# Programmer's Guide 5

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

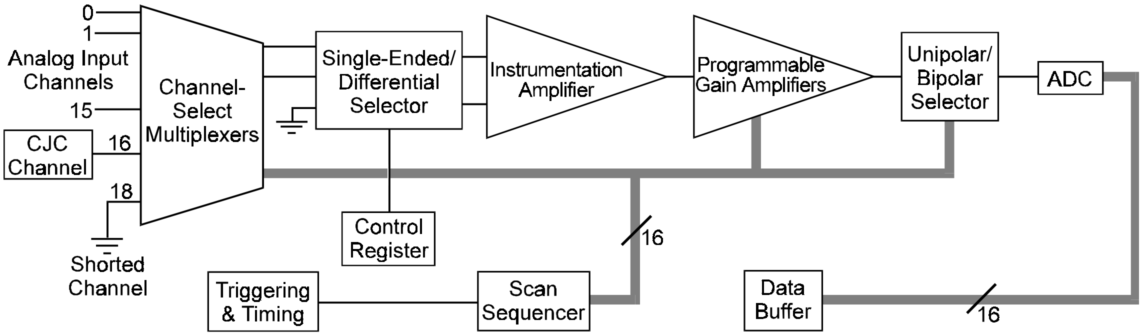
## A Programmer's View of TempBook Operations

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

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

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

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


0
1
Analog Input Channels
15
CJC Channel
16
18
Shorted Channel
Channel-Select Multiplexers
Single-Ended/ Differential Selector
Instrumentation Amplifier
Programmable Gain Amplifiers
Unipolar/ Bipolar Selector
ADC
Control Register
16
Triggering & Timing
Scan Sequencer
Data Buffer
16

*Block Diagram*

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

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

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

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

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

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

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

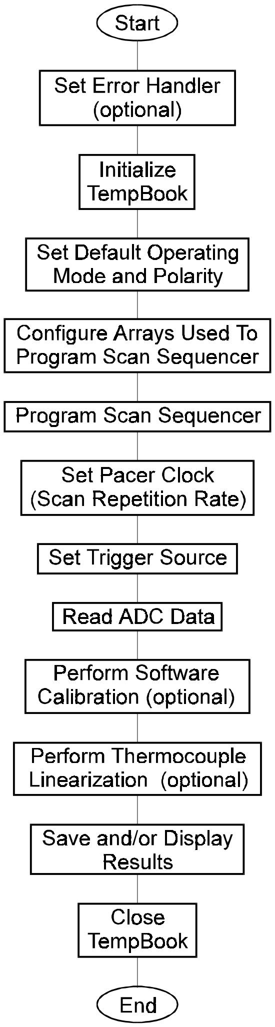
In addition to the 16 externally available analog input channels, two internal channels are provided for offset correction and thermocouple cold junction compensation. The external analog input channels are addressed as channels 0 - 15, the CJC channel is at address 16, and the internal shorted channel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

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

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

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

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


Start
Set Error Handler (optional)
Initialize TempBook
Set Default Operating Mode and Polarity
Configure Arrays Used To Program Scan Sequencer
Program Scan Sequencer
Set Pacer Clock (Scan Repetition Rate)
Set Trigger Source
Read ADC Data
Perform Software Calibration (optional)
Perform Thermocouple Linearization (optional)
Save and/or Display Results
Close TempBook
End

*Data Acquisition Program Flowchart*

# *Driver Options*

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

TempBook applications can be written to either the 16-bit TempBook API or to the 32-bit Daq* API. 16-bit API functions have the **`tbk`**… prefix. The 32-bit API is a new format which can be generically used with the TempBook, WaveBook, DaqBook, DaqBoard, and Daq PC-Card product lines. 32-bit API functions share the **`daq`**… prefix. Generally,

- If starting with an existing TempBook Application written to Windows 3.1, the quickest port is to use or re-write code to the 16-bit API.
- If writing a new application, it is best to write code to the 32-bit API due to its improved performance and enhanced feature set (see following).

