar user`	Set to non-zero to choose the user calibration table or 0 to choose the factory calibration table.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetUserCalConstants, wbkWriteUserCalConstants, wbkGetCalConstants`
Program References	None

`wbkSetCalTable` - To achieve rated accuracy, the WaveBook uses a digital signal processor (DSP) to transparently compensate each channel while it is being read. For plug-and-play purposes, each WaveBook component, including option and expansion hardware, contain factory-installed compensation tables in an on-board, non-volatile memory device. After the WaveBook and its options are integrated into a system, all of the individual calibration tables are merged by the DSP into one system calibration table for real-time channel compensation.

If the user desires, the WaveBook can be calibrated in the field as a system using the included WaveCal application. WaveCal ultimately creates a system calibration table which could be used in place of the factory-installed tables.

This command allows you to choose between the factory calibration and the user calibration. If the user calibration is selected, WaveCal must be executed first.

wbkSetChanOption

Prototype	**`int wbkSetChanOption(unsigned int chan, unsigned char optionCard, unsigned int optionType, double optionValue)`**
Parameters	
`unsigned int chan`	The number of the channel to be configured.
`unsigned char optionCard`	Whether the option is to be applied to an option card or a main unit.
`unsigned int optionType`	The configurable option to be set (enum). (see table WBK Channel Option Type Definitions)
`double optionValue`	The value to set the option to. Enum or value depending on option type (see table WBK Channel Option Value Definitions)
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter)
See Also	
Program References	None

`wbkSetChanOption` allows the user to configure channel parameters for WBK modules with software-configurable settings.

wbkSetDataPacking

Prototype	**`int wbkSetDataPacking(uchar packed);`**
Parameters	
`uchar packed`	Packed is 0 to disable compression or non-zero to enable it.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	wbkBufferUnpack
Program References	None

`wbkSetDataPacking` - If the sample rate is high and buffer overruns are occurring, data compression is recommended. To maximize the sample acquisition rate through the parallel port, the WaveBook/512 can pack 4 12-bit samples into 3 16-bit words before they are transferred to the PC. **`wbkSetDataPacking`** controls whether or not packing is enabled. If packing is enabled, then once the data has been transferred with **`wbkBufferTransfer`**, it should be unpacked with **`wbkBufferUnpack`** before any further processing can be performed.

If the buffer is large enough to hold the unpacked samples, then they may be unpacked in place with the unpacked samples overwriting the packed samples. This reduces the memory requirements by eliminating the need for a separate buffer.

wbkSetDefaultProtocol

Prototype	**`wbkSetDefaultProtocol (int protocol);`**
Parameters	
`int protocol`	Valid protocols are: **`WbkProtocolNone, WbkProtocol8, WbkProtocol4, WbkProtocolFPort, WbkProtocolSL, WbkProtocolSMC666, WbkProtocolFastEPP, WbkProtocolEPPBios`**.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	**`wbkSetProtocol, wbkGetProtocol`**
Program References	None

`wbkSetDefaultProtocol` - The **`wbkInit`** function tries to establish reliable communications with the WaveBook by using the default protocol (4-bit). The **`wbkSetDefaultProtocol`** command can be called before **`wbkInit`** to force the use of the specific printer port protocol that was found to work using the **`wbkTest`** program rather than the 4-bit protocol that would be used by **`wbkInit`**.

Fastest	1. WbkProtocolFastEPP
.	2. WbkProtocolSMC666
.	3. WbkProtocolSL
.	4. WbkProtocolFPort
.	5. WbkProtocolEPPBIOS
.	6. WbkProtocol8
Slowest	7. WbkProtocol4

wbkSetDiskFile

Prototype	`wbkSetDiskFile (const char _far *file, uchar openMode, ulong preWrite);`
Parameters	
`const char _far *file`	Specifies the name of the file.
`uchar openMode`	Specifies whether data is to be appended to the file, overwritten, or a new file created
`ulong preWrite`	The number of scans for which space on the disk is to be allocated
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX8

wbkSetDiskFile requests that an acquisition be copied to disk. Once the disk file has been set up to accept the data, as the data is transferred into the buffer, it will also be transferred to disk. The actual transfer is performed by the **wbkGetBackStat** command. If a foreground transfer is in progress, the **wbkGetBackStat** command is internally used to monitor the progress and thereby automatically transferring to disk. If a background transfer is being performed and the data is to be transferred to disk, the **wbkGetBackStat** must be called often enough to ensure that data is not lost on its way to the disk. If this transfer is enabled and data is lost as it is transferred to the disk due to the data rate being too high or **wbkGetBackStat** not being called often enough, an error will be generated and the acquisition stopped. The **wbkSetDiskFile** command applies only to the immediately subsequent acquisition, and must be invoked before the **wbkArm** command that starts the acquisition. The following acquisition will not automatically go to disk unless another **wbkSetDiskFile** command is specified.

wbkSetErrorHandler

Prototype	`wbkSetErrorHandler(wbkSetErrorHandlerFPT wbkErrorHandler);`
Parameters	
`wbkErrorHandler`	This is a function that takes an integer (error code) and returns nothing, or NULL to disable.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkDefaultHandler`
Program References	ADCEX1

***** For C and Pascal Only *****

wbkSetErrorHandler - 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 *****

wbkSetErrorHandler (ERRNUM%) - 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.

wbkSetFreq

Prototype	`wbkSetFreq(double preTrigFreq, double postTrigFreq);`
Parameters	
`double preTrigFreq`	The pre-trigger frequency from 1000000.0 to 0.01 Hz.
`double postTrigFreq`	The post-trigger frequencies from 1000000.0 to 0.01 Hz.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetPeriod, wbkGetFreq, wbkGetPeriod, wbkGetMaxFreq, wbkGetMinPeriod`
Program References	ADCEX2, ADCEX3, ADCEX4, ADCEX5, ADCEX6, ADCEX7, ADCEX8

wbkSetFreq sets both the pre- and post-trigger scan rates in Hertz.

Every acquisition is composed of one or more repetitions of a scan. A scan is a list of some or all of the input channels and their respective input ranges. Scans can vary in length from a single sample up to 128 samples.

wbkSetModuleOption

Prototype	`int wbkSetModuleOption(unsigned int chan, unsigned char optionCard, unsigned int optionType, double optionValue)`
Parameters	
`unsigned int chan`	Any channel on the module (expansion chassis) to be configured.
`unsigned char optionCard`	Whether the option is to be applied to an option card or a main unit.
`unsigned int optionType`	The configurable option to be set (enum). (see table WBK Module Option Type Definitions)
`double optionValue`	The value to set the option to. Enum or value depending on option type. (see table WBK Module Option Value Definitions)
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	None

`wbkSetModuleOption` allows the user to configure parameters that apply to the whole module (for WBK modules with software-configurable settings).

wbkSetMux

Prototype	`int wbkSetMux(uint startChan, uint endChan, uchar gain, uchar bipolar);`
Parameters	
`uint startChan`	The starting channel of the scan sequence range. StartChan can be 0 for the high speed digital input, 1 through 8 for local channels and 9 through 72 for expansion channels.
`uint endChan`	The ending channel of the scan sequence range. endChan cannot be less than startChan. 1 through 8 are local channels and 9 through 72 are expansion channels.
`uchar gain`	The global gain setting for all of the channels in the specified range. Valid gain codes are: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	An array of bipolar flags that can be 0 for unipolar mode and non-zero for bipolar mode.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX2, ADCEX4, ADCEX5, ADCEX7, ADCEX8

`wbkSetMux` - Simpler and less flexible than the **`wbkSetScan`** command, this command configures a scan sequence of channels from a specified start channel to a specified end channel, all with the same gain and polarity setting.

wbkSetPeriod

Prototype	`wbkSetPeriod(double preTrigPeriod, double postTrigPeriod);`
Parameters	
`preTrigPeriod`	The pre-trigger scan period from 1000 to 100,000,000,000 nanoseconds.
`postTrigPeriod`	The post-trigger scan period from 1000 to 100,000,000,000 nanoseconds.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetFreq, wbkGetPeriod, wbkGetFreq, wbkGetMinPeriod, wbkGetMaxFreq`
Program References	None

`wbkSetPeriod` - Like the **`wbkSetFreq`** command, this command sets the rate at which the scans are sampled. This command allows the rate to be set in terms of time between scans rather than scans per second.

wbkSetProtocol

Prototype	`wbkSetProtocol(int protocol);`
Parameters	
`int protocol`	Valid protocols are: `WbkProtocolNone, WbkProtocol8, WbkProtocol4, WbkProtocolFPort, WbkProtocolSL, WbkProtocolSMC666, WbkProtocolFastEPP, WbkProtocolEPPBios`
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkGetProtocol`
Program References	None

`wbkSetProtocol` - Depending on your printer port, one of several printer port communication protocols might be appropriately selected. This command sets the desired protocol. For maximum throughput, the highest protocol supported by your parallel port should be selected. Use the included **`WbkTest`** program to evaluate your parallel port hardware.

`WbkProtocolFastEPP`	Specifies the ISA-bus WBK20 and PCMCIA card WBK21 high-speed EPP interfaces which are capable of more than 2 MBytes/second. All of the other protocols are typically much slower than this with the EPP protocols typically achieving 600-800 kbytes/second, and the non-EPP protocols limited to 50-200 kbytes/second.
`WbkProtocolSMC666`	Specifies a Standard-Microsystem FDC37C666-based EPP interface board, such as the Quatech MMP-100, that is jumper-configurable for EPP operation.
`WbkProtocolSL`	Specifies an Intel 80386SL compatible EPP interface.
`WbkProtocolFPort`	Specifies an FarPoint EPP interface.
`WbkProtocolEPPBios`	Specifies an EPP Draft Revision 3.0 compatible software driver which is supplied by the computer manufacturer to perform EPP operation.
`WbkProtocol8`	Is a non-EPP protocol for IBM AT compatible printer ports. As this protocol is limited to less than 200 kbytes/second, its use is not recommended with the WaveBook/512.
`WbkProtocol4`	Is an even slower non-EPP protocol that is compatible with virtually any printer port, but is typically limited to 60 kbytes/second.
Note: Protocols here are arranged from fastest to slowest.	

wbkSetScan

Prototype	`int wbkSetScan(uint *chans, uchar *gains, uchar *bipolar, uint count);`
Parameters	
`uint *chans`	An array of up to 128 channels. The first location can be 0 for the high speed digital input. Channel number 1 through 8 are local channels and 9 through 72 are expansion channels.
`uchar *gains`	An array of up to 128 gain codes. Valid gain codes are: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar *bipolar`	An array of bipolar flags that can be 0 for unipolar mode and non-zero for bipolar mode.
`unit count`	The number of elements in each of the previous arrays and can be between 1 and 128.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX3, ADCEX6

`wbkSetScan` configures a scan sequence consisting of multiple channels and corresponding gain and bipolar/unipolar settings. As many as 128 entries can be made in the scan configuration. Any analog input channel, with any configuration, in any order, can be included in the scan. Channels can be entered multiple times at the same or different configurations. The WBK11 option card cannot have the same channels in the scan sequence at different gains. The high speed digital input port can also be included as the first location although its gain and bipolar/unipolar setting will be ignored.

The channel arrays are organized as follows.

Array Element	Channel Number	Gain Setting	Bi/uni-Polar Setting
0	chans(0)	gains(0)	bipolar(0)
1	chans(1)	gains(1)	bipolar(1)
2	chans(2)	gains(2)	bipolar(2)
3	chans(3)	gains(3)	bipolar(3)
etc.	etc.	etc.	etc.
count-1	chans(count-1)	gains(count-1)	bipolar(count-1)

wbkSetTimeout

Prototype	`int wbkSetTimeout(ulong timeout);`
Parameters	
`ulong timeout`	The time-out value in milliseconds, if non-zero (or disable time-out if 0). The default time-out value is 10,000 milliseconds and the maximum is about 1 day (86,400,000 milliseconds).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkBufferTransfer`
Program References	None

`wbkSetTimeout` sets the amount of time (in milliseconds) that a foreground operation will wait before aborting with a time-out error.

When using the foreground acquisition mode, data collection routines will not return until the acquisition is complete. If the trigger is never satisfied or the acquisition never completes, a potential system lock-up can occur. This command provides a means of recovering by setting a maximum time-out after which the foreground routine will be aborted. If the foreground routine fails to return before the time-out period, the acquisition command is automatically aborted.

wbkSetTrigAnalog

Prototype	`int wbkSetTrigAnalog(uint chan, uchar gain, uchar bipolar, uchar rising, float level, float hysteresis);`
Parameters	
`uint chan`	Sets the trigger channel. Channel numbers 1 through 8 are local channels and 9 through 72 are expansion channels. The high-speed digital input cannot be used as a complex trigger channel.
`uchar gain`	Sets the channel gain using a gain code. Each gain code can be one of the following: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar bipolar`	Sets the bipolar/unipolar mode. Bipolar is 0 for unipolar mode and non-zero for bipolar mode.
`uchar rising`	Specifies whether the trigger is rising or falling, or edge-sensitive or level-sensitive. Valid values are: **`WcrRisingEdge, WcrFallingEdge, WcrAboveLevel, WcrBelowLevel`**.
`float level`	Sets the trigger level in volts.
`float hysteresis`	Sets the level in positive volts above (falling) or below (rising) the trigger level that a signal must exceed before the trigger is armed.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetTrigComplex, wbkSetTrigHardware`
Program References	ADCEX6

`wbkSetTrigAnalog` sets the trigger source to any one of the analog input channels in the system. Only the hardware of channel one is designed to perform automatic level triggering. Using the built-in digital signal processor, this command allows any analog input channel to trigger the system. Due to the use of the DSP for level detection, trigger latency is greater than the hardware triggering of channel one. Since the specified channel does not need to be part of the configured scan, this command fully describes the characteristics of the desired trigger channel.

wbkSetTrigComplex

Prototype	`wbkSetTrigComplex(uint *chans, uchar *gains, uchar *bipolar, uchar *rising, float *levels, float *hysteresis, uint count, char *opstr);`
Parameters	
`uint *chans`	An array of up to 72 channels. Channel number 1 through 8 are local channels and 9 through 72 are expansion channels. The high-speed digital input cannot be used as a complex trigger channel.
`uchar *gains`	An array of up to 72 gain codes. Valid codes are: WgcX1, WgcX2, WgcX5, WgcX10, WgcX20 (SSH), WgcX50 (SSH), WgcX100 (SSH).
`uchar *bipolar`	An array of bipolar flags that can be 0 for unipolar mode and non-zero for bipolar mode.
`uchar *rising`	An array that specifies whether the trigger is rising or falling, or edge-sensitive or level-sensitive. Rising can be one of the following: **WcrRisingEdge, WcrFallingEdge, WcrAboveLevel, WcrBelowLevel, WcrAfterRisingEdge, WcrAfterFallingEdge, WcrAfterAboveLevel**, or **WcrAfterBelowLevel**.
`float *levels`	An array of trigger level voltages.
`float *hysteresis`	An array of positive voltages that sets the level above (falling) or below (rising) the trigger level that a signal must exceed before the trigger is armed.
`uint count`	The number of elements in each of the previous arrays. Valid values: 1 - 72
`char *opstr`	A character string which defines whether each individual analog trigger is ANDed or ORed together to produce a single complex trigger. If the AND operation is chosen, all individual triggers must be true for the acquisition to trigger. If the OR operation is chosen, only one channel needs to be true. The **opstr** should be "*" for the AND operation or "+" for the OR operation.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	
Program References	ADCEX5

wbkSetTrigComplex sets the trigger source to a combination of multiple analog input channels in a boolean equation of ANDs or ORs. Using arrays, a list of channels is characterized which are independent of the acquisition channel list. The arrays are arranged as follows.

Array Element	Channel Number	Gain Setting	Bipolar/ Unipolar	Rising/ Falling	Voltage Level	Hysteresis
0	chan(0)	gain(0)	bipolar(0)	edge(0)	level(0)	hyst(0)
1	chan(1)	gain(2)	bipolar(1)	edge(1)	level(1)	hyst(1)
2	chan(2)	gain(2)	bipolar(2)	edge(2)	level(2)	hyst(2)
3	chan(3)	gain(3)	bipolar(3)	edge(3)	level(3)	hyst(3)
etc.	etc.	etc.	etc.	etc.	etc.	etc.
count-1	chan (count-1)	gain (count-1)	bipolar (count-1)	edge (count-1)	level (count-1)	hyst (count-1)

Each channel in the channel array provides a TRUE or FALSE to the boolean equation that provides the system trigger signal. Between the channel terms in the equation are either ANDs or ORs, depending on the operation string parameter.

The same channel can be used in both the trigger channel list and the acquisition channel list. If the channel is from a WBK11 option card, the gain must be the same for that channel in both lists.

Chan	Edge	Level	Hyst
3	Rising	1.2V	0.02V
4	Rising	2.2V	0.02V
7	Falling	-4.0V	0.02V

The following example has an operation parameter of OR and 3 channels in the trigger channel array with the following Parameters.

The Boolean trigger equation is:

System trigger = (CH3 rising above 1.2V) OR (CH4 rising above 2.2V) OR (CH7 falling below -4.0V)

When any of these terms becomes TRUE, the system trigger becomes TRUE, triggering the system.