## 16-Bit API (`tbk...`)

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

- When developing a new or existing DOS application
- When developing a new or existing Windows 3.1 application

## 32-Bit API (`daq...`)

The API for 32-bit systems has several features that are not present in the 16-bit 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 95/98/Me/NT/2000.

Because of these new features and other improvements, we recommend you use the 32-bit API whenever feasible. Use the 32-bit API:

- When developing new or existing Windows 95/98/Me/NT/2000 applications
- When developing new or existing Windows NT applications
- When porting an existing 16-bit API application to 32-bit mode to take advantage of the enhanced API features.

## Language Support

The following table shows language support for the 16-bit and 32-bit API drivers.

| 16-Bit API Supported Languages | 32-Bit API Supported Languages |
|---|---|
| C/C++<br>Microsoft Visual C++<br>Borland C++ (v4.0 and greater) | C/C++<br>Microsoft Visual C++ (v2.2 and greater)<br>Borland C++ (v4.0 and greater) |
| BASIC<br>Microsoft Visual Basic (v4.0 and previous)<br>QuickBASIC | BASIC<br>Microsoft Visual Basic (v4.0 and greater) |
| Pascal<br>Turbo Pascal | Delphi<br>Borland Delphi (v2.0) |

Notes

# 16-Bit API Programming of the TempBook with C 6

**Reference Note**: The 32-bit API commands do not work exactly like the 16-bit API commands; refer to chapters 10 and 11 if you are seeking information regarding 32-bit API.

## *Accessing TempBook from a Windows Program*

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

## *Accessing TempBook from a C-for-Windows Program*

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

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

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

## *High-Level Analog Input*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Low-Level Analog Input

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

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

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

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

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

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

Set the scan frequency for 1kHz.

```
tbkSetFreq(1000);
```

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

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

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

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

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

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

## Analog Input in the Background

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

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

Initialize the TempBook on LPT1 with interrupt 7.

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

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

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

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

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

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

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

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

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

Make the Pacer Clock the trigger source.

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

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

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

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

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

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

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

## General Purpose Digital I/O Functions

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

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

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

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

Depending on its state, print a message.

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

Set the digital output bit 5 to a high state.

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

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

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

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

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

## High-Speed Digital Input

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

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

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

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

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

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

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

Initialize the TempBook on LPT1 with interrupt 7.

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

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

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

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

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

Set the trigger source to software trigger.

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

Trigger a scan.

```
tbkSoftTrig();
```

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

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

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

## Counter/Timer Functions

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

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

Configure CTR0 to use the internal 100 kHz clock.

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

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

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

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

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

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

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

## High-Level Thermocouple Data Acquisition

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

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

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

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

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

```
tbkRdTemp(0, TbkTypeJ, &temp);
sprintf(tempstr, "\r\nResults of tbkRdTemp\r\n");
strcat(response, tempstr);
sprintf(tempstr, "Temperature: %4,1f \r\n", (float)temp/10.0);
strcat(response, tempstr);
```

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

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

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

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

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

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

# *Thermocouple Linearization*

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

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

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

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

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

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

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

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

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

Set the pacer clock for a 1 msec period.

```
tbkSetFreq(1000);
```

Use the Pacer Clock as the trigger source.

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

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

```
tbkTCAutoZero(1);
```

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

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

Reset the trigger to prevent a buffer overrun.

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

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

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

The converted temperatures can now be printed to the screen.

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

## *Sample Programs*

### High-Level Analog Input

```
/************************************************
File:Adcex1.c
Description:This example demonstrates the use of TempBook/66s highest level
ADC functions.  These functions have combined scan sequencer setup and ADC
data collection.
Functions Used:tbkRd(unsigned chan, unsigned *sample, unsigned char gain)
tbkRdN(unsigned chan, unsigned *buf, unsigned count, unsigned char trigger,
unsigned char oneShot, float freq, unsigned char gain) tbkRdScan(unsigned
startChan, unsigned endChan, unsigned * buf, unsigned char gain)
tbkRdScanN(unsigned startChan, unsigned endChan, unsigned * buf, unsigned
count, unsigned char trigger, unsigned char oneShot, float freq, unsigned
char gain)
 ************************************************/

unsigned sample, buf[10], data[7], data2[80];
int i, scan, chan;   sprintf(response,"\r\nAdcex1.c\r\n\r\n");

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

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

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

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

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

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

for(i=0;i<8;i++)     {
    sprintf(tempstr,"%4d  ", buf[i]>4);
    strcat(response,tempstr);
}
      /* Get 1 sample from channels 0 through 7 at unity gain.  */
```

```
tbkRdScan(0, 7, data, TgainX1);

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

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

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

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

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

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

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

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

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

         sprintf(tempstr,"%4d  ", data2[(scan * 8) + chan]>4)
;        strcat(response,tempstr);
      }
  }
     SendDlgItemMessage(myDlg, 101, WM_SETTEXT, 0, (LONG)(LPSTR)response)
; EmptyMessageQueue(myDlg);
}
```

## Low-Level Analog Input

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

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

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

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

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

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

      /* Read 10 scans of data */
   tbkRdNFore(buf, 10);

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

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

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

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

## Analog Input in the Background

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## General Purpose Digital I/O

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

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

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

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

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

     // Write that byte to the digital output port
```

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

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

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

## High-Speed Digital Input

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

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

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

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

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

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

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

   tbkSetMode(0, 1);

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

   tbkSetScan(chans, gains, 0, 9);

      /* Set software trigger source */

   tbkSetTrig(TtsSoftware, 0, 0, 0);

      /* Trigger a scan */

   tbkSoftTrig();

      /* Read the ADC FIFO buffer */

   tbkRdNFore(data, 1);

      /* Print the results */

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

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

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

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

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

## Counter Timer Functions

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

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

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

   tbkSetTrig(TtsSoftware, 0, 1, 0);

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

   tbkConfCntr0(Tc0cOneShot);

   tbkWtCntr0(1000);

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

   tbkConfCntr0(Tc0cSquareWave);

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

## High-Level Thermocouple Measurement

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

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

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

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

   tbkSetMode(1, 1);

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

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

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

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

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

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

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

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

   tbkRdTempScan(0, 7, TbkTypeJ, temps);

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

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

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

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

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

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

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

## Low-Level Thermocouple Linearization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# 16-Bit API Command Summary, for C Language

| Command | Description |
|---|---|
| **Initialization, Communication and Error Handling:** | |
| tbkInit (uint lptPort, uchar lptIntr) | Establish communication with the TempBook at the specified LPT port and interrupt |
| tbkSelectPort (uint lptPort) | Select an initialized TempBook as the current TempBook |
| tbkClose (void) | End communication with the TempBook |
| tbkSetErrHandler (tbkErrorHandlerFPT tbkErrHandler) | Specify a user-defined routine to call when error occurs |
| tbkDefaultHandler (int tbkErrnum) | Default error handling routine |
| tbkGetProtocol (int *protocol) | Get the current parallel port communication protocol |
| tbkSetProtocol (int protocol) | Set the parallel port communication protocol |
| **Separate Scan Sequence, Pacer Clock and Trigger Commands:** | |
| tbkSoftTrig (void) | Issue a software trigger to the TempBook |
| tbkSetFreq (float freq) | Set the pacer clock frequency |
| tbkGetFreq (float _far *freq) | Read the pacer clock frequency |
| tbkSetClk (uint ctr1, uint ctr2) | Set the pacer clock frequency divider registers |
| tbkSetMode (uchar di_se, uchar polarity) | Specify input signal type (single-ended or differential) |
| tbkSetMux (uint startChan, uint endChan, uchar gain) | Setup the scan sequencer for a range of channels at the same gain |
| tbkGetScan (uint *chans, uchar *gains, uchar *polarity, uint *count) | Read the scan sequencer contents |
| **Digital I/O and Counter/Timer Functions:** | |