Refer to the *Operations Guide* (chapter 4) for more information on triggering.

wbkSetTrigHardware

Prototype	`wbkSetTrigHardware(uchar source, float level);`
Parameters	
`uchar source`	Specifies the trigger source as WtsSoftware, WtsTTLRise, WtsTTLFall, WtsAnalogRise and WtsAnalogFall.
`float level`	The analog level in volts.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetTrigAnalog, wbkSetTrigComplex`
Program References	ADCEX2, ADCEX3, ADCEX4, ADCEX7, ADCEX8

wbkSetTrigHardware selects one of 5 hardware-based trigger sources for high speed system triggering. The trigger sources are a rising or falling analog input on channel 0, a rising or falling TTL input, or a software command from the PC.

wbkSetUserCalConstants

Prototype	`wbkSetUserCalConstants (uint chan, uchar gain, uchar bipolar,uint gainConstant, int offsetConstant);`
Parameters	
`uint chan`	Channel to change cal constants.
`uchar gain`	Gain to change cal constants.
`uchar bipolar`	Range to change cal constants.
`uint gainConstant`	Gain calibration constant (gain times 0x8000).
`int offsetConstant`	Offset calibration constant (1 LSB = 0x0010).
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkWriteUserCalConstants, wbkSetCalTable`
Program References	None

wbkSetUserCalConstants sets the user accessible calibration constants.

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

wbkSoftTrig

Prototype	`wbkSoftTrig(void);`
Parameters	None
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetTrigHardware`
Program References	ADCEX2, ADCEX3, ADCEX4, ADCEX7, ADCEX8

wbkSoftTrig - When the trigger source has been set to **WtsSoftware** by the **wbkSetTrigHardware** command, the issuance of this command causes the system trigger to be satisfied. This command can also be used to force a trigger, even if the trigger source was set to something other than **WcsSoftware**.

wbkStopBack

Prototype	`wbkStopBack( void);`
Parameters	None
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkBufferTransfer`
Program References	None

`wbkStopBack` aborts an active background transfer.

wbkWriteUserCalConstants

Prototype	`wbkWriteUserCalConstants ( uint chan );`
Parameters	
`uint chan`	The channel number which is contained in the chassis for which constants are to be written.
Returns	An error number, or 0 if no error (see Error Code table at end of this chapter).
See Also	`wbkSetUserCalConstants`
Program References	None

`wbkWriteUserCalConstants` writes the working copy of the calibration constants (as set by the **`wbkSetUserCalConstants`** command) into the NVRAM. Similar to the **`wbkGetChannelCount`** command, writing to any channel in a specific chassis writes all the calibration constants for all the channels in that chassis.

API Reference Tables

These tables provide information for programming with the WaveBook/512 Application Programming Interface. The tables are organized as follows:

API Parameter Reference Tables

Table Title	Page
Parallel Port Protocols	11-24
LPT Channels	11-24
IRQ Settings	11-24
Hardware Trigger Sources	11-25
Multi-Channel Trigger Types	11-25
Acquisition Modes	11-25
Main-Chassis and Option-Card Channel Types	11-25
Calibration Input Sources	11-25
Gain Codes	11-26
Channel Information Codes	11-26
Disk File Open Modes	11-26
WBK Channel Option Type Definitions	11-26
WBK Module Option Value Definitions	11-27
API Error Codes	11-28

Parallel Port Protocols

Description	Value	Description
`WbkProtocolNone`	0	Communications not established
`WbkProtocol8`	1	Standard LPT Port 8-bit mode
`WbkProtocol4`	2	Standard LPT Port 4-bit mode
`WbkProtocolFPort`	10	Far Point F/Port EPP mode
`WbkProtocolSL`	20	Generic 82360 SL EPP mode
`WbkProtocolSM666`	30	SMC 37C666 EPP mode
`WbkProtocolEPPBIOS`	40	EPP bios mode
`WbkProtocolFastEPP`	50	Fast EPP mode

LPT Channels

Description	Value
LPT1	0
LPT2	1
LPT3	2
LPT4	3

IRQ Settings

Description	Value
IRQ2	2
IRQ3	3
IRQ4	4
IRQ5	5
IRQ6	6
IRQ7	7

Hardware Trigger Sources

Definition	Value	Trigger Source
`WtsSoftware`	0	Software (`wbkSoftTrig`)
`WtsTTLRise`	1	External TTL rising edge
`WtsTTLFall`	2	External TTL falling edge
`WtsAnalogRise`	3	Channel 1 rising above a setpoint
`WtsAnalogFall`	4	Channel 1 falling below a setpoint

Multi-Channel Trigger Types

Definition	Value	Trigger Type
`WctRisingEdge`	0	Rising-edge trigger
`WctFallingEdge`	1	Falling-edge trigger
`WctAboveLevel`	2	Above level trigger
`WctBelowLevel`	3	Below level trigger
`WctAfterRisingEdge`	4	After rising-edge trigger
`WctAfterFallingEdge`	5	After falling-edge trigger
`WctAfterAboveLevel`	6	After above-level trigger
`WctAfterBelowLevel`	7	After below-level trigger

Acquisition Modes

Definition	Value	Acquisition Mode
`WamNShot`	0	N-Shot after trigger
`WamNShotRearm`	1	N-Shot after trigger with automatic re-arm
`WamPrePost`	2	Pre-trigger / Post-trigger
`WamInfinitePost`	3	Infinite post-trigger

Main-Chassis and Option-Card Channel Types

Definition	Value	Description
`WmctNone`	0	Channel does not exist.
`WmctWbk512`	1	WaveBook/512 channel
`WmctWbk10`	2	WBK10 channel
`WmctWbk14`	3	WBK14 channel
`WmctWbk15`	4	WBK15 channel
`WoctNone`	0	Channel does not exist or no option installed in channel.
`WoctWbk11`	1	WBK11 sample and hold channel
`WoctWbk12`	2	WBK12 filter card
`WoctWbk13`	3	WBK13 filter/SSH card

Calibration Input Sources

Definition	Value	Description
`WciNormal`	0	Samples the external input
`WciCalGnd`	1	Samples the 0 volt calibration input
`WciCal5V`	2	Samples the 5 volt calibration input
`WciCal500mV`	3	Samples the 0.5 volt calibration input(requires WBK11)

Gain Codes

Description	Value
`WgcX1`	0
`WgcX2`	1
`WgcX5`	2
`WgcX10`	3
`WgcX20`	4 (requires WBK11)
`WgcX50`	5 (requires WBK11)
`WgcX100`	6 (requires WBK11)

Channel Information Codes

Definition	Value	Description
`WinfoLastChangedDate`	0	Retrieve the date that the contents of the non-volatile memory was last written
`WinfoLastChangedTime`	1	Retrieve the time that the contents of the non-volatile memory was last written

Disk File Open Modes

Definition	Value	Description
`WfdAppendFile`	0	Append data to the end of an existing file, if any
`WfdWriteFile`	1	Overwrite an existing file, if any
`WfdCreateFile`	2	Create a new file, or report an error if file exists

WBK Channel Option Type Definitions

Definition	Value	Description
Wbk12		
`WcotWbk12FilterCutOff`	0	Value: 400 Hz - 100 kHz Valid only for channels 1 and 5 Channel 1 controls channels 1 - 4 Channel 5 controls channels 5 - 8
`WcotWbk12FilterType`	1	Enumeration - Elliptic, Linear Phase
`WcotWbk12FilterMode`	2	Enumeration - Filter On, Filter Bypassed
`WcotWbk12PreFilterMode`	3	Enumeration - Default, Override
Wbk13		
`WcotWbk13FilterCutOff`	0	Value: 400 Hz - 100 kHz Valid only for channels 1 and 5 Channel 1 controls channels 1 - 4 Channel 5 controls channels 5 - 8
`WcotWbk13FilterType`	1	Enumeration - Elliptic, Linear Phase
`WcotWbk13FilterMode`	2	Enumeration - Filter On, Filter Bypassed
`WcotWbk13PreFilterMode`	3	Enumeration - Default, Override
Wbk14		
`WcotWbk14LowPassMode`	0	Enumeration - Bypass, On
`WcotWbk14LowPassCutOff`	1	Value: 30Hz - 100 kHz
`WcotWbk14HighPassCutOff`	2	Enumeration - 0.1 Hz, 10 Hz
`WcotWbk14CurrentSrc`	3	Enumeration - Off, 2 mA, 4 mA
`WcotWbk14PreFilterMode`	4	Enumeration - Default, Override

WBK Module Option Type Definitions

Wbk14 Definition	Value	Description
`WmotWbk14ExcSrcWaveform`	0	Enumeration - Sine, Random
`WmotWbk14ExcSrcFreq`	1	Value: 20 - 100,000 Hz
`WmotWbk14ExcSrcAmplitude`	2	Value: 0 - 10 Vp-p for sine waveform; 0 - 1.7 Vrms for random waveform
`WmotWbk14ExcSrcOffset`	3	Value: -5.0 to +5.0 V

WBK Channel Option Value Definitions

Definition	Values
Wbk12	
`WcotWbk12FilterType`	WcovWbk12FilterElliptic, WcovWbk12FilterLinearPhase
`WcotWbk12FilterMode`	WcovWbk12FilterBypass, WcovWbk12FilterOn
`WcotWbk12AntiAliasMode`	WcovWbk12PreFilterDefault, WcovWbk12PreFilterOff
Wbk13	
`WcotWbk13FilterType`	WcovWbk13FilterElliptic, WcovWbk13FilterLinearPhase
`WcotWbk13FilterMode`	WcovWbk13FilterBypass, WcovWbk13FilterOn
`WcotWbk13AntiAliasMode`	WcovWbk13PreFilterDefault, WcovWbk13PreFilterOff
Wbk14	
`WcotWbk14CurrentSrc`	WcovWbk14CurrentSrcOff, WcovWbk14CurrentSrc2mA, WcovWbk14CurrentSrc4mA
`WcotWbk14HighPassCutOff`	WcovWbk14HighPass0_1Hz, WcovWbk14HighPass10Hz
`WcotWbk14LowPassMode`	WcovWbk14LowPassBypass, WcovWbk14LowPassOn
`WcotWbk14AntiAliasMode`	WcovWbk14PreFilterDefault, WcovWbk14PreFilterOff

WBK Module Option Value Definitions

Wbk14 Definition	Values
`WmotWbk14ExcSrcWaveform`	WmovWbk14WaveformSine, WmovWbk14WaveformRandom

API Error Codes

Error	Decimal Code	Hex Code	Description
`WerrNoError`	0	00H	No error
`WerrBadChannel`	1	01H	Specified LPT channel was out-of-range
`WerrNotOnLine`	2	02H	Requested WaveBook is not on-line
`WerrNoWaveBook`	3	03H	WaveBook is not on the requested channel
`WerrBadAddress`	4	04H	Bad function address
`WerrFIFOFull`	5	05H	FIFO Full detected, possible data corruption
`WerrOutofMemory`	6	06H	Memory allocation error
`WerrInvFreq`	16	10H	Invalid frequency or period parameter
`WerrInvCalInput`	17	11H	Invalid calibration input
`WerrInvChan`	18	12H	Invalid channel
`WerrInvCount`	19	13H	Invalid count parameter
`WerrInvGain`	20	14H	Invalid channel gain parameter
`WerrInvOpstr`	21	15H	Invalid complex trigger operation string
`WerrArmed`	22	16H	Attempt to reconfigure an acquisition while still active
`WerrDspCommFailure`	23	17H	Communications with the WaveBook DSP failed
`WerrEepromCommFailure`	24	18H	Communications with the WaveBook EEPROM failed
`WerrInvTrigSource`	25	19H	Invalid trigger source parameter
`WerrTimeout`	26	1AH	A time-out occurred during a foreground read operation
`WerrInvMode`	27	1BH	Invalid acquisition mode parameter
`WerrInvTrigLevel`	28	1CH	Invalid trigger level parameter
`WerrTypeConflict`	29	1DH	A number >255 or <0 was passed to a function requiring an unsigned character (0-255)
`WerrMultBackXfer`	30	1EH	A second background transfer was requested
`WerrOverrun`	31	1FH	ADC data acquired too fast, but all data is valid
`WerrInvDigAddress`	32	20H	Invalid digital I/O address
`WerrInvCalConstant`	33	21H	Out-of-range user Cal Constant
`WerrInvComplexTrig`	34	22H	Invalid complex trigger type
`WerrInvIntLevel`	35	23H	Invalid interrupt level specified
`WerrFileOpenErr`	36	24H	Error opening specified disk file
`WerrFileWriteErr`	37	25H	Error writing specified disk file
`WerrUserOverrun`	38	26H	User-supplied buffer overrun (cycle mode)
`WerrInvTimeout`	39	27H	Invalid time-out value
`WerrInvInfo`	40	28H	Invalid channel information selector
`WerrInvOptionType`	41	29H	Invalid channel option type
`WerrInvOptionValue`	42	2AH	Invalid channel option value

Enhanced API Programming Models (WaveBook) 11

Overview

The enhanced Application Programming Interface (API) allows you to create custom software to satisfy your WaveBook data acquisition requirements. Chapters 11 and 12 give you the basic concepts and details to write effective programs. Chapter 12 describes the API functions in detail. This chapter explains how to combine those 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 WaveBook enhanced API is a subset of the `DaqX` API which provides a common interface for 32-bit data acquisition applications (WaveBook, DaqBook, DaqBoard, Daq PC-Card, etc). This manual describes the commands that pertain to the WaveBook.

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 12 (*WaveBook Command Reference Enhanced API)* supplies the details about how each function is used (parameters, hardware applicability, etc). In addition, you may need to consult your language and computer documentation.

Enhanced vs Standard API

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

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

Note: Coding for the enhanced and standard API cannot be used together; enhanced and standard models are slightly different (this chapter is for the enhanced API models; chapters 6 to 9 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

Important data acquisition concepts for programmers are listed here and explained in the chapter *Operation Guide (4)*. 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 12.
- **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 12 and *Triggering Capabilities* in chapter 4.
- **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

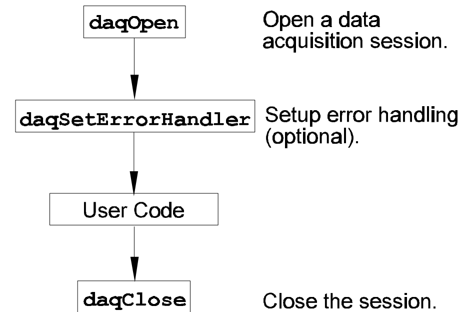
This section outlines basic programming steps commonly used for data acquisition. Consider the models as building blocks that can be put together in different ways or modified as needed. As a general tutorial, these examples use Visual Basic since most programmers know BASIC and can translate to other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

Model Type	Model Name	Page
Configuration	Initialization and Error Handling	11-3
Acquisition	Foreground Acquisition with One-Step Commands	11-5
	Counted Acquisition Using Linear Buffers	11-7
	Indefinite Acquisition, Direct-To-Disk Using Circular Buffers	11-9
	Multiple Hardware Scans, Software Triggering	11-12
	Background Acquisition	11-14
	Complex Triggering	11-16
Data Handling	Data Packing and Rotating	11-18
	Double Buffering	11-20
	Direct-to-Disk Transfers	11-22
	Transfers With Driver-Allocated Buffers	11-25

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&("WaveBook0")
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&("WaveBook0") <  0) Then
  "Cannot open DaqBook0"
```

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

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

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

```
handle& = VBdaqOpen&("WaveBook0")
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.

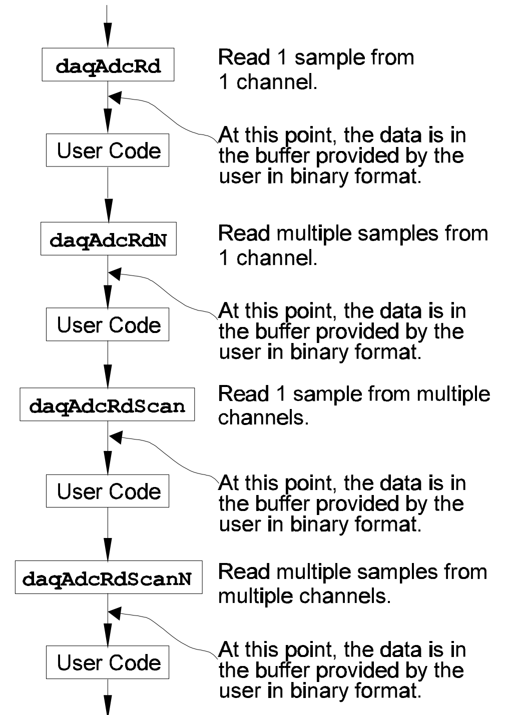
Daq* errors will send the program into the error handling routine. This is the error handler. Program control is sent here on error.

```
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!                             '1000Hz sample rate
Const gain& = DgainX1&                          'gain of x1
Const flags& = DafAnalog&+DafUnipolar&          'unipolar mode on
Const scans& = 9                                'number of scans to acquire
Const channels& = 8                             'number of channels to scan
Const rising& = DatdRisingEdge                  'XXXX I have no idea
Const HYSTERESIS& = 0.1                         'with a hysteresis of .1
Dim buf%(scans& * channels&)                    'array buffer to hold data
Dim handle&                                     'handle for WaveBook device
Dim i&, j&                                      'counter variables
Dim sample%                                     'hold a single reading
Dim ret&                                        '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 ×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 ×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&)
```

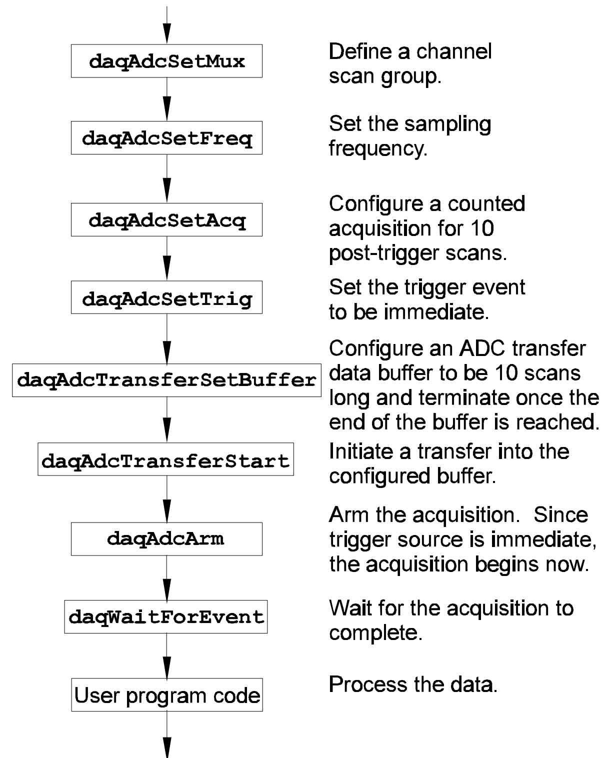
Counted Acquisitions Using Linear Buffers

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 ×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 WaveBook/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&)
```

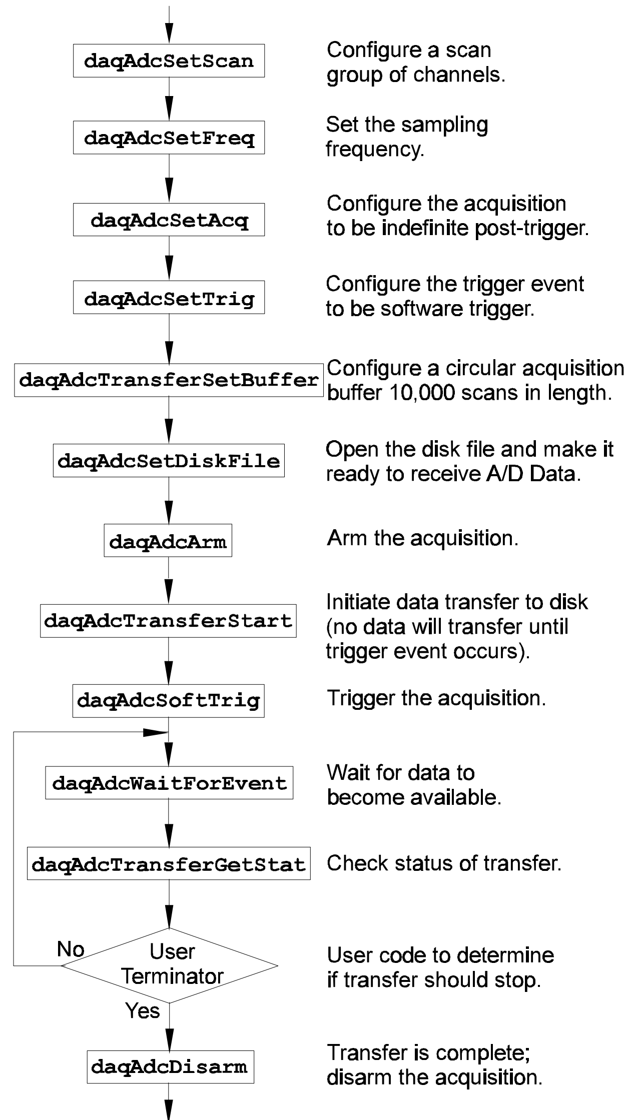
Wait for the transfer to complete in its entirety, then proceed with normal application processing. This can be accomplished with the **daqWaitForEvent** command. The **daqWaitForEvent** allows the application processing to become blocked until the specified event has occurred. **DteAdcDone**, indicates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

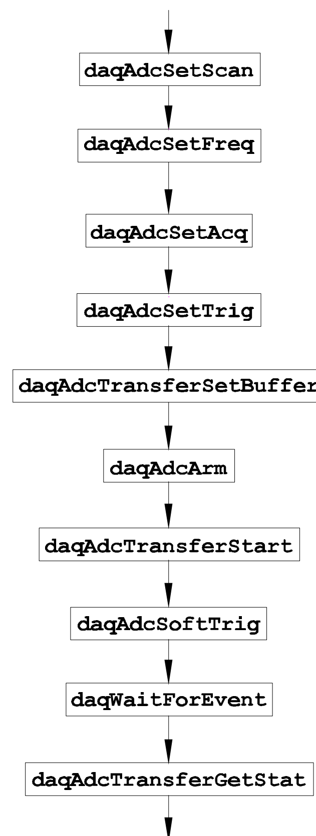
```
ret& = VBdaqAdcDisarm&(handle&)
```

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 WaveBook/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("WaveBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC3
```

Now set the scan configuration:

```
ret& = VBdaqAdcSetScan&(handle&,chans&(), gains&(), flagss&(), 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&)
```

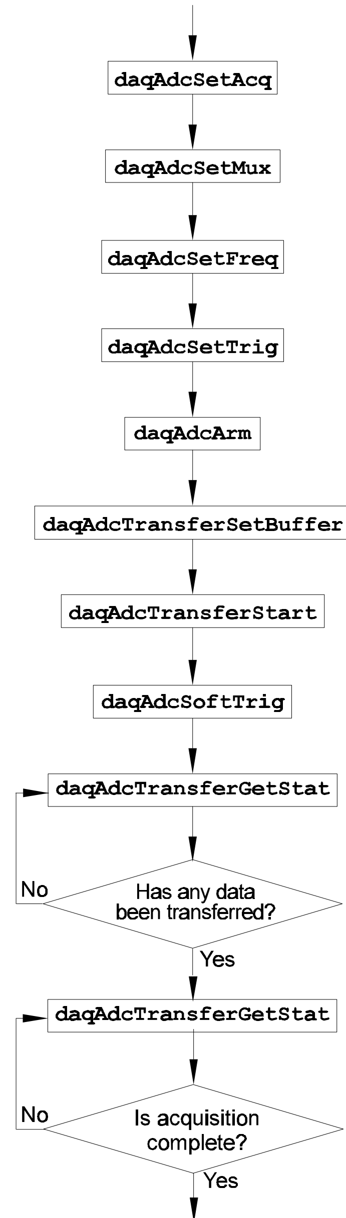
Once this function returns, the acquisition as well as the data transfer has been stopped. We should check the status one more time to get the total number of scans actually transferred to disk.

```
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("WaveBook0")`**
- **`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("WaveBook0")
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."
```

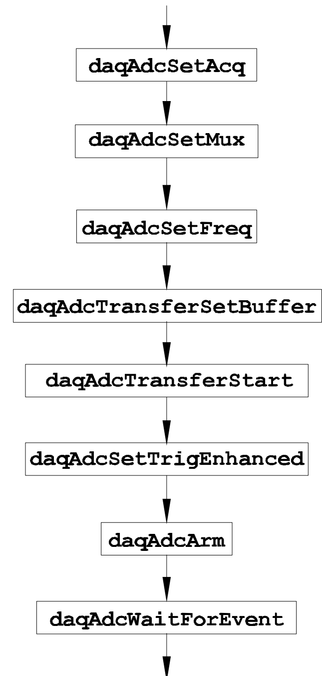
Now the data can be displayed or manipulated:

```
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&)
```

Complex Triggering



This example takes multiple scans from hardware using a complex analog trigger. The acquisition will start on a rising-edge of channel 1 at 2 volts OR a falling-edge on channel 2 at 3 volts.

Functions used include:

- `VBdaqAdcSetAcq&(handle&,mode&,preTrigCount&,postTrigCount&)`
- `VBdaqAdcSetMux&(handle&, startChan&, endChan&, gain&, flags&)`
- `VBdaqAdcSetTrig&(handle&, triggerSource&, rising&, level%, hysteresis%,channel&)`
- `VBdaqAdcSetFreq&(handle&, freq!)`
- `VBdaqAdcArm&(ByVal handle&, ByVal deviceType&)`
- `VBdaqAdcTransferSetBuffer&(handle&, deviceType&, chan&, buf%(), scanCount&, transferMask&)`
- `VBdaqAdcTransferStart&(handle&)`
- `VBdaqAdcSoftTrig&(handle&)`
- `VBdaqAdcTransferGetStat(handle&, active&, retCount&)`
- `VBdaqAdcClose(handle&)`

The constants used in this example are defined as follows:

```
Const freq! = 1000
Const scans& = 9
Const channels& = 3
Const NUM_TRIG& = 2
```

Arrays are dimensioned for configuring individual channels:

```
Dim chan&(NUM_TRIG&)
Dim gains&(NUM_TRIG&), polarity&(NUM_TRIG&)
Dim rising&(NUM_TRIG&)
Dim levels%(NUM_TRIG&), HYSTERESIS%(NUM_TRIG&)
```

Initialize these arrays which can specify individual settings for each channel:

```
chan&(0) = 1
gains&(0) = DgainX1&
polarity&(0) = 1
rising&(0) = DatdRisingEdge&
levels%(0) = 2
HYSTERESIS%(0) = 0.1

chan&(1) = 2
gains&(1) = DgainX1&
polarity&(1) = 1
rising&(1) = DatdFallingEdge&
levels%(1) = 3
HYSTERESIS%(1) = 0.1
```

Now, to open the WaveBook and specify the error handler:

```
handle& = VBdaqOpen("WaveBook0")
ret& = BdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC5
```

Set up the acquisition for `Nshot`, with 0 pre-trigger scans and 9 post-trigger scans.

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

Set the scan configuration to channels 1 through 3, with a gain of ×1, and the `flags` parameter specifying unipolar analog mode.

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&,
 DafAnalog&+DafUnipolar&)
```

Set the post-trigger scan rates:

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

Create the buffer to hold the collected data:

```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& +
  DatmUpdateSingle&)
```

Set a complex trigger at channels 1 and 2:

```
ret& = VBdaqAdcSetTrigEnhanced&(handle&, rising&(1),gains&(0),ranges&(0),
  DetsFallingEdge,levels(0)%,HYSTERESIS%(1), chan&(0))
```

`VBdaqAdcSetTrigEnhanced()` is used in place of **`VBdaqAdcSetTrig()`**. It defines the complex trigger event that will initiate the acquisition. Since it can configure independant trigger events on multiple channels, arrays of the parameters are given as arguments with each index referring to a specific channel.

Start the transfer:

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

Arm the acquisition; data transfer will begin upon detection of the trigger event defined by:

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

Wait for the transfer to end:

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

Results can now be displayed or otherwise manipulated:

```
Print "Results of BufferTransfer:"
   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&
```

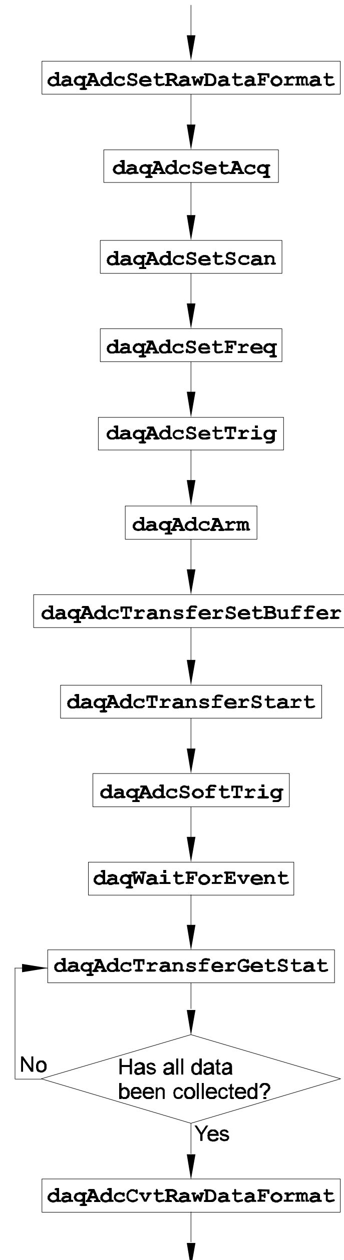
Close the device when finished:

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

Data Packing and Rotating

This section demonstrates an acquisition made up of pre-trigger and post-trigger scans from multiple channels using a DSP-based analog trigger. It also uses data packing and rotating. Functions used include:

- **VBdaqAdcArm&(handle&)**
- **VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)**
- **VBdaqAdcSetFreq&(handle&, POST_freq!)**
- **VBdaqAdcSetRawDataFormat(handle&, DadfPacked&)**
- **VBdaqAdcSetScan&(handle&, chans&(), gains&(), polarities&(), channels&)**
- **VBdaqAdcSetTrig&(handle&, DatsHardwareAnalog&, rising&, level%, HYSTERESIS%, chans&(0))**
- **VBdaqAdcSoftTrig&(handle&)**
- **VBdaqAdcTransferGetStat&(handle&, active&, retCount&)**
- **VBdaqAdcTransferSetBuffer(handle&, buf%(), BLOCK&, DatmCycleOff& + DatmUpdateSingle&)**
- **VBdaqAdcTransferStart(handle&)**
- **VBdaqClose(handle&)**
- **VBdaqCvtRawDataFormat&(buf%(), DacaUnpack, retCount&, BLOCK&, channels&)**
- **VBdaqOpen("WaveBook0")**
- **VBdaqSetErrorHandler(handle&, 100)**
- **VBdaqWaitForEvent(handle&, DteAdcData&)**



These declarations specify the number of channels, the pre- and post-scan counts and frequencies, and block size for the acquisition:

```
Const channels& = 4
Const PRE_SCANS& = 5
Const POST_SCANS& = 9
Const PRE_freq! = 100#
Const POST_freq! = 200#
Const BLOCK& = (PRE_SCANS& + POST_SCANS&)
```

Also declared are the trigger level, hysteresis, and rising parameters:

```
Const level% = 0
Const HYSTERESIS% = 0
Const rising& = 0
```

The buffer is dimensioned, as well as arrays to hold values for channels, gains, and polarities:

```
Dim buf%(BLOCK& * channels&)
Dim chans&(channels&), gains&(channels&),
  polarities&(channels&)
```

Initialize the arrays with the channel numbers and gains for each channel:

```
chans&(0) = 1        ' channel numbers
chans&(1) = 3
chans&(2) = 5
chans&(3) = 7
For i& = 0 To channels& - 1
   gains&(i&) = DgainX1&  ' unity gain
   polarities&(i&) = 1  ' bipolar
Next i&
```

Now to open the device and set up the error handler:

```
handle& = VBdaqOpen("WaveBook0")
   ret& = VBdaqSetErrorHandler(handle&, 100)
   On Error GoTo ErrorHandlerADC6
```

Specify the packed-data format for the transfer:

```
ret& = VBdaqAdcSetRawDataFormat(handle&, DadfPacked&)
```

Configure the acquisition for **Nshot**, with 9 post-trigger scans:

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

Set the scan configuration using the arrays initialized earlier:

```
ret& = VBdaqAdcSetScan&(handle&, chans&(), gains&(), polarities&(),
channels&)
```

Set the post-trigger scan rate:

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

Set the trigger source to an analog trigger on channel 1 at 2 volts:

```
ret& = VBdaqAdcSetTrig&(handle&, DatsHardwareAnalog&, rising&, level%,
HYSTERESIS%, chans&(0))
```

Arm the acquisition:

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

Start reading data in the background mode with cycle mode and updateSingle off:

```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf%(), BLOCK&,
DatmCycleOff& + DatmUpdateSingle&)
```

Start the transfer:

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

Issue a software trigger command to the hardware, and wait for the transfer to begin:

```
ret& = VBdaqAdcSoftTrig&(handle&)
retCount& = 0
While retCount& < BLOCK '= 0
  ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
Print "Triggered. Transfer in progress."
```

Wait for first data to be received:

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

Wait for the rest of the data to be transmitted:

```
While retCount& < BLOCK
   ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
Print "Acquisition complete:"; retCount&; "scans acquired."
Print
```

Unpack the packed data using the same buffer:

```
ret& = VBdaqCvtRawDataFormat&(buf%(), DacaUnpack, retCount&, BLOCK&,
channels&)
```

Rotate the unpacked data so that the earliest data starts at the beginning of the buffer and the latest is at the end:

```
ret& = VBdaqCvtRawDataFormat&(buf%(), DacaRotate, retCount&, BLOCK&,
channels&)
```

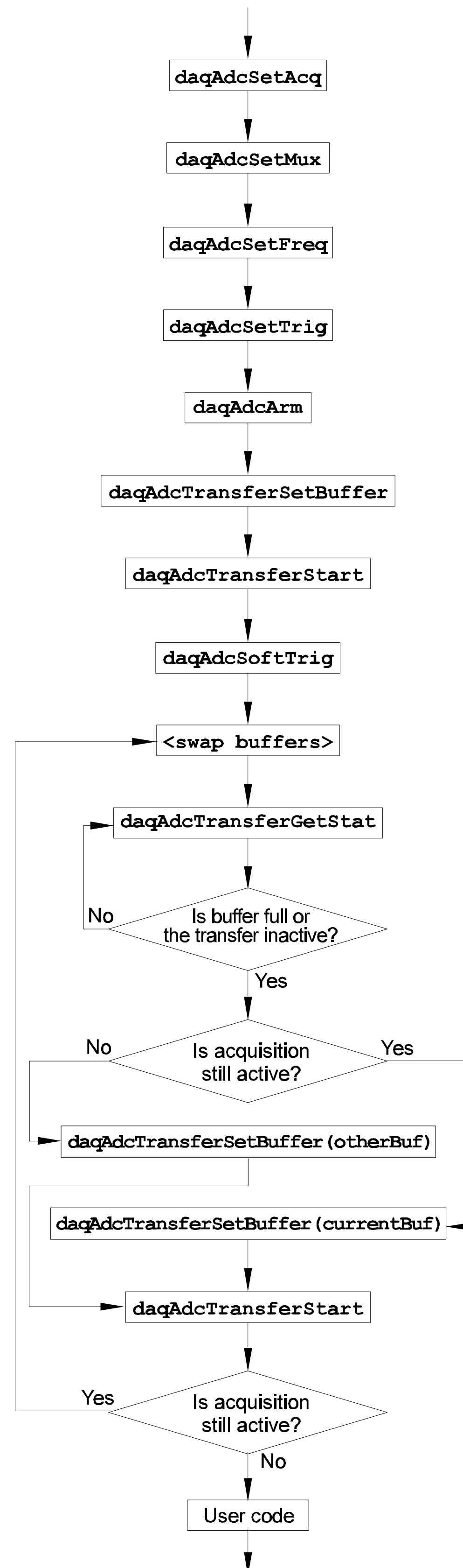
Display the results:

```
Print "Pre-trigger data acquired:"
For i& = 0 To channels& - 1
   Print "Channel"; i& + 1; "Data:";
   For j& = 0 To PRE_SCANS& - 1
      Print Tab(j& * 7 + 17); buf%(j& * channels& + i&);
   Next j&
   Print
Next i&
Print
Print "Post-trigger data acquired:"
For i& = 0 To channels& - 1
   Print "Channel"; i& + 1; "Data:";
   For j& = PRE_SCANS& To BLOCK& - 1
      Print Tab((j& - PRE_SCANS&) * 7 + 17); buf%(j& * channels& +
      i&);
   Next j&
   Print
Next i&
```

Finally, close the device before exiting:

```
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("WaveBook0")`
- `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 WaveBook:

```
handle& = VBdaqOpen("WaveBook0")
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)
```

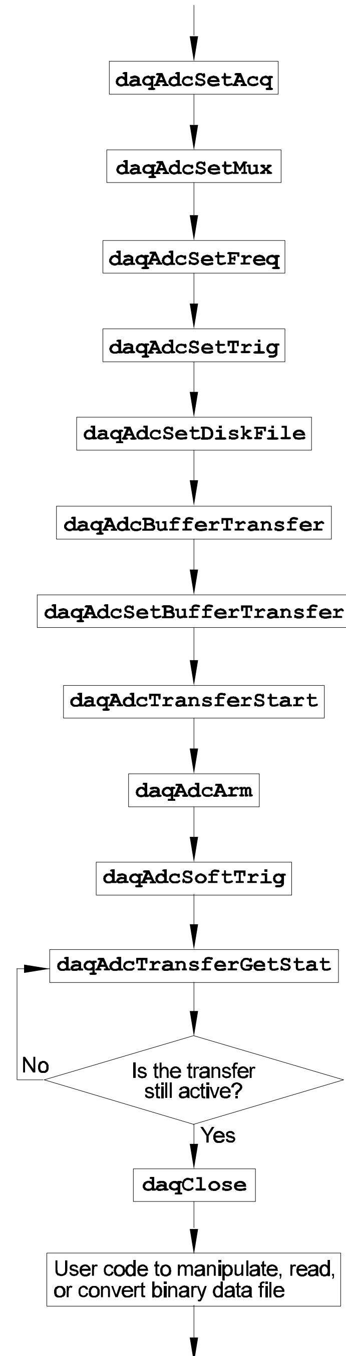
Close the device before exiting:

```
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:

- `VBdaqAdcSetRawDataFormat&(handle&, DadfPacked&)`
- `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%(), DacaUnpack, BLOCK&, channels&, scanCount&)`
- `VBdaqOpen&("WaveBook0")`
- `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&("WaveBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC8
```

Enable data packing:

```
ret& = VBdaqAdcSetRawDataFormat&(handle&, DadfPacked&)
```

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
```

Next, convert the data just read into the buffer from packed bytes to unpacked scans:

```
wordCount& = byteCount& / 2
sampleCount& = wordCount& * 4 / 3
scanCount& = sampleCount& / channels&
```

Unpack the packed data using the same buffer. This command can be called even if the WaveBook if not online or connected.

```
ret& = VBdaqCvtRawDataFormat&(buf%(), DacaUnpack, BLOCK&,
  channels&, scanCount&)
```

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:

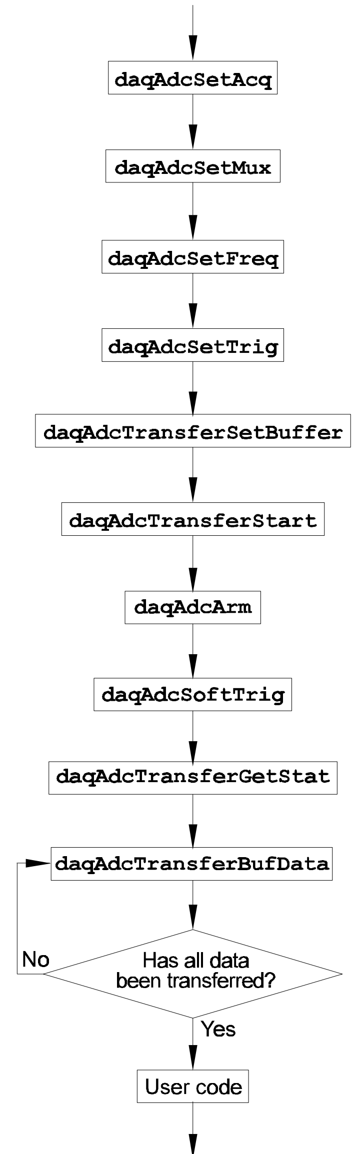
```
      voltage! = buf%(i& * channels& + j&) * 5! / 32768!
      Print #2, Format$(voltage!, ".000") + termChar$;
   Next j&
Next i&
```

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 WaveBook 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("WaveBook0")**
- **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("WaveBook0")
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)
```

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 notes on when to use them.

Simple One-Step Routines

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

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

Complex A/D Scan Group Configuration Routines

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

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

Trigger Options

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

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

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

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

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

Multiple Scan Timing

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

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

A/D Acquisition

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

Function	Description
`daqAdcArm`	Arm an A/D acquisition using the current configuration. If the trigger source was set to be immediate, the acquisition will be triggered immediately.
`daqAdcDisarm`	Disarm the current acquisition if one is active. This command will disarm the current acquisition and terminate any current A/D transfers.
`daqAdcSetAcq`	Define the mode of the acquisition and set the pre-trigger and post-trigger acquisition counts, if applicable.

A/D Data Transfer

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

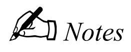
Function	Description
`daqAdcTransferSetBuffer`	Configure a buffer for A/D transfer. Allows configuration of the buffer for block and single reading update modes as well as linear and circular buffer definitions.
`daqAdcTransferStart`	Start a transfer from the Daq* device to the buffer specified in the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

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

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

Digital Functions

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

 *Notes*

WaveBook Command Reference (Enhanced API) 12

Overview

The first part of this chapter describes the WaveBook 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 WaveBook API is a subset of the Daq* API which also applies to other products; only WaveBook-related commands are described here.

Beginning on page 12-38, 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 WaveBook)	12-33
daqClose	End communication with the Daq* (including WaveBook)	12-25
daqOnline	Check online status of the Daq* (including WaveBook)	12-32
daqGetDeviceCount	Return the number of currently configured devices	12-28
daqGetDeviceList	Return the list of currently configured devices	12-28
daqGetDeviceProperties	Return the properties of specified device	12-29
Error Handler Function Prototypes		
daqSetDefaultErrorHandler	Set the default error handler	12-34
daqSetErrorHandler	Specify a user defined routine to call when an error occurs in any command	12-34
daqProcessError	Process a driver defined error condition	12-33
daqGetLastError	Return the last logged error condition	12-30
daqDefaultErrorHandler	Call the default error handler	12-27
daqFormatError	Return text string for specified error	12-27
Event Handling Function Prototypes		
daqSetTimeout	Set the time-out value for the Daq* operation (including WaveBook)	12-35
daqWaitForEvent	Wait for specified Daq* device event (including WaveBook)	12-37
daqWaitForEvents	Wait for multiple specified Daq* device events (including WaveBook)	12-37
Utility Function Prototypes		
daqGetDriverVersion	Return the software version	12-29
daqGetHardwareInfo	Return the hardware version	12-29
Expansion Configuration Prototypes		
daqAdcExpSetBank	Set bank specific configurations	12-5
daqAdcExpSetChanOption	Set channel specific configurations	12-5
daqAdcExpSetModuleOption	Set module specific configurations	12-6
daqSetOption	Set options for a device's channel/signal path configuration	12-35
Custom ADC Acquisition Prototypes - Scan Sequence		
daqAdcSetMux	Configure a scan specifying start and end channels	12-14
daqAdcSetScan	Configure up to 256 channels making up an A/D or HS digital input scan	12-15
daqAdcGetScan	Read the current scan configuration	12-7
Custom ADC Acquisition Prototypes - Trigger		
daqAdcCalcTrig	Calculate the trigger level and trigger source for an analog trigger	12-4
daqAdcSetTrig	Configure an A/D trigger	12-16
daqAdcSetTrigEnhanced	Configure an A/D trigger with multiple trigger-event conditions	12-17
daqAdcSoftTrig	Save a software trigger command to the DaqBook/DaqBoard	12-18
Custom ADC Acquisition Prototypes - Scan Rate and Source		
dacAdcSetRate	Configure the ADC scan rate with the `mode` parameter	12-14
daqAdcSetFreq	Configure the pacer clock frequency in Hz	12-13
daqAdcGetFreq	Read the current pacer clock frequency	12-6
Custom ADC Acquisition Prototypes - Scan Count, Rate and Source		
daqAdcSetAcq	Set acquisition configuration information	12-11
Custom ADC Acquisition Prototypes - Direct-to-Disk		
daqAdcSetDiskFile	Specify the disk file for direct-to-disk transfers	12-13
Custom ADC Acquisition Prototypes - Acquisition Control		
daqAdcArm	Arm an acquisition	12-2
daqAdcDisarm	Disarm an acquisition	12-4

Function	Description	Page
Custom ADC Acquisition Prototypes - Data Transfer without Buffer Allocation		
`daqAdcTransferBufData`	Transfer scans from driver-allocated buffer to user-specified buffer	12-19
`daqAdcTransferSetBuffer`	Setup a destination buffer for an ADC transfer	12-21
`daqAdcTransferStart`	Start an ADC transfer	12-22
`daqAdcTransferGetStat`	Retrieve status of an ADC transfer	12-20
`daqAdcTransferStop`	Stop an ADC transfer	12-22
Custom ADC Acquisition Prototypes - Buffer Manipulation		
`daqAdcBufferRotate`	Reorganize a circular buffer so that oldest data is oriented towards the front	12-3
One-Step ADC Acquisition Prototypes		
`daqAdcRd`	Configure an A/D acquisition and read one sample from a channel	12-7
`daqAdcRdScan`	Configure an A/D acquisition and read one scan	12-9
`daqAdcRdN`	Configure an A/D acquisition and read multiple scans from a channel	12-8
`daqAdcRdScanN`	Configure an A/D acquisition and read multiple scans	12-10
Data Format and Conversion Prototypes		
`daqAdcSetDataForma`	Set the raw and the post-acquisition data formats	12-12
`daqCvtRawDataFormat`	Convert raw data to a specified format	12-26
`daqCvtSetAdcRange`	Set the ADC Voltage Range for the conversion routines	12-27
Software Calibration Prototypes		
`daqCalSelectCalTable`	Select calibration-table source for the device	12-24
`daqCalSelectInputSignal`	Select input signal source for user calibration	12-24
`daqCalGetConstants`	Get calibration constants from selected calibration table	12-23
`daqCalSetConstants`	Set user-accessible calibration constants	12-25
`daqCalSaveConstants`	Save current calibration table	12-23
General I/O Prototypes - Read/Write		
`daqIOReadBit`	Read a DIO bit (channel)	12-31
`daqIORead`	Read a DIO byte (8 channels)	12-30
`daqIOWriteBit`	Write a DIO bit (channel)	12-32
`daqIOWrite`	Write a DIO byte (8 channels)	12-31
Test Prototypes		
`daqTest`	Perform a specified test on a Daq* device	12-36

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 *API Error Codes* on page 12-42)
See Also	`daqAdcDisarm`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

daqAdcBufferRotate

DLL Function	`daqAdcBufferRotate(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD chanCount, DWORD retCount);`
C	`daqAdcBufferRotate(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD chanCount, DWORD retCount);`
Visual BASIC	`VBdaqAdcBufferRotate&(ByVal handle&, buf%(), ByVal scanCount&, ByVal chanCount&, ByVal retCount&)`
Delphi	`daqAdcBufferRotate(handle:DaqHandleT; buf:PWORD; scanCount:DWORD; chanCount:DWORD; retCount:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC transfer buffer is to be rotated
`buf`	Pointer to the buffer to rotate
`scanCount`	Total number of scans in the buffer
`chanCount`	Number of channels in each scan
`retCount`	Last value returned in the retCount parameter of the daqAdcTransferGetStat function
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcTransferGetStat, daqAdcTransferSetBuffer`
Program References	None
Used With	All devices

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

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

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

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

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

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

Note: daqAdcBufferRotate only works on unpacked samples.

daqAdcCalcTrig

DLL Function	`daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);`
C	`daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);`
Visual BASIC	`VBdaqAdcCalcTrig&(ByVal handle&, ByVal bipolar&, ByVal gainVal!, ByVal voltageLevel!, triggerLevel%)`
Delphi	`daqAdcCalcTrig(handle:DaqHandleT; bipolar:longbool; gainVal:single; voltageLevel:single; var triggerLevel:DWORD)`
Parameters	
`handle`	Handle to the device for which the trigger level is to be calculated
`bipolar`	A flag that should be non-zero if the trigger channel is bipolar, or zero if it is unipolar
`gainVal`	A gain value of the trigger channel
`voltageLevel`	Voltage level to trigger at.
`triggerLevel`	Returned count to program the trigger using the daqAdcSetTrig function
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetTrig`
Program References	None
Used With	All devices

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

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

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

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

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

daqAdcDisarm

DLL Function	`daqAdcDisarm(DaqHandleT handle);`
C	`daqAdcDisarm(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcDisarm&(ByVal handle&)`
Delphi	`daqAdcDisarm(handle:DaqHandleT)`
Parameters	
`handle`	handle to the device to disable ADC acquisitions
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcArm`
Program References	None
Used With	All devices

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

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

daqAdcExpSetBank

DLL Function	`daqAdcExpSetBank(DaqHandleT handle, DWORD chan, DaqAdcExpType bankType);`
C	`daqAdcExpSetBank(DaqHandleT handle, DWORD chan, DaqAdcExpType bankType);`
Visual BASIC	`VBdaqAdcExpSetBank&(ByVal handle&, ByVal chan&, ByVal bankType&)`
Delphi	`daqAdcExpSetBank(handle:DaqHandleT; chan:DWORD; bankType:DaqAdcExpType)`
Parameters	
`handle`	Handle to the device for which to set the expansion bank
`chan`	Channel number on the DBK card. Channel numbers are in groups of 16 channels per bank.
`bankType`	Type of channel bank.
Returns	`DerrInvChan` - Invalid Channel Number (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcExpSetChanOption, daqAdcExpSetModuleOption, daqAdcSetOption`
Program References	None
Used With	All devices

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

daqAdcExpSetChanOption

DLL Function	`daqAdcExpSetChanOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
C	`daqAdcExpSetChanOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
Visual BASIC	`VBdaqAdcExpSetChanOption&(ByVal handle&, ByVal chan&, ByVal optionType&, ByVal optionValue!)`
Delphi	`daqAdcExpSetChanOption(handle:DaqHandleT; chan:DWORD; const optionType:DaqChanOptionType; optionValue:single)`
Parameters	
`handle`	Handle to the device for which to set the channel option
`chan`	The number of the channel to be configured.
`optionType`	The configurable option to be set (see table DBK Card Definitions)
`optionValue`	The configurable option to be set (see table DBK Card Definitions)
Returns	`DerrNoError` - No Errors (also, refer to *API Error Codes* on page 12-42) `DerrInvChan` - Invalid Channel Number
See Also	`daqAdcExpSetModuleOption, daqAdcSetOption`
Program References	None
Used With	All devices

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

daqAdcExpSetModuleOption

DLL Function	`daqAdcExpSetModuleOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
C	`daqAdcExpSetModuleOption(DaqHandleT handle, DWORD chan, DaqChanOptionType optionType, FLOAT optionValue);`
Visual BASIC	`VBdaqAdcExpSetModuleOption&(ByVal handle&, ByVal chan&, ByVal optionType&, ByVal optionValue!)`
Delphi	`daqAdcExpSetModuleOption(handle:DaqHandleT; chan:DWORD; const optionType:DaqChanOptionType; optionValue:single)`
Parameters	
`handle`	Handle to the device for which to set the module option.
`chan`	Any channel on the module (expansion chassis) to be configured.
`optionType`	The configurable option to be set (see table *DBK Card Definitions*).
`optionValue`	The configurable option to be set (see table *DBK Card Definitions*).
Returns	An error number, or 0 if no error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcExpSetChanOption, daqAdcSetOption`
Program References	None
Used With	All devices

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

daqAdcGetFreq

DLL Function	`daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);`
C	`daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);`
Visual BASIC	`VBdaqAdcGetFreq&(ByVal handle&, freq!)`
Delphi	`daqAdcGetFreq(handle:DaqHandleT; var freq:single)`
Parameters	
`handle`	Handle to the device for which to get the current frequency setting
`freq`	A variable to hold the currently defined sampling frequency in Hz Valid values: `100000.0 - 0.0002`
Returns	`DerrNoError` - No errors (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetFreq, daqAdcSetClock`
Program References	None
Used With	All devices