| tbkSetScan (uint *chans, uchar *gains, uchar *polarity, uint count) | Setup the scan sequencer with specific channels, gains, and polarities |
| tbkSetTrig (uchar trigger, uchar oneShot, uchar ctr0mode, uchar pacerMode) | Set the trigger source for analog data acquisition |
| **Separate Data Read and Background Transfer Commands:** | |
| tbkRdFore (uint _far *sample) | Read a single ADC sample in the foreground and increment the scan sequencer |
| tbkRdNFore (uint _far *buf, uint count) | Read multiple scans in the foreground |
| tbkRdNBack (uint _far *buf, uint count, uchar cycle, uchar updateSingle) | Read multiple scans in the background using interrupts |
| tbkGetBackStat (uchar _far *active, uint long _far *count) | Determine is a background transfer is still in progress |
| tbkStopBack (void) | Stop the background transfer |
| **Data Read Commands with Combined Scan Sequencer, Pacer Clock and Trigger Setup and Thermocouple Linearization:** | |
| tbkRd (uint chan, uint *sample, uchar gain) | Read a single sample from the specified channel |
| tbkRdN (uint chan, uint *buf, uint count, uchar trigger, uchar oneShot, float uchar gain) | Read multiple samples from the specified channel |
| tbkRdScan (uint startChan, uint endChan, uint *buf, uchar gain) | Read a single sample from the specified range of channels |
| tbkRdScanN (uint startChan, uint endChan, uint *buf, uint count, uchar trigger, uchar oneShot, float freq, uchar gain) | Read multiple samples from the specified range of channels |
| tbkRdTemp (uint chan, uint tcType, int temp) | Read the thermocouple temperature once from the specified channel |
| tbkRdTempN (uint chan, uint tcType, uint count, int *temp, uint *buf, float freq, uint avg) | Read the thermocouple temperature from the specified channel multiple times with optional averaging |
| tbkRdTempScan (uint startChan, uint endChan, uint tcType, int temp) | Read the thermocouple temperature once from each channel in a range |
| tbkRdTempScanN (uint startChan, uint endChan, uint tcType, uint count, int *temp, uint *buf, float freq, uint avg) | Read the thermocouple temperature from each channel in a range multiple times using optional averaging |
| **Thermocouple Linearization Commands:** | |
| tbkTCSetup (uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg) | Specify scan information used by tbkTCConvert |
| tbkTCConvert (uint _far *counts, uint scans, int _far *temp, uint ntemp) | Perform thermocouple linearization on scan data |
| tbkTCSetupConvert (uint nscan, uint cjcPosition, uint ntc, uint tcType, uchar bipolar, uint avg, uint _far *counts, uint scans, int *far *temp, uint ntemp) | Combined tbkTCSetup and tbkTCConvert |
| tbkTCAutoZero (uint zero) | Tell the thermocouple linearization functions that auto zeroing will be used |

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

# Software Calibration and Zero Compensation 7

**Reference Note**: The 32-bit API commands do not work exactly like the 16-bit API commands that are discussed in this chapter. Refer to chapters 10 and 11 if you are seeking information regarding 32-bit API.

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

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

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

Zero compensation removes offset errors while a program is running. This is useful in systems where the offset of a channel may change due to temperature changes, long-term drift or hardware calibration changes. By reading a shorted channel on the same card at the same gain as the desired channel, the offset can be removed at run-time. **Note**: The TempBook has channel 18 permanently shorted for performing zero compensation.

## Software Calibration

Software calibration functions are designed to adjust TempBook readings to compensate for gain and offset errors. Calibration constants are calculated at the factory by measuring the gain and offset errors of a card at each programmable gain setting. These constants are stored in a calibration text file which can be read by a program at runtime. This allows new boards to be configured for calibration by updating this calibration file rather than re-compiling the program.

The calibration process is divided up into three steps:

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

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

### Initializing the Calibration Constants

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

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

```
int tbkReadCalFile(char_*calfile)
```

| | |
|---|---|
| `char_*calfile` | calfile contains the path (optional) and filename of the calibration file. If calfile is NULL or empty (""), the default calibration file TempBook.CAL will be read. |

This function, which is usually called once at the beginning of a program, will read all the calibration constants from the specified file. If calibration constants for a specific channel number and gain setting are not contained in the file, ideal calibration constants will be used—essentially performing no calibration for that channel. If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and a non-zero error code will be returned by the **`tbkReadCalFile`** function.

## Calibration Setup and Conversion

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

```
int
tbkCalSetup ()
```

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

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