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

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

daqAdcGetScan

DLL Function	`daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);`
C	`daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);`
Visual BASIC	`VBdaqAdcGetScan&(ByVal handle&, channels&(), gains&(), flags&(), chanCount&)`
Delphi	`daqAdcGetScan( handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:PDWORD )`
Parameters	
`handle`	Handle to the device for which to get the current scan configuration.
`channels`	An array to hold up to 512 channel numbers or 0 if the channel information is not desired.
`*gains`	An array to hold up to 512 gain values or 0 if the channel gain information is not desired
`flags`	Channel configuration flags in the in the form of a bit mask
`chanCount`	A variable to hold the number of values returned in the chans and gains arrays
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetScan, daqAdcSetMux`
Program References	None
Used With	All devices

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

daqAdcRd

DLL Function	`daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRd&(ByVal handle&, ByVal chan&, sample%, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRd(handle:DaqHandleT; chan:DWORD; var sample:WORD; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC reading is to be acquired
`chan`	A single channel number
`sample`	A pointer to a value where an A/D sample is stored. Valid values: (See **`daqAdcSetTag`**)
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrFIFOFull` - Buffer Overrun `DerrInvGain` - Invalid gain `DerrInvChan` - Invalid channel `DerrNoError` - No Error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetMux, daqAdcSetTrig, daqAdcSoftTrig`
Program References	DACEX.PAS (Delphi)
Used With	All devices

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

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

daqAdcRdN

DLL Function	`daqAdcRdN(DaqHandleT handle, DWORD chan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdN(DaqHandleT handle, DWORD chan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdN&(ByVal handle&, ByVal chan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdN(handle:DaqHandleT; chan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC channel samples are to be acquired
`chan`	A single channel number
`buf`	An array where the A/D scans will be returned
`scanCount`	The number of scans to be taken Valid values: `1 - 32767`
`triggerSource`	The trigger source
`rising`	Boolean flag to indicate the rising or falling edge for the trigger source
`level`	The trigger level if an analog trigger is specified Valid values: `0 -4095`
`freq`	The sampling frequency in Hz (`100000.0 to 0.0002`)
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrFIFOFull` - Buffer overrun `DerrInvGain` -Invalid gain `DerrIncChan` - Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid level (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetFreq, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig`
Program References	None
Used With	All devices

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

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

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

daqAdcRdScan

DLL Function	`daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device from which the ADC scan is to be acquired
`startChan`	The starting channel of the scan group
`endChan`	The ending channel of the scan group
`buf`	An array where the A/D scans will be placed
`gain`	The channel gain
`flags`	Channel configuration flags in the form of a bit mask.
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcRdNScan, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig`
Program References	DACEX.PAS (Delphi)
Used With	All devices

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

daqAdcRdScanN

DLL Function	`daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
C	`daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device from which ADC scans are to be acquired
`startchan`	The starting channel of the scan group (see table at end of chapter)
`endchan`	The ending channel of the scan group (see table at end of chapter)
`buf`	An array where the A/D scans will be placed
`scanCount`	The number of scans to be read Valid values: `1 - 65536`
`triggerSource`	The trigger source (see table at end of chapter)
`rising`	Boolean flag to indicate the rising or falling edge for the trigger source
`level`	The trigger level if an analog trigger is specified Valid values: `0 -4095`
`freq`	The sampling frequency in Hz Valid values: `100000.0 - 0.0002`
`gain`	The channel gain (See tables at end of chapter).
`flags`	Channel configuration flags in the form of a bit mask.
Returns	`DerrInvGain` - Invalid gain `DerrInvChan` -Invalid channel `DerrInvTrigSource` - Invalid trigger `DerrInvLevel` - Invalid Level `DerrFIFOFull` -Buffer Overrun `DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcRd, daqAdcRdN, daqAdcRdScan, daqAdcSetClock, daqAdcSetTrig`
Program References	None
Used With	All devices

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

daqAdcSetAcq

DLL Function	`daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);`
C	`daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);`
Visual BASIC	`VBdaqAdcSetAcq&(ByVal handle&, ByVal mode&, ByVal preTrigCount&, ByVal postTrigCount&)`
Delphi	`daqAdcSetAcq(handle:DaqHandleT; mode:DaqAdcAcqMode; preTrigCount:DWORD; postTrigCount:DWORD)`
Parameters	
`handle`	Handle to the device for which the ADC acquisition is to be configured
`mode`	Selects the mode of the acquisition
`preTrigCount`	Number of pre-trigger ADC scans to be collected
`postTrigCount`	Number of post-trigger ADC scans to be collected
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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	`daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);`
C	`daqAdcSetDataFormat(DaqHandleT handle, DaqAdcRawDataFormatT rawFormat, DaqAdcPostProcDataFormatT postProcFormat);`
Visual BASIC	`VBdaqAdcSetDataFormat &(ByVal handle&, ByVal rawFormat&, ByVal postProcFormat&)`
Delphi	`daqAdcSetDataFormat(Handle:DaqHandleT; rawFormat:DaqAdcRawDataFormatT rawFormat; postProcFormat:DaqAdcPostProcDataFormatT);`
Parameters	
`handle`	The handle to the device for which to set the option
`rawFormat`	The channel number on the device for which the option is to be set
`postProcFormat`	Flags specifying the options to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCvtRawDataFormat,daqCvtRawDataFormat`
Program References	None
Used With	All devices

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

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

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

daqAdcSetDiskFile

DLL Function	`daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);`
C	`daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);`
Visual BASIC	`VBdaqAdcSetDiskFile&(ByVal handle&, ByVal filename$, ByVal openMode&, ByVal preWrite&)`
Delphi	`daqAdcSetDiskFile(handle:DaqHandleT; filename:PChar; openMode:DaqAdcOpenMode; preWrite:DWORD)`
Parameters	
`handle`	Handle to the device for which direct to disk ADC acquisition is to be performed.
`filename`	String representing the path and name of the file to place the raw ADC acquisition data.
`openMode`	Specifies how to open the file for writing
`preWrite`	Specifies the amount to pre-write(in bytes) the file
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcTransferGetStat, daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

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

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

daqAdcSetFreq

DLL Function	`daqAdcSetFreq(DaqHandleT handle, FLOAT freq);`
C	`daqAdcSetFreq(DaqHandleT handle, FLOAT freq);`
Visual BASIC	`VBdaqAdcSetFreq&(ByVal handle&, ByVal freq!)`
Delphi	`daqAdcSetFreq(handle:DaqHandleT; freq:single)`
Parameters	
`handle`	Handle to the device for which the ADC acquisition frequency is to be set.
`freq`	The sampling frequency in Hz Valid values: `100000.0 - 0.0002`
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcGetFreq, daqAdcSetClockSource`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

daqAdcSetMux

DLL Function	`daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);`
C	`daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);`
Visual BASIC	`VBdaqAdcSetMux&(ByVal handle&, ByVal startChan&, ByVal endChan&, ByVal gain&, ByVal flags&)`
Delphi	`daqAdcSetMux(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; const gain:DaqAdcGain; flags:DWORD)`
Parameters	
`handle`	Handle to the device for which to configure the ADC channel scan group
`startChan`	The starting channel of the scan group
`endChan`	The ending channel of the scan group
`gain`	The gain value for all channels
`flags`	Channel configuration flags in the form of a bit mask
Returns	`DerrInvGain` - Invalid gain `DerrIncChan` - Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	**`daqAdcSetScan, daqAdcGetScan`**
Program References	DACEX1.C, DAQEX.FRM (VB)
Used With	All devices

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

daqAdcSetRate

DLL Function	`daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);`
C	`daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);`
Visual BASIC	`VBdaqAdcSetRate(ByVal handle&, ByVal mode&, ByVal acqState&, ByVal reqRate!, actualRate!);`
Delphi	`daqAdcSetRate(handle: DaqHandleT; mode: DaqAdcRateMode, acqState: DaqAdcAcqState; reqRate:FLOAT; actualRate:PFLOAT);`
Parameters	
`handle`	Handle to the device for which to set ADC scanning frequency.
`mode`	Specifies the rate mode (frequency or period).
`acqState`	Specifies the acquisition state to which the rate is to be applied.
`reqRate`	Specifies the requested rate.
`actualRate`	Returns the actual rate applied. This may be different from the requested rate.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	**`daqAdcSetAcq, daqAdcSetTrig, daqAdcArm, daqAdcSetFreq, daqAdcGetFreq`**
Program References	
Used With	All devices

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

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

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

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

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

daqAdcSetScan

DLL Function	`daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);`
C	`daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);`
Visual BASIC	`VBdaqAdcSetScan&(ByVal handle&, channels&(), gains&(), flags&(), ByVal chanCount&)`
Delphi	`daqAdcSetScan(handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:DWORD)`
Parameters	
`handle`	Handle to the device for which ADC scan group is to be configured
`channels`	An array of up to 512 channel numbers
`*gains`	An array of up to 512 gain values
`flags`	Channel configuration flags in the form of a bit mask
`chanCount`	The number of values in the chans and gains arrays Valid values: `1 -512`
Returns	`DerrNotCapable` - No high speed digital `DerrInvGain` - Invalid gain `DerrInvChan` - Invalid channel `DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcGetScan, daqAdcSetMux`
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

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

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

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

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

daqAdcSetTrig

DLL Function	daqAdcSetTrig(DaqHandleT handle, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, WORD hysteresis, DWORD channel);
C	daqAdcSetTrig(DaqHandleT handle, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, WORD hysteresis, DWORD channel);
Visual BASIC	VBdaqAdcSetTrig&(ByVal handle&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal hysteresis%, ByVal channel&)
Delphi	daqAdcSetTrig(handle:DaqHandleT; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; hysteresis:WORD; channel:DWORD)
Parameters	
handle	Handle to the device for which the ADC acquisition trigger is to be configured.
triggerSource	Sets the trigger source.
rising	Boolean flag to indicate the rising or falling edge for the trigger source
level	The trigger level (in A/D counts) for an analog level trigger
hysteresis	hysteresis value for analog level trigger (if selected)
channel	Channel for which the analog level trigger(if selected) is to be detected.
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 12-42)
See Also	daqAdcSetAcq
Program References	ADCEX1.C, DACEX1.C, DAQEX.FRM (VB), ADCEX.PAS, ERREX.PAS (Delphi)
Used With	All devices

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

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

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

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

daqAdcSetTrigEnhanced

DLL Function	`daqAdcSetTrigEnhanced(DaqHandleT handle, DaqAdcTriggerSource *triggerSources, PDWORD gains, PDWORD adcRanges, DaqEnhTrigDef trigDef, PFLOAT levels, PFLOAT hysteresis, PDWORD channels,DWORD chanCount, char *opStr);`
C	`daqAdcSetTrigEnhanced(DaqHandleT handle, DaqAdcTriggerSource *triggerSources, PDWORD gains, PDWORD adcRanges, DaqEnhTrigDef trigSense, PFLOAT levels, PFLOAT hysteresis, PDWORD channels,DWORD chanCount, char *opStr);`
Visual BASIC	`VBdaqAdcSetTrigEnhanced&(ByVal handle&, triggerSources&, gains&, adcRanges&, trigSense&, levels!, hysteresis!, channels&, chanCount&, opStr$)`
Delphi	`daqAdcSetTrigEnahanced(handle:DaqHandleT; triggerSources:DaqAdcTriggerSource; gains: PDWORD; adcRanges: PDWORD; trigSense:DaqEnhTrigDef; levels : PFLOAT; hysteresis : PFLOAT; channels:PDWORD; chanCount:DWORD; opStr: String )`
Parameters	
`handle`	Handle to the device for which the ADC acquisition trigger is to be configured.
`triggerSource`	A pointer to an array of trigger sources for each defined trigger channel.
`gains`	A pointer to an array of gains for each defined A/D trigger channel.
`levels`	A pointer to an array of A/D analog trigger levels for each defined A/D trigger channel.
`hysteresis`	A pointer to an array of hysteresis values for each defined A/D trigger channel.
`trigSense`	A pointer to an array of trigger sensitivity flags for each defined A/D channel trigger source.
`adcRanges`	A pointer to an array of polarity flag definitions for each defined A/D channel.
`channels`	A pointer to an array of trigger channels representing the actual A/D trigger channels to trigger on.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetAcq, daqAdcSetTrig, daqAdcSetScan`
Program References	
Used With	WaveBook/512, WaveBook/516

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

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

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

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

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

daqAdcSoftTrig

DLL Function	`daqAdcSoftTrig(DaqHandleT handle);`
C	`daqAdcSoftTrig(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcSoftTrig&(ByVal handle&)`
Delphi	`daqAdcSoftTrig(handle:DaqHandleT)`
Parameters	
`handle`	Handle to the device to which the ADC software trigger is to be applied
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcSetTrig, daqAdcSetAcq`
Program References	None
Used With	All devices

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

daqAdcTransferBufData

DLL Function	`daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);`
C	`daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);`
Visual BASIC	`VBdaqAdcTransferBufData(ByVal handle, buf%, ByVal scanCount&, ByVal bufMask&, retCount&);`
Delphi	`daqAdcTransferBufData(handle: DaqHandleT; buf : PWORD, scanCount : DWORD, bufMask: DaqAdcBufferXferMask; retCount: PDWORD);`
Parameters	
`handle`	Handle to the device for which the ADC buffer should be retrieved.
`buf`	Pointer to an application-supplied buffer to place the buffered data.
`scanCount`	Number of scans to retrieve from the acquisition buffer.
`bufMask`	A mask defining operation depending on the current state of the acquisition buffer
`retCount`	A pointer to the total number of scans returned, if any.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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 *API Error Codes* on page 12-42)
See Also	`daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop`
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

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

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

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

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

daqAdcTransferSetBuffer

DLL Function	DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);
C	DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);
Visual BASIC	VBdaqAdcTransferSetBufferAllocMem&(ByVal handle&, ByVal scanCount&, ByVal transferMask&)
Delphi	daqAdcTransferSetBufferAllocMem(handle:DaqHandleT; scanCount:DWORD; transferMask:DWORD)
Parameters	
handle	Handle to the device for which an ADC transfer is to be performed.
buf	Pointer to the buffer for which the acquired data is to be placed.
scanCount	The total length of the buffer (in scans).
transferMask	Configures the buffer transfer mode.
Returns	DerrNoError - No error (also, refer to *API Error Codes* on page 12-42)
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 *API Error Codes* on page 12-42)
See Also	`daqAdcTranferSetBuffer, daqAdcTransferGetStat, daqAdcTransferStop`
Program References	ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

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

daqAdcTransferStop

DLL Function	`daqAdcTransferStop(DaqHandleT handle);`
C	`daqAdcTransferStop(DaqHandleT handle);`
Visual BASIC	`VBdaqAdcTransferStop&(ByVal handle&)`
Delphi	`daqAdcTransferStop( handle:DaqHandleT )`
Parameters	
`handle`	Handle to the device for which the Adc data transfer is to be stopped
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferGetStat`
Program References	None
Used With	All devices

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

daqCalGetConstants

DLL Function	`daqCalGetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, PWORD gainConstant, PSHORT offsetConstant );`
C	`daqCalGetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, PWORD gainConstant, PSHORT offsetConstant );`
Visual BASIC	`VBdaqCalGetConstants(ByVal handle&, ByVal channel&, ByVal gain&, ByVal range&, al gainConstant%, offsetConstant% );`
Delphi	`daqCalGetConstants(handle: DaqHandleT;channel:DWORD; gain: DaqAdcGain; range: DaqAdcRangeT; gainConstant:PWORD; offsetConstant:PSHORT );`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`channel`	Channel number to apply the calibration settings
`gain`	Gain range to apply the calibration settings
`range`	A/D input range to apply the calibration settings
`gain`	Pointer to the gain value for the current entry
`offset`	Pointer to the offset value for the current entry
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCalSetConstants, daqCalSelectCalTable, daqCalSelectInputSignal,daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

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

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

daqCalSaveConstants

DLL Function	`daqCalSaveConstants(DaqHandleT handle, DWORD channel );`
C	`daqCalSaveConstants(DaqHandleT handle, DWORD channel );`
Visual BASIC	`VBdaqCalSaveConstants(ByVal handle&, ByVal channel& )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; channel: DWORD)`
Parameters	
`handle`	Handle to the device for which the calibration constants are to be saved.
`channel`	Channel to save to the current calibration settings for
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectInputSignal, daqCalSelectCalTable`
Program References	None
Used With	WaveBook/512, WaveBook/516

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

daqCalSelectCalTable

DLL Function	`daqCalSelectCalTable(DaqHandleT handle, DaqCalTableTypeT tableType);`
C	`daqCalSelectCalTable(DaqHandleT handle, DaqCalTableTypeT tableType);`
Visual BASIC	`VBdaqCalSelectCalTable(ByVal handle&, ByVal tableType as DaqCalTableTypeT )`
Delphi	`daqCalSelectCalTable( handle: DaqHandleT; tableType : DaqCalTableTypeT)`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`tableType`	Calibration table type to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectInputSignal, daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

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

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

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

daqCalSelectInputSignal

DLL Function	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
C	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
Visual BASIC	`VBdaqCalSelectInputSignal(ByVal handle&, ByVal input as DaqCalInputT )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; input: DaqCalInputT)`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`input`	Calibration input signal source to use
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCalGetConstants, daqCalSetConstants, daqCalSelectCalTable, daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

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

daqCalSetConstants

DLL Function	`daqCalSetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, WORD gainConstant, SHORT offsetConstant );`
C	`daqCalSetConstants(DaqHandleT handle, DWORD channel, DaqAdcGain gain, DaqAdcRangeT range, WORD gainConstant, SHORT offsetConstant );`
Visual BASIC	`VBdaqCalSetConstants(ByVal handle&, ByVal channel&, ByVal gain&, ByVal range&, ByVal gainConstant%, ByVal offsetConstant% );`
Delphi	`daqCalSetConstants(handle: DaqHandleT;channel:DWORD; gain: DaqAdcGain; range: DaqAdcRangeT; gainConstant:WORD; offsetConstant:SHORT );`
Parameters	
`handle`	Handle to the device for which ADC transfer status is to be retrieved
`channel`	Channel number to apply the calibration settings
`gain`	Gain range to apply the calibration settings
`range`	A/D input range to apply the calibration settings
`gain`	Gain value to apply
`offset`	Offset value to apply
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqCalGetConstants, daqCalSelectCalTable, daqCalSelectInputSignal,daqCalSaveConstants`
Program References	None
Used With	WaveBook/512, WaveBook/516

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

daqClose

DLL Function	`daqClose(DaqHandleT handle);`
C	`daqClose(DaqHandleT handle);`
Visual BASIC	`VBdaqClose&(ByVal handle&)`
Delphi	`daqClose(handle:DaqHandleT)`
Parameters	
`handle`	Handle to the device to be closed
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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 *API Error Codes* on page 12-42)
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 *API Error Codes* on page 12-42)
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.

daqDefaultErrorHandler

DLL Function	`daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);`
C	`daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);`
Visual BASIC	`VBdaqDefaultErrorHandler(ByVal handle&, ByVal errCode&)`
Delphi	`daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device to which the default error handler is to be attached.
`ErrCode`	The error code number of the detected error (see table *API Error Codes* at end of this chapter).
Returns	Nothing (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetLastError, daqProcessError, daqSetDefaultErrorHandler`
Program References	None
Used With	All devices

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

daqFormatError

DLL Function	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
C	`daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);`
Visual BASIC	`VBdaqCalSelectInputSignal(ByVal handle&, ByVal input as DaqCalInputT )`
Delphi	`daqCalSelectInputSignal( handle: DaqHandleT; input: DaqCalInputT)`
Parameters	
`daqError`	Daq* Enhanced API error code
`msg`	Pointer to a string to return the error text
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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 12-42.

daqGetDeviceCount

DLL Function	`daqGetDeviceCount(PDWORD deviceCount);`
C	`daqGetDeviceCount(PDWORD deviceCount);`
Visual BASIC	`VBdaqGetDevice&(deviceCount&)`
Delphi	`daqGetDeviceCount( var deviceCount:DWORD )`
Parameters	
`deviceCount`	Pointer to which the device count is to be returned
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetDeviceList, daqGetDeviceProperties`
Program References	None
Used With	All devices

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

daqGetDeviceList

DLL Function	`daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);`
C	`daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);`
Visual BASIC	`VBdaqGetDeviceList(deviceList as DaqDeviceListT, deviceCount&)`
Delphi	`daqGetDeviceList( var deviceList: DaqDeviceListT; var deviceCount: DWORD)`
Parameters	
`deviceList`	Pointer to memory location to which the device list is to be returned
`deviceCount`	Number of devices returned in the device list
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetDeviceCount, daqGetDeviceProperties, daqOpen,`
Program References	None
Used With	All devices

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

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

daqGetDeviceProperties

DLL Function	`daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);`
C	`daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);`
Visual BASIC	`VBdaqGetDeviceProperties( daqName$, deviceProps as DaqDevicePropsT)`
Delphi	`daqGetDeviceProperties( daqName: string; var deviceProps: DaqDevicePropsT)`
Parameters	
`daqName`	Pointer to a character string representing the name of the device for which to retrieve properties
`deviceCount`	Number of devices returned in the device list
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetDeviceCount, daqGetDeviceList, daqOpen`
Program References	None
Used With	All devices

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

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

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

daqGetDriverVersion

DLL Function	`daqGetDriverVersion(PDWORD version);`
C	`daqGetDriverVersion(PDWORD version);`
Visual BASIC	`VBdaqGetDriverVersion&(version&)`
Delphi	`daqGetDriverVersion(var version:DWORD)`
Parameters	
`version`	Pointer to the version number of the current device driver.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetHardwareInfo`
Program References	ERREX.PAS (Delphi)
Used With	All devices

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

daqGetHardwareInfo

DLL Function	`daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);`
C	`daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);`
Visual BASIC	`VBdaqGetHardwareInfo&(ByVal handle&, ByVal whichInfo&, info As Variant)`
Delphi	`daqGetHardwareInfo(handle:DaqHandleT; whichInfo:DaqHardwareInfo; info:pointer)`
Parameters	
`handle`	Handle to the device
`whichInfo`	Specifies what type of device information to retrieve
`* info`	Pointer to the returned device information
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqGetDriverVersion, daqOpen`
Program References	DACEX.PAS, ERREX.PAS (Delphi)
Used With	All devices

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

daqGetLastError

DLL Function	`daqGetLastError(DaqHandleT handle, DaqError *errCode);`
C	`daqGetLastError(DaqHandleT handle, DaqError *errCode);`
Visual BASIC	`VBdaqGetLastError&(ByVal handle&, errCode&)`
Delphi	`daqGetLastError(handle:DaqHandleT; var errCode:DaqError): DaqError; stdcall; external DAQX_DLL; procedure daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device
`*errCode`	Returned last error code
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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.

daqIORead

DLL Function	`daqIORead(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, PDWORD value);`
C	`daqIORead(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, PDWORD value);`
Visual BASIC	`VBdaqIOReadBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, bitValue&)`
Delphi	`daqIOReadBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; var bitValue:longbool)`
Parameters	
`handle`	Handle to the device to perform the IO read
`devType`	IO Device type
`devPort`	IO port selection
`whichDevice`	IO device instance to read from
`whichExpPort`	IO device expansion port to read from
`value`	IO value read
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqIOReadBit, daqIOWrite, daqIOWriteBit`
Program References	None
Used With	All devices

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

daqIOReadBit

DLL Function	`daqIOReadBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, PBOOL bitValue);`
C	`daqIOReadBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, PBOOL bitValue);`
Visual BASIC	`VBdaqIOReadBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, bitValue&)`
Delphi	`daqIOReadBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; var bitValue:longbool)`
Parameters	
`handle`	Handle to the device from which to perform the IO
`devType`	IO Device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO expansion port address
`bitNum`	IO port bit location to read
`bitValue`	IO port bit value (`TRUE - high, FALSE - low`)
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqIORead, daqIOWrite, daqIOWriteBit`
Program References	None
Used With	All devices

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

daqIOWrite

DLL Function	`daqIOWrite(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD value);`
C	`daqIOWrite(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD value);`
Visual BASIC	`VBdaqIOWriteBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, ByVal bitValue&)`
Delphi	`daqIOWriteBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; bitValue:longbool)`
Parameters	
`handle`	Handle of the device to perform an IO write operation
`devType`	IO device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO device expansion port address
`value`	Value to write
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqIORead, daqIOWriteBit, daqIOReadBit`
Program References	None
Used With	All devices

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

daqIOWriteBit

DLL Function	`daqIOWriteBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, BOOL bitValue);`
C	`daqIOWriteBit(DaqHandleT handle, DaqIODeviceType devType, DaqIODevicePort devPort, DWORD whichDevice, DaqIOExpansionPort whichExpPort, DWORD bitNum, BOOL bitValue);`
Visual BASIC	`VBdaqIOWriteBit&(ByVal handle&, ByVal devType&, ByVal devPort&, ByVal whichDevice&, ByVal whichExpPort&, ByVal bitNum&, ByVal bitValue&)`
Delphi	`daqIOWriteBit(handle:DaqHandleT; devType:DaqIODeviceType; dvPort:DaqIODevicePort; whichDevice:DWORD; whichExpPort:DaqIOExpansionPort; bitNum:DWORD; bitValue:longbool)`
Parameters	
`handle`	Handle of the device to perform an IO write to
`devType`	IO device type
`devPort`	IO device port selection
`whichDevice`	IO device selection
`whichExpPort`	IO device expansion port address
`bitNum`	Bit number to write
`bitValue`	Bit value to write (`TRUE - high, FALSE - low`)
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqIOWrite, daqIORead, daqIOReadBit`
Program References	None
Used With	All devices

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

daqOnline

DLL Function	`daqOnline(DaqHandleT handle, PBOOL online);`
C	`daqOnline(DaqHandleT handle, PBOOL online);`
Visual BASIC	`VBdaqOnline&(ByVal handle&, online&)`
Delphi	`daqOnline(handle: DaqHandleT; var online: longbool)`
Parameters	
`handle`	Handle of the device to test for online
`online`	Boolean indicating whether the device is currently online
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqOpen, daqClose`
Program References	ERREX.PAS (Delphi)
Used With	All devices

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

daqOpen

DLL Function	`daqOpen(LPSTR daqName);`
C	`daqOpen(LPSTR daqName);`
Visual BASIC	`VBdaqOpen&(ByVal daqName$)`
Delphi	`daqOpen(devName: PChar)`
Parameters	
`daqName`	String representing the name of the device to be opened
Returns	A handle to the specified device (also, refer to *API Error Codes* on page 12-42)
See Also	`daqClose, daqOnline`
Program References	ADCEX1.C, DIGEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS, ADCEX.PAS (Delphi)
Used With	

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

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

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

daqProcessError

DLL Function	`daqProcessError(DaqHandleT handle, DaqError errCode);`
C	`daqProcessError(DaqHandleT handle, DaqError errCode);`
Visual BASIC	`VBdaqProcessError&(ByVal handle&, ByVal errCode&)`
Delphi	`daqProcessError(handle:DaqHandleT; errCode:DaqError)`
Parameters	
`handle`	Handle to the device for which the specified error is to be processed.
`errCode`	Specifies the device error code to process
Returns	Refer to *API Error Codes* on page 12-42
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 *API Error Codes* on page 12-42)
See Also	`daqDefaultErrorHandler, daqGetLastError, daqProcessError, daqSetErrorHandler`
Program References	ERREX.PAS (Delphi)
Used With	All devices

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

daqSetErrorHandler

DLL Function	`daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);`
C	`daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);`
Visual BASIC	`VBdaqSetErrorHandler&(ByVal handle&, ByVal handler&)`
Delphi	`daqSetErrorHandler(handle:DaqHandleT; handler:DaqErrorHandlerFPT)`
Parameters	
`handle`	Handle to the device to which to attach the specified error handler
`handler`	Pointer to a user defined error handler function.
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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 *API Error Codes* on page 12-42)
See Also	`daqAdcExpSetChanOption,`
Program References	None
Used With	All devices

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

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

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

daqSetTimeout

DLL Function	`daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);`
C	`daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);`
Visual BASIC	`VBdaqSetTimeout&(ByVal handle&, ByVal mSecTimeout&)`
Delphi	`daqSetTimeout(handle:DaqHandleT; mSecTimeout:DWORD)`
Parameters	
`handle`	Handle to the device for which the event time-out is to be set
`mSecTimeout`	Specifies time-out (ms) for events
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqWaitForEvent, daqWaitForEvents`
Program References	None
Used With	All devices

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

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

daqTest

DLL Function	`daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);`
C	`daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);`
Visual BASIC	`VBdaqTest&(ByVal handle&, ByVal command&, ByVal count&, cmdAvailable&, result&)`
Delphi	[not supported]
Parameters	
`handle`	Handle to the device for which the test is to be performed
`command`	Specifies the type of test to be run
`count`	Optional parameter which specifies the length of the test
`cmdAvailable`	Return Boolean indicating the availability of the test for the device
`result`	Pointer to the test result field
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	`daqOpen`
Program References	None
Used With	All devices

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

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

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

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

Performance tests measure the speed at which certain operations can be performed on the device. In general, the performance test results indicate the maximum rate at which the operation can be performed on the device. The valid performance test types are defined as follows:

- **`DtsAdcFifoInputSpeed`** - This test will determine the maximum rate at which analog input can be acquired and transferred to system memory. Analog input performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsDacFifoOutputSpeed`** - (DaqBoard only) This test will determine the maximum rate at which analog output data can be read from system memory and transferred to the device's DAC FIFO. Analog output performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsIOInputSpeed`** - This test will determine the maximum rate at which digital input can be read from the device's DIO port and transferred to system memory. Digital input performance results will be returned in the **`result`** parameter with units of bytes/second.
- **`DtsIOOutputSpeed`** - This test will determine the maximum rate at which digital output can be read from system memory and output to the device's DIO port. Digital output performance results will be returned in the **`result`** parameter with units of bytes/second.

The **`cmdAvailable`** parameter is a pointer to a Boolean value that indicates whether or not the specified test is available for the device.

The **`count`** parameter can be used to indicate the duration or length of the test. For instance, a resource test will be run **`count`** times; and if any one iteration of the test fails, it will indicate an overall failure of the test. For a performance test, the **`count`** parameter will indicate the number of times to run the test, and the test result will be an average of all the tests performed.

daqWaitForEvent

DLL Function	**`daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);`**
C	**`daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);`**
Visual BASIC	**`VBdaqWaitForEvent&(ByVal handle&, ByVal daqEvent&)`**
Delphi	**`daqWaitForEvent(handle:DaqHandleT; daqEvent:DaqTransferEvent)`**
Parameters	
`handle`	Handle of the device for which to wait of the specified event
`daqEvent`	Specifies the event to wait on
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
See Also	**`daqWaitForEvents, daqSetTimout`**
Program References	ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi)
Used With	All devices

`daqWaitForEvent` allows you to wait on a specific Daq* event to occur on the specified device. This function will not return until the specified event has occurred or the wait has timed out—whichever comes first. The event time-out can be set with the **`daqSetTimout`** function. See the *Transfer Event Definitions* table for event definitions.

daqWaitForEvents

DLL Function	**`daqWaitForEvents(DaqHandleT *handles, DaqTransferEvent *daqEvents, DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);`**
C	**`daqWaitForEvents(DaqHandleT *handles, DaqTransferEvent *daqEvents, DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);`**
Visual BASIC	**`VBdaqWaitForEvents&(handles&(), daqEvents&(), ByVal eventCount&, eventSet&(), ByVal waitMode&)`**
Delphi	**`daqWaitForEvents(handles:DaqHandlePT; daqEvents:DaqTransferEventP; eventCount:DWORD; eventSet:PLONGBOOL; waitMode:DaqWaitMode)`**
Parameters	
`*handles`	Pointer to an array of handles which represent the list of device on which to wait for the events
`*daqEvents`	Pointer to an array of events which represents the list of events to wait on
`eventCount`	Number of defined events to wait on
`*eventSet`	Pointer to an array of Booleans indicating if the events have been satisfied.
`waitMode`	Specifies the mode for the wait
Returns	`DerrNoError` - No error (also, refer to *API Error Codes* on page 12-42)
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.

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	12-39
Event-Handling Definitions	Transfer Event Definitions - **DaqTransferEvent** Transfer Event Wait Mode Definitions - **DaqWaitMode**	12-39
Hardware Information Definitions	Hardware Information Selector Definitions - **DaqHardwareInfo** Hardware Version Definitions - **DaqHardwareVersion**	12-39
ADC Trigger Source Definitions	**DaqAdcTriggerSource** **DaqEnhTrigSensT**	12-40
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**	12-40
WBK Card Definitions	WBK Option Values - **DaqChanOptionValue** WBK Channel Options - **DaqAdcExpType** WBK Module Option-Types - **DaqOptionType**	12-41
General I/O Definitions	I/O Operation Code Definitions - **DaqIOOperationCode**	12-41
DaqTest Command Definitions	**DaqTestCommand**	12-41
Calibration Input Signal Sources	**DaqCalInputT** **DaqCalTableTypeT**	12-41
API Error Codes	Identifies API errors by number and description	12-42

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

Transfer Event Definitions - `daqTransferEvent`	
`DteAdcData`	0
`DteAdcDone`	1
`DteDacData`	2
`DteDacDone`	3
`DteIOData`	4
`DteIODone`	5

Transfer Event Wait Mode Definitions - `daqWaitMode`	
`DwmNoWait`	0
`DwmWaitForAny`	1
`DwmWaitForAll`	2

Hardware Information Definitions

Hardware Information Selector Definitions - `daqHardwareInfo`

Definition	Value
`DhiHardwareVersion`	0
`DhiProtocol`	1
`DhiAdcBits`	2
`DhiADmin`	3
`DhiADmax`	4

Hardware Version Definitions - `daqHardwareVersion`

Definition	Value
`DaqBook100`	0
`DaqBook112`	1
`DaqBook120`	2
`DaqBook200`	3
`DaqBook216`	4
`DaqBoard100`	5
`DaqBoard112`	6
`DaqBoard200`	7
`DaqBoard216`	8
`Daq112`	9
`Daq216`	10
`WaveBook512`	11
`WaveBook516`	12
`TempBook66`	13

ADC Trigger Source Definitions

daqAdcTriggerSource		DaqEnhTrigSensT	
DatsImmediate	0	DetsRisingEdge	0
DatsSoftware	1	DetsFallingEdge	1
DatsAdcClock	2	DetsAboveLevel	2
DatsGatedAdcClock	3	DetsBelowLevel	3
DatsExternalTTL	4	DetsAfterRisingEdge	4
DatsHardwareAnalog	5	DetsAfterFallingEdge	5
DatsSoftwareAnalog	6	DetsAfterAboveLevel	6
DatsEnhancedTrig	7	DetsAfterBelowLevel	7