```
int
tbkCalConvert ()
```

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

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

```
int
tbkCalSetupConvert ()
```

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

## Calibration Example

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

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

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

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

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

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

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

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

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

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

     /* Close and exit */
tbkClose
}
```

## *Zero Compensation*

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

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

**`int tbkZeroSetup ()`**

| | |
|---|---|
| `uint nscan` | The number of readings in a single scan. |
| `uint zeroPosition` | The position of the zero reading within the scan. |
| `uint readingsPosition` | The position of the readings to be zeroed within the scan. |
| `uint nReadings` | The number of readings per scan to be zero compensate. |

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

**`int tbkZeroConvert ()`**

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

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

**`int tbkZeroSetupConvert ()`**

| | |
|---|---|
| `uint nscan` | The number of readings in a single scan. |
| `uint zeroPosition` | The position of the zero reading within the scan. |
| `uint readingsPosition` | The position of the readings to be zeroed. |
| `uint nReadings` | The number of readings per scan to be zero compensated. |
| `uint *counts` | The raw data from one or more scans. |
| `uint scans` | The number of scans of raw data in the counts array. |

## Zero Compensation Example

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

```
void main(void)
{
 unsigned int data[9], chans[9], i;
 unsigned char gains[9], polarities[9];
 printf(“/nAdcex5.c/n”);

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

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

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

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

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

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

      /* Print the uncompensated samples using a 12-bit format.*/
 printf(“/nResults of tbkRdNFore before zero compensation:/n”);
 for(i=0; i<8; i++)
      printf(“Channel: %2d Data: %4d/n”, i, data[i]>4);
     /* Setup and perform zero compensation of the data */
tbkZeroSetup(9,  /* 9 readings within a scan */
           0,    /* Shorted channel at position 0 */
           1,    /* First reading to be compensated at position 1 */
           8,    /* Compensate 8 readings per scan */
tbkZeroConvert(data,  /*Pointer to array of readings */
           1);   /* 1 scan in that array */

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

     /* Close and exit */
tbkClose
}
```

## Automatic Zero Compensation

The **`tbkTCAutoZero`** function will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation when making thermocouple measurements. When enabled, the thermocouple conversion functions will require a CJC zero reading and a TC zero reading to precede the actual CJC and TC reading.

```
int tbkTCAutoZero ()
```

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

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

Notes

# Thermocouple Measurement 8

**Reference Note**: The 32-bit API commands do not work exactly like the 16-bit API commands; refer to chapters 10 and 11 for details regarding 32-bit API.

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

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

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

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

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

## *Low-Level Thermocouple Data Conversion Functions*

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

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

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

The gain settings for the CJC and thermocouple types depend on the bipolar/unipolar setting of the TempBook as specified in the table below.

Do not use unipolar operations for thermocouple measurement unless the temperatures to be measured are guaranteed to be greater than the TempBook temperature.

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

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

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

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

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

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

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

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

| | |
|---|---|
| `uint nscan` | The number of readings in a single scan of TempBook data. The `tbkTC` functions can convert several consecutive scans worth of data in a single invocation.<br>Valid range: 2 to 512. |
| `uint cjcPosition` | The position of the actual cold-junction compensation circuit (CJC) reading within each scan (not the CJC zero reading, if any). The first reading of the scan is position 0, and the last reading is position nscan-1. Each scan of temperature data must include a reading of the CJC. The CJC readings must be taken with the appropriate unipolar or bipolar CJC gain setting.<br>Valid range: 0 to nscan-2 with no zero compensation; 2 to nscan-2 with zero compensation. |
| `uint ntc` | The number of thermocouple readings that are to be converted to temperature values. The thermocouple signal readings must immediately follow the CJC reading in the scan data. The first thermocouple signal is at scan position cjcPosition+1, the next is at cjcPosition+2, and so on.<br>Valid range: 1 to nscan-1-cjcPosition. |
| `uint tcType` | The type of thermocouples that generated the measurements: J, K, T, E, N28, N14, S, R, or B.<br>Valid range: One of the predefined values: `TbkTCTypeJ, TbkTCTypeK, TbkTCTypeT, TbkTCTypeE, TbkTCTypeN28, TbkTCTypeN14, TbkTCTypeS, TbkTCTypeR or TbkTCTypeB.` |