ADC Miscellaneous Definitions

ADC Flag Definitions - daqAdcFlag

Analog/High Speed Digital Flag		Unsigned/Signed ADC Data Flag		SSH Hold/Sample Flag - For Internal Use Only	
DafAnalog	00h	DafUnsigned	00h	DafSSHSample	00h
DafHighSpeedDigita	01h	DafSigned	04h	DafSSHHold	10h
Unipolar/Bipolar Flag		**Single Ended/Differential Flag**		**Clear or shift the least significant nibble - typically used with 12-bit ADCs**	
DafUnipolar	00h	DafSingleEnded	00h	DafIgnoreLSNibble	00h
DafBipolar	02h	DafDifferential	08h	DafClearLSNibble	20h
				DafShiftLSNibble	40h

Frequency vs Period - daqAdcRateMode		*ADC Acquisition Mode Definitions* - daqAdcAcqMode		*ADC Transfer Mask Definitions* - daqAdcTransferMask	
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

ADC Clock Source Definitions -daqAdcClockSource		*ADC File Open Mode Definitions* - daqAdcOpenMode		*ADC Acquisition/Transfer Active Flag Definitions* - daqAdcActiveFlag	
DacsAdcClock	0	DaomAppendFile	0	DaafAcqActive	01h
DacsGatedAdcClock	1	DaomWriteFile	1	DaafAcqTriggered	02h
DacsTriggerSource	2	DaomCreateFile	2	DaafTransferActive	04h

ADC Acquisition State - daqAdcAcqState		*ADC Buffer Transfer Mask* - daqAdcBufferXferMask	
DaasPreTrig	0	DabtmOldest	1
DaasPostTrig	1	DabtmNewest	2
		DabtmWait	3
		DabtmReturn	4

WBK Card Definitions

WBK Option Values - DaqChanOptionValue	
WBK12 Filter-Type - `WcotWbk12FilterType`	
`DcovWbk12FilterElliptic`	0
`DcovWbk12FilterLinear`	1
WBK12 Filter-Mode - `WcotWbk12FilterMode`	
`DcovWbk12FilterBypass`	0
`DcovWbk12FilterOn`	1
WBK12 Anti-Aliasing Filter-Mode-`WcotWbk12PreFilterMode`	
`DcovWbk12PreFilterDefault`	0
`DcovWbk12PreFilterOff`	1
WBK13 Filter-Type - `WcotWbk13FilterType`	
`DcovWbk13FilterElliptic`	0
`DcovWbk13FilterLinear`	1
WBK13 Filter-Mode - `WcotWbk13FilterMode`	
`DcovWbk13FilterBypass`	0
`DcovWbk13FilterOn`	1
WBK13 Anti-Aliasing Filter-Mode- `WcotWbk13PreFilterMode`	
`DcovWbk13PreFilterDefault`	0
`DcovWbk13PreFilterOff`	1
WBK14 Current-Source - `WcotWbk14CurrentSrc`	
`DcovWbk14CurrentSrcOff`	0
`DcovWbk14CurrentSrc2mA`	1
`DcovWbk14CurrentSrc4mA`	2
WBK14 High-Pass Filter - `WcotWbk14HighPassCutOff`	
`DcovWbk14HighPass0_1Hz`	0
`DcovWbk14HighPass10Hz`	1
WBK14 Low-Pass Filter-Mode - `WcotWbk14LowPassMode`	
`DcovWbk14LowPassBypass`	0
`DcovWbk14LowPassOn`	1
WBK-14 Low-Pass Filter-Mode - `WcotWbk14PreFilterMode`	
`DcovWbk14PreFilterDefault`	0
`DcovWbk14PreFilterOff`	1
WBK14 Excitation-Source Waveform - `WmotWbk14ExcSrcWaveform`	
`DmovWbk14ExcSrcRandom`	0
`DmovWbk14ExcSrcSine`	1

WBK Channel Options - DaqAdcExpType	
`DoctWbk11`	6
`DoctWbk12`	7
`DoctWbk13`	8
`DmctWbk512`	9
`DmctWbk10`	10
`DmctWbk14`	11
`DmctWbk15`	12
`DmctResponseDac`	*13
*Response DAC on WaveBook	

WBK Module Option-Types - `DaqOptionType`	
`DcotWbk12FilterCutOff`	0
`DcotWbk12FilterType`	1
`DcotWbk12FilterMode`	2
`DcotWbk12PreFilterMode`	3
`DcotWbk13FilterCutOff`	0
`DcotWbk13FilterType`	1
`DcotWbk13FilterMode`	2
`DcotWbk13PreFilterMode`	3
`DcotWbk14LowPassMode`	0
`DcotWbk14LowPassCutOff`	1
`DcotWbk14HighPassCutOff`	2
`DcotWbk14CurrentSrc`	3
`DcotWbk14PreFilterMode`	4
`DmotWbk14ExcSrcWaveform`	5
`DmotWbk14ExcSrcFreq`	6
`DmotWbk14ExcSrcAmplitude`	7
`DmotWbk14ExcSrcOffset`	8

General I/O Definitions

I/O Operation Code Definitions - `daqIOOperationCode`	
`DioocReadByte`	0
`DioocWriteByte`	1
`DioocReadWord`	2
`DioocWriteWord`	3
`DioocReadDWord`	4
`DioocWriteDWord`	5

`daqTest` Command Definitions

`DaqTestCommand`	
`DtstBaseAddressValid`	0
`DtstInterruptLevelValid`	1
`DtstDmaChannelValid`	2
`DtstAdcFifoInputSpeed`	3
`DtstDacFifoOutputSpeed`	4
`DtstIOInputSpeed`	5
`DtstIOOutputSpeed`	6

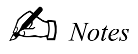
Calibration Input Signal Sources

`DaqCalInputT`		
`DciNormal`	0	External signal from device input connector(s)
`DciCalGround`	1	Internal calibration ground signal
`DciCal5V`	2	Internal 5 V calibration signal
`DciCal500mV`	3	Internal 500 mV calibration signal
`DaqCalTableTypeT`		
`DcttFactory`	0	Factory calibration constants
`DcttUser`	1	User-defined calibration constants

API Error Codes

Error Name	Code # hex - dec	Description
DerrNoError	00h - 0	No error
DerrBadChannel	01h - 1	Specified LPT channel was out-of-range
DerrNotOnLine	02h - 2	Requested device is not online
DerrNoDaqbook	03h - 3	DaqBook is not on the requested channel
DerrBadAddress	04h - 4	Bad function address
DerrFIFOFull	05h - 5	FIFO Full detected, possible data corruption
DerrBadDma	06h - 6	Bad or illegal DMA channel or mode specified for device
DerrBadInterrupt	07h - 7	Bad or illegal INTERRUPT level specified for device
DerrDmaBusy	08h - 8	DMA is currently being used
DerrInvChan	10h - 16	Invalid analog input channel
DerrInvCount	11h - 17	Invalid count parameter
DerrInvTrigSource	12h - 18	Invalid trigger source parameter
DerrInvLevel	13h - 19	Invalid trigger level parameter
DerrInvGain	14h - 20	Invalid channel gain parameter
DerrInvDacVal	15h - 21	Invalid DAC output parameter
DerrInvExpCard	16h - 22	Invalid expansion card parameter
DerrInvPort	17h - 23	Invalid port parameter
DerrInvChip	18h - 24	Invalid chip parameter
DerrInvDigVal	19h - 25	Invalid digital output parameter
DerrInvBitNum	1Ah - 26	Invalid bit number parameter
DerrInvClock	1Bh - 27	Invalid clock parameter
DerrInvTod	1Ch - 28	Invalid time-of-day parameter
DerrInvCtrNum	1Dh - 29	Invalid counter number
DerrInvCntSource	1Eh - 30	Invalid counter source parameter
DerrInvCtrCmd	1Fh - 31	Invalid counter command parameter
DerrInvGateCtrl	20h - 32	Invalid gate control parameter
DerrInvOutputCtrl	21h - 33	Invalid output control parameter
DerrInvInterval	22h - 34	Invalid interval parameter
DerrTypeConflict	23h - 35	An integer was passed to a function requiring a character
DerrMultBackXfer	24h - 36	A second background transfer was requested
DerrInvDiv	25h - 37	Invalid Fout divisor
Temperature Conversion Errors		
DerrTCE_TYPE	26h - 38	TC type out-of-range
DerrTCE_TRANGE	27h - 39	Temperature out-of-CJC-range
DerrTCE_VRANGE	28h - 40	Voltage out-of-TC-range
DerrTCE_PARAM	29h - 41	Unspecified parameter value error
DerrTCE_NOSETUP	2Ah - 42	dacTCConvert called before dacTCSetup
DaqBook		
DerrNotCapable	2Bh - 43	Device is incapable of function
Background		
DerrOverrun	2Ch - 44	A buffer overrun occurred
Zero and Cal Conversion Errors		
DerrZCInvParam	2Dh - 45	Unspecified parameter value error
DerrZCNoSetup	2Eh - 46	dac…Convert called before dac…Setup
DerrInvCalFile	2Fh - 47	Cannot open the specified cal file
Environmental Errors		
DerrMemLock	30h - 48	Cannot lock allocated memory from operating system
DerrMemHandle	31h - 49	Cannot get a memory handle from operating system
Pre-trigger acquisition Errors		
DerrNoPreTActive	32h - 50	No pre-trigger configured
Daq FIFO Errors (DaqBoard only)		
DerrInvDacChan	33h - 51	DAC channel does not exist
DerrInvDacParam	34h - 52	DAC parameter is invalid
DerrInvBuf	35h - 53	Buffer points to NULL or buffer size is zero
DerrMemAlloc	36h - 54	Could not allocate the needed memory
DerrUpdateRate	37h - 55	Could not achieve the specified update rate
DerrInvDacWave	38h - 56	Could not start waveforms because of missing or invalid parameters
DerrInvBackDac	39h - 57	Could not start waveforms with background transfers
DerrInvPredWave	3Ah - 58	Predefined waveform not supported
RTD Conversion Errors		
DerrRtdValue	3Bh - 59	rtdValue out-of-range
DerrRtdNoSetup	3Ch - 60	rtdConvert called before rtdSetup

Error Name	Code # hex - dec	Description
`DerrRtdArraySize`	`3Dh - 61`	Temperature array not large enough
`DerrRtdParam`	`3Eh - 62`	Incorrect RTD parameter
`DerrInvBankType`	`3Fh - 63`	Invalid bank-type specified
`DerrBankBoundary`	`40h - 64`	Simultaneous writes to DBK cards in different banks not allowed
`DerrInvFreq`	`41h - 65`	Invalid scan frequency specified
`DerrNoDaq`	`42h - 66`	No Daq112B/216B installed
`DerrInvOptionType`	`43h - 67`	Invalid option-type parameter
`DerrInvOptionValue`	`44h - 68`	Invalid option-value parameter
New API Error Codes		
`DerrTooManyHandles`	`60h - 96`	No more handles available to open
`DerrInvLockMask`	`61h - 97`	Only a part of the resource is already locked, must be all or none
`DerrAlreadyLocked`	`62h - 98`	All or part of the resource was locked by another application
`DerrAcqArmed`	`63h - 99`	Operation not available while an acquisition is armed
`DerrParamConflict`	`64h - 100`	Each parameter is valid, but the combination is invalid
`DerrInvMode`	`65h - 101`	Invalid acquisition/wait/dac mode
`DerrInvOpenMode`	`66h - 102`	Invalid file-open mode
`DerrFileOpenError`	`67h - 103`	Unable to open file
`DerrFileWriteError`	`68h - 104`	Unable to write file
`DerrFileReadError`	`69h - 105`	Unable to read file
`DerrInvClockSource`	`6Ah - 106`	Invalid acquisition mode
`DerrInvEvent`	`6Bh - 107`	Invalid transfer event
`DerrTimeout`	`6Ch - 108`	Time-out on wait
`DerrInitFailure`	`6Dh - 109`	Unexpected result occurred while initializing the hardware
`DerrBufTooSmall`	`6Eh - 110`	Unexpected result occurred while initializing the hardware
`DerrInvType`	`6Fh - 111`	Invalid acquisition/wait/dac mode
`DerrArraySize`	`70h - 112`	Used as a catch all for arrays not large enough
`DerrBadAlias`	`71h - 113`	Invalid alias names for Vxd lookup
`DerrInvCommand`	`72h - 114`	Invalid command
`DerrInvHandle`	`73h - 115`	Invalid handle
`DerrNoTransferActive`	`74h - 116`	Transfer not active
`DerrNoAcqActive`	`75h - 117`	Acquisition not active
`DerrInvOpstr`	`76h - 118`	Invalid operation string used for enhanced triggering
`DerrDspCommFailure`	`77h - 119`	Device with DSP failed communication
`DerrEepromCommFailure`	`78h - 120`	Device with EEPROM failed communication
`DerrInvEnhTrig`	`79h - 121`	Device using enhanced trigger detected invalid trigger type
`DerrInvCalConstant`	`7Ah - 122`	User calibration constant out of range
`DerrInvErrorCode`	`7Bh - 123`	Invalid error code
`DerrInvAdcRange`	`7Ch - 124`	Invalid analog input voltage range parameter
`DerrInvCalTableType`	`7Dh - 125`	Invalid calibration table type
`DerrInvCalInput`	`7Eh - 126`	Invalid calibration input signal selection
`DerrInvRawDataFormat`	`7Fh - 127`	Invalid raw-data format selection
`DerrNotImplemented`	`80h - 128`	Feature/function not implemented yet
`DerrInvDioDeviceType`	`81h - 129`	Invalid digital I/O device type
`DerrInvPostDataFormat`	`82h - 130`	Invalid post-processing data format selection



Notes

Accessories and Specifications

Available Accessories

The WaveBook/512 includes the CA-140 cable and a TR-40U power supply (the TR-27 power supply is optional).

The WBK10 includes the CA-128, & CA-129 cables and a TR-40U power supply (the TR-27 power supply is optional).

Related accessories and options include:

CA-115	Daisy-chain power cable (DIN5, male-to-male) - 6 in.
CA-116	DIN5 to automobile cigarette lighter power cable - 8 ft
CA-128	Analog expansion signal cable, 12-inch, BNC-F to BNC-F
CA-129	Analog expansion control cable, 15-inch, HD15M to HD15F
CA-140	Cable, WaveBook/512 to PC
CA-150-1	1, BNC-M to BNC-M (CE-compliant cable)
CA-150-8	8, BNC-M to BNC-M (CE-compliant cable)

DBK30A	Rechargeable battery module

HA-111	Fastener-panel handle

TR-40U	Power supply: 90 to 264 VAC to 15 VDC @ 2.5 A
TR-27	Power supply: 115 VAC to 18 VDC @ 0.8 A

WBK10	8-channel Expansion Chassis
WBK11	Simultaneous Sample & Hold (SSH) Card
WBK12	Programmable Low-Pass Filter Card
WBK13	Programmable Low-Pass Filter Card with SSH
WBK14	Dynamic Signal Conditioning Module
WBK15	8-Slot 5B Signal Conditioning Module
WBK20	PCMCIA card & cable for laptop PC use
WBK21	ISA bus card for desktop PC use
WBK61/62	High-Voltage Adapter and Probes

Note: New WBK options regularly become available for new applications. Call your sales agent for the availability of new WBK products. Also, many different kinds of 5B modules are available for the WBK15.

Specifications

The following pages include specifications for the WaveBook/512 and related products (DBK30A, WBK10, WBK11, WBK12, WBK13, WBK14, WBK15, WBK20, WBK21, and WBK61/62).

WaveBook/512 - Specifications

General

Power Consumption: 12 to 17 Watts typ
Input Power Range: 10 to 30 VDC
Operating Temperature: 0 to 50°C
Storage Temperature: 0 to 70°C
Humidity: 0 to 95% RH, non-condensing
Size: 220 mm wide × 285 mm long × 35 mm high (8.5" × 11" × 1.375")
Weight: 1.5 kg (3.3 lb)

Analog Inputs

Channels: 8 differential, expandable up to 72 differential
Connector: BNC
Resolution: 12 bit
Input Ranges:
Unipolar:
0 to +10 V (x1 gain)
0 to +5 V (x2 gain)
0 to +2 V (x5 gain)
0 to +1 V (x10 gain)
Bipolar:
0 to ±5.0 V (x1 gain)
0 to ±2.5 V (x2 gain)
0 to ±1.0 V (x5 gain)
0 to ±0.5 V (x10 gain)
Accuracy: 0.025% FS max using external or factory calibration
Sampling Rate: 1 MHz (1 μs)
Sequencer: 1 to 128 step channel/gain
Common mode rejection: >70 dB from 0 to 100 Hz
Maximum Overvoltage:
±30 VDC to Analog Common
±45 VDC between channels
Input Current:
50 nA typ
500 nA max
Input Impedance:
Single-ended: 5 MΩ in parallel with 30 pF
Differential: 10 MΩ in parallel with 30 pF

Triggering

Analog Trigger:
12 bit resolution from -5 V to +10 VDC
Trigger to A/D Latency: 300 nanoseconds max
External TTL Trigger:
Logic Level Range: 0.8 V low/2.2 V high
Trigger to A/D Latency: 200 nanoseconds max
Software Trigger:
Trigger to A/D Latency: 100 μs typ

Sequencer

Arbitrarily programmable for channel & gain; and for unipolar/bipolar ranges
Depth: 128 location
Channel to channel rate: 1 μs/channel, fixed
Maximum repeat rate: 1 MHz
Minimum repeat rate: 100 seconds per scan
Expansion channel sample rate: Same as on-board channels (1 μs/channel)
Pre-trigger duration: 0 - 100,000,000 scans
Post-trigger duration: 1 - 100,000,000 scans or infinite

Digital I/O Connector

Connector: DB25F
Capacity: 8 input/output signals; 5 address signals; read/write/enable signals
Sampling rate: 1 Mbyte/second max
Signal levels: TTL, 0.8 V low, 2.0 V high
Termination: 10 KΩ to +5 V
Power Connections: +5 V @ 250 mA max; ±15 V @ 50 mA max

DBK30A - Specifications

Name/Function: Rechargeable Battery Module
Battery Type: Nickel-cadmium
Number of Battery Packs: 2
Battery Pack Configuration: 12 series-connected sub-C cells
Output Voltage: 14.4 V or 28.8 V (depending on the selected mode)
Output Fuses: 2 A
Battery Amp-Hours: 3.4 A-hr (1.7 A-hr/pack)
Charge Termination: Peak detection
Charge Time: 2 hours
Charging Voltage from Supplied AC Adapter: 22 to 26 VDC @ 2 A
AC Adapter Input: 95 to 265 VAC @ 47 to 63 Hz
Size: 221 mm × 285 mm × 35 mm (11" × 8-1/2" × 1-3/8")
Weight: 2.4 kg (6 lb)

WBK10 - Specifications

Name/Function: WBK10 8-Channel Analog Expansion Module
Number of Channels: 8 differential
Connector: BNC
Accuracy: ±0.025% FS
Offset: ±1 LSB max
Maximum Overvoltage: 30 VDC
Ranges: Unipolar/Bipolar operation is software selectable via sequencer
Unipolar: 0 to +10 V, 0 to +5 V, 0 to +2 V, 0 to +1 V
Bipolar: -5 to +5 V, -2.5 to +2.5 V, -1 to +1 V, -0.5 to +0.5 V
Input Current: 50 nA typ, 500 nA max
Input Impedance
Single-ended: 5 MΩ in parallel 30 pF
Differential: 10 MΩ in parallel 30 pF
Gain Temperature Coefficient: 5 ppm/°C typ
Offset Temperature Coefficient: 12 uV/°C max
Sampling Rate: 1 MHz (1 µs)
Common mode rejection: >70 dB from 0 to 100 Hz
Power: 0.6 A max @ 15 VDC (7 to 12 Watts typ)
Operating Temperature: 0 to 50°C
Storage Temperature: 0 to 70°C
Humidity: 0 to 95% RH, non-condensing
Dimensions: 220 mm wide × 285 mm long × 35 mm high (8.5” × 11” × 1.375”)
Weight: 1.3 kg (2.8 lb)

WBK11 - Specifications

Name/Function: WBK11 8-Channel Simultaneous Sample-and-Hold Card
Number of Channels: 8
Connectors: Internal to the WaveBook/512 (36-pin sockets mate with 36-pin connectors)
Accuracy: ±0.025% FS
Offset: ±1 LSB max
Aperture Uncertainty: 75 ps max
Voltage Droop: 0.1 mV/ms max
Maximum Signal Voltage: ±5.00 VDC (×1)
Input Voltage Ranges: Software programmable prior to a scan sequence; expands WaveBook/512 ranges to:
Unipolar: 0 to +10 V, 0 to +5 V, 0 to +2 V, 0 to +1 V, 0 to +0.5 V
Bipolar: -5 to +5 V, -2.5 to +2.5 V, -1 to +1 V, -0.5 to +0.5 V, -0.05 to +0.05 V
Programmable Gain Amplifier Gain Ranges: ×1, 2, 5, 10, 20, 50, 100
Weight: 0.14 kg (0.3 lb)

WBK12/13 - Specifications

Name/Function: WBK12, Programmable Low-Pass Filter Card
WBK13, Programmable Low-Pass Filter Card With SSH
Number of Channels: 8
Connector: Internal to WaveBook/512 and WBK10 (two 36-pin sockets mate with 36-pin connectors)
Input Voltage Ranges: Software programmable prior to a scan sequence

Unipolar		Bipolar	
Voltage Range	Gain	Voltage Range	Gain
0 to +0.1 V	×100	±0.05 V	×100
0 to +0.2 V	×50	±0.1 V	×50
0 to +0.5 V	×20	±0.25 V	×20
0 to +1 V	×10	±0.5 V	×10
0 to +2 V	×5	±1 V	×5
0 to +5 V	×2	±2.5 V	×2
0 to +10 V	×1	±5 V	×1

Programmable Gain Amplifier Ranges: ×1, 2, 5, 10, 20, 50, and 100
Switched Capacitor Filter Cutoff Frequencies Range: 400 Hz to 100 kHz
Number of Cutoff Frequencies: 1024
Filter Grouping: 4 channels each in 2 programmable banks
Low-Pass Filter: Software selectable, 8-pole elliptic filter
Low-Pass Filter Type: Software selectable, elliptic or linear phase
Low-Pass Filter Frequency Cutoff Range: 100 kHz, 75 kHz, 60 kHz...400 Hz, bypass defined as Fc = 300 kHz/N where N = 3 to 750
Anti-Alias Frequencies: determined by software control
Accuracy: ±0.05% FS DC
Offset: ±1 LSB max
Aperture Uncertainty: 75 ps max
Voltage Droop: 1 mV/ms max (0.01 mV/ms typ)
Maximum Signal Voltage: ±5.00 VDC (×1)
THD: -65 dB (-70 dB typ)
Noise: 3 counts (RMS)
DC Offset: ±2.5 mV (2 LSB) max at any cutoff frequency
Number of Cutoff Frequencies Simultaneously Set: 2, one for each 4-channel bank of inputs
Weight: 0.14 kg (0.3 lb)

WBK14 - Specifications

Name/Function: **WBK14, 8-Channel Dynamic Signal Conditioning Module**
Connectors: BNC connector, mates with expansion signal input on the WaveBook/512; two 15-pin connectors, mate with expansion signal control on the WaveBook/512; signals via 1 BNC per channel
Channels: 8
Gain Ranges: ×1, 2, 5, 10, 20, 50, 100, 200
Power Consumption: 15 Watts typical
Input Power Range: 10 to 30 VDC
Operating Temperature: 0°C to 50°C
Storage Temperature: 0°C to 70°C
Dimensions: 216 mm wide × 279 mm long × 35 mm high (8.5” × 11” × 1.375”)
Weight: 1.32 kg (2.9 lb)
ICP Current Source:
Output Impedance: > 1.0 MΩ @ 20 kHz
Compliance: 27 V
Current Levels: 2 & 4 mA
Coupling: AC
10 Hz High-Pass Filter - Input Impedance: 590K
0.1 Hz High-Pass Filter - Input Impedance: 10 MΩ
Input Ranges:
±5.0 V, ±2.5 V, ±1.0 V, ±500 mV, ±250 mV, ±100 mV, ±50 mV, ±25 mV

Anti-Aliasing Low-Pass Filter:
Accuracy: ±0.5 dB at the passband center
Frequency Span: 30 Hz to 100 kHz
Frequency Settings: 300 kHz / N; N = 3,4,...10000
Dynamic Range @ 1 kHz: 69 dB
THD @ 1 kHz: 70 db
Amplitude Matching: ± 0.1 dB
Phase Matching: ± 2°
Excitation Source:
Max. Output Voltage: ± 10 V
Max. Output Current: 10 mA
DC Output: ± 5 V
Sine:
Frequency: 20 Hz - 100 kHz
Distortion: < 0.1%
Amplitude: ± 5 V
Steps: 256
Random:
Spectral Distribution: White, Band-limited
Amplitude Distribution: Gaussian
Bandwidth: 20 Hz - 100 kHz
RMS level: Adjustable in binary steps
External Clock:
Digital: TTL levels
Sine: > 500 mV peak

WBK15 - Specifications

Name/Function: WBK15 Multi-Purpose Isolated Signal Conditioning Module
Connector: 2 BNC connectors, mate with expansion signal input on the WaveBook/512; two 15-pin connectors, mate with expansion signal control on the WaveBook/512
Module Capacity: Eight 5B modules (optional)
Input Connections: Removable 4-terminal plugs
(Weidmuller type BL4, PN 12593.6 or type BLTOP4, PN 13360.6)
Power Requirements: 10 to 30 VDC or 120 VAC with included adapter
With 8 thermocouple-type modules: 12 VDC @ 0.25 A, 15 VDC @ 0.20 A, 18 VDC @ 0.2 A
With 8 strain-gage-type modules: 12 VDC @ 0.95 A, 15 VDC @ 0.75 A, 18 VDC @ 0.65 A
Cold-Junction Sensor: Standard per channel
Shunt-Resistor Socket: One per channel for current loop inputs
Isolation
Signal Inputs to System: 1500 VDC (600 VDC for CE compliance)
Input Channel-to-Channel: 1500 VDC (600 VDC for CE compliance)
Power Supply to System: 50 VDC
Dimensions: 221 mm × 285 mm × 36 mm (8.5” × 11” × 1.375”)
Weight: 1.8 kg (4 lb) with no modules installed

WBK20 - Specifications

Name/Function: WBK20 PCMCIA/EPP Interface Card
Bus Interface: 8-bit PCMCIA Card Standard 2.1
Dimensions: 5 mm (PCMCIA Type II) card
Connector: DB25F
Transfer Rate: > 2 Mbytes/s
Cable: 2 ft (included)

WBK21 - Specifications

Name/Function: WBK21 ISA/EPP Interface Plug-in Board
Bus Interface: 16-bit ISA-bus interface
Transfer Rate: > 2.5 Mbytes/s
LPT Address: 378 or 278
LPT Interrupts: 5 or 7
Connector: DB25F
Serial-Port: high-speed 16C550 via DB9
Serial-Port Address: 3F8, 2F8, 3E8, or 2E8
Serial-Port Interrupt: 2, 3, 4, or 5
Connector: DB9M

WBK61/62 - Specifications

Name/Function:
WBK61, High-Voltage Adapter with Probes, 200:1 Voltage Divider
WBK62, High-Voltage Adapter with Probes, 20:1 Voltage Divider
Number of Channels: 1
Dimensions: 83 mm × 61 mm × 28 mm (3.25” × 2.375” × 1.1”)
Cables: 60” leads with detachable probe tips and alligator clips
Output Connector: BNC female
Voltage Divider:
WBK61: 200:1 fixed
WBK62: 20:1 fixed
Maximum Voltage
WBK61: 1000 Vpeak (on either input reference to earth ground)
WBK62: 100 Vpeak (on either input reference to earth ground)
Maximum Differential Voltage:
WBK61: 2000 Vpeak (if neither input exceeds 1000 Vp rating to earth ground)
WBK62: 200 Vpeak (if neither input exceeds 100 Vp rating to earth ground)
Frequency Characteristics: approximates a single-pole frequency response
-3 dB Bandwidth: 200 kHz minimum

Voltage Ranges: * Note: The asterisk indicates the range is obtained with the use of a WBK11, WBK12, or WBK13.
WBK61 Effective Ranges:
Unipolar: +1000V, 500V, 200V, 100V*, 40V*, 20V*
Bipolar: ±1000V, 500V, 200V, 100V, 50V*, 20V*, 10V*
WBK62 Effective Ranges: Note: For the WBK62 ranges which are followed by an asterisk, the WaveBook/512 (or WBK10) will exhibit superior performance with no WBK62 present.
Unipolar: +100V, 50V, 20V, 10V*, 4V*, 2V*
Bipolar: ±100V, 50V, 20V, 10V, 5V*, 2V*, 1V*
Measurement Errors:
The following values include total system error, i.e., they include errors from WaveBook/512, WBK10, WBK11, WBK12, and WBK13. The value for gain error does not include offset error.
Gain Error:
0.1% FS (unipolar)
0.2% FS (bipolar)
Offset Error:
0.1% FS (unipolar)
0.2% FS (bipolar)

Notes

WARRANTY/DISCLAIMER

OMEGA ENGINEERING, INC. warrants this unit to be free of defects in materials and workmanship for a period of **13 months** from date of purchase. OMEGA Warranty adds an additional one (1) month grace period to the normal **one (1) year product warranty** to cover handling and shipping time. This ensures that OMEGA's customers receive maximum coverage on each product.

If the unit should malfunction, it must be returned to the factory for evaluation. OMEGA's Customer Service Department will issue an Authorized Return (AR) number immediately upon phone or written request. Upon examination by OMEGA, if the unit is found to be defective it will be repaired or replaced at no charge. OMEGA's WARRANTY does not apply to defects resulting from any action of the purchaser, including but not limited to mishandling, improper interfacing, operation outside of design limits, improper repair, or unauthorized modification. This WARRANTY is VOID if the unit shows evidence of having been tampered with or shows evidence of being damaged as a result of excessive corrosion; or current, heat, moisture or vibration; improper specification; misapplication; misuse or other operating conditions outside of OMEGA's control. Components which wear are not warranted, including but not limited to contact points, fuses, and triacs.

OMEGA is pleased to offer suggestions on the use of its various products. However, OMEGA neither assumes responsibility for any omissions or errors nor assumes liability for any damages that result from the use of its products in accordance with information provided by OMEGA, either verbal or written. OMEGA warrants only that the parts manufactured by it will be as specified and free of defects. OMEGA MAKES NO OTHER WARRANTIES OR REPRESENTATIONS OF ANY KIND WHATSOEVER, EXPRESSED OR IMPLIED, EXCEPT THAT OF TITLE, AND ALL IMPLIED WARRANTIES INCLUDING ANY WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE HEREBY DISCLAIMED. LIMITATION OF LIABILITY: The remedies of purchaser set forth herein are exclusive and the total liability of OMEGA with respect to this order, whether based on contract, warranty, negligence, indemnification, strict liability or otherwise, shall not exceed the purchase price of the component upon which liability is based. In no event shall OMEGA be liable for consequential, incidental or special damages.

CONDITIONS: Equipment sold by OMEGA is not intended to be used, nor shall it be used: (1) as a "Basic Component" under 10 CFR 21 (NRC), used in or with any nuclear installation or activity; or (2) in medical applications or used on humans. Should any Product(s) be used in or with any nuclear installation or activity, medical application, used on humans, or misused in any way, OMEGA assumes no responsibility as set forth in our basic WARRANTY/DISCLAIMER language, and additionally, purchaser will indemnify OMEGA and hold OMEGA harmless from any liability or damage whatsoever arising out of the use of the Product(s) in such a manner.

RETURN REQUESTS/INQUIRIES

Direct all warranty and repair requests/inquiries to the OMEGA Customer Service Department. BEFORE RETURNING ANY PRODUCT(S) TO OMEGA, PURCHASER MUST OBTAIN AN AUTHORIZED RETURN (AR) NUMBER FROM OMEGA'S CUSTOMER SERVICE DEPARTMENT (IN ORDER TO AVOID PROCESSING DELAYS). The assigned AR number should then be marked on the outside of the return package and on any correspondence.

The purchaser is responsible for shipping charges, freight, insurance and proper packaging to prevent breakage in transit.

FOR **WARRANTY** RETURNS, please have the following information available BEFORE contacting OMEGA:

1. P.O. number under which the product was PURCHASED,
2. Model and serial number of the product under warranty, and
3. Repair instructions and/or specific problems relative to the product.

FOR **NON-WARRANTY** REPAIRS, consult OMEGA for current repair charges. Have the following information available BEFORE contacting OMEGA:

1. P.O. number to cover the COST of the repair,
2. Model and serial number of the product, and
3. Repair instructions and/or specific problems relative to the product.

OMEGA's policy is to make running changes, not model changes, whenever an improvement is possible. This affords our customers the latest in technology and engineering.

OMEGA is a registered trademark of OMEGA ENGINEERING, INC.

© Copyright 1996 OMEGA ENGINEERING, INC. All rights reserved. This document may not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without prior written consent of OMEGA ENGINEERING, INC.

Where Do I Find Everything I Need for Process Measurement and Control? OMEGA...Of Course!

TEMPERATURE

- ☑ Thermocouple, RTD & Thermistor Probes, Connectors, Panels & Assemblies
- ☑ Wire: Thermocouple, RTD & Thermistor
- ☑ Calibrators & Ice Point References
- ☑ Recorders, Controllers & Process Monitors
- ☑ Infrared Pyrometers

PRESSURE, STRAIN AND FORCE

- ☑ Transducers & Strain Gauges
- ☑ Load Cells & Pressure Gauges
- ☑ Displacement Transducers
- ☑ Instrumentation & Accessories

FLOW/LEVEL

- ☑ Rotameters, Gas Mass Flowmeters & Flow Computers
- ☑ Air Velocity Indicators
- ☑ Turbine/Paddlewheel Systems
- ☑ Totalizers & Batch Controllers

pH/CONDUCTIVITY

- ☑ pH Electrodes, Testers & Accessories
- ☑ Benchtop/Laboratory Meters
- ☑ Controllers, Calibrators, Simulators & Pumps
- ☑ Industrial pH & Conductivity Equipment

DATA ACQUISITION

- ☑ Data Acquisition & Engineering Software
- ☑ Communications-Based Acquisition Systems
- ☑ Plug-in Cards for Apple, IBM & Compatibles
- ☑ Datalogging Systems
- ☑ Recorders, Printers & Plotters

HEATERS

- ☑ Heating Cable
- ☑ Cartridge & Strip Heaters
- ☑ Immersion & Band Heaters
- ☑ Flexible Heaters
- ☑ Laboratory Heaters

ENVIRONMENTAL MONITORING AND CONTROL

- ☑ Metering & Control Instrumentation
- ☑ Refractometers
- ☑ Pumps & Tubing
- ☑ Air, Soil & Water Monitors
- ☑ Industrial Water & Wastewater Treatment
- ☑ pH, Conductivity & Dissolved Oxygen Instruments