| | |
|---|---|
| `uchar bipolar` | Non-zero if the TempBook is configured for bipolar readings. |
| `uint avg` | The type of averaging to be performed.<br>Valid range: any unsigned integer.<br>Note: Since the thermocouple voltage may be small compared to the ambient electrical noise, averaging may sometimes be necessary to yield a steady temperature output.<br>0 specifies block averaging in which all of the scans are averaged together to compute a single temperature measurement for each of the ntemp thermocouples.<br>1 specifies no averaging. Each scan's readings are converted into ntemp measured temperatures for a total of scans*ntemp results.<br>2 or more specifies moving average of the specified number of scans. Each scan's readings are averaged with the avg-1 preceding scans' readings before conversion. The first avg-1 scans are averaged with all of the preceding scans because they do not have enough preceding scans. For example, if avg is 3, then the results from the first scan are not averaged at all, the results from the second scan are averaged with the first scan, the results from the third and subsequent scans are averaged with the preceding two scans as shown in the next table. |

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

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

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

| | |
|---|---|
| `uint *counts` | A array of one or more scans of raw data as received from the TempBook. The ADC data bits are in the 12 most significant bits of the 16-bit integers.<br>Valid range: Each raw data item may be any 16-bit value. |
| `uint scans` | The number of scans of data in counts.<br>Valid range: 1 to 32768/nscan (counts is limited to 64 Kbytes) |
| `int *temp` | The converted temperature results. The integer values are 10 times the temperatures in degrees C; e.g., 50°C would be represented as 500 and -10°C would be -100.<br>Valid range: Results range from -2000 (-200°C) to +13720 (+1372°C) depending on the thermocouple type. |
| `uint ntemp` | The number of entries in the temp array. This is checked by the functions to avoid writing past the end of the temp array.<br>Valid range: If avg is 0, then ntc or greater.<br>If avg is non-zero, then scans * ntc or greater. |

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

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

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

```
tbkTCAutoZero (uint zero)
```

| | |
|---|---|
| `uint zero` | Non-zero to enable auto-zeroing;<br>zero (0) to disable auto-zeroing. |

**Reference Note**: See the tempex2 example programs in the individual language support chapters foreexamples of using these functions.

# *High-Level Thermocouple Measurement Functions*

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

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

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

## Single-Channel Measurement (`tbkRdTemp`)

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

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

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

## Multiple Measurements from a Single Channel (`tbkRdTempN`)

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

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

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

## Multiple-Channel Measurement (`tbkRdTempScan`)

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

**Reference Note**:
Refer to the *Command Reference* chapter for additional information, if needed.

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

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

## Multiple Measurements from Multiple Channels (`tbkRdTempScanN`)

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

**Reference Note**:
Refer to the *Command Reference* chapter for additional information, if needed.

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

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

Notes

# tbkCommand Reference (16-Bit API) 9

## Overview

The first part of this chapter describes the TemBook/66 driver commands. These are the **16-bit API** commands. Do not confuse them with the **32-bit API** commands that are discussed in chapters 10 and 11. The first table lists the commands by their function types as defined in the driver header files. Then, the prototype commands are described in alphabetical order as indexed below. At the end of the chapter (beginning on page 9-32), several reference tables define parameters for: A/D Channel Descriptions, A/D Gain Definitions, A/D Trigger Source Definitions, Pre-Trigger Functions, Thermocouple Types, and the API Error Codes.

These TempBook software commands are described on the following pages:

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

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

## *Commands in Alphabetical Order*

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

## tbkCalConvert

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

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

## tbkCalSetup

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

The **tbkCalSetup** function will configure the order and type of data to be calibrated. This function requires all data to be calibrated to be from channels configured for the same gain, polarity and channel type. The calibration can be configured to only use the gain calibration constant and not the offset constant. This allows the offset to be removed at runtime using the zero compensation functions. See the Software Calibration and Zero Compensation chapter for a complete description of calibration.

## tbkCalSetupConvert

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

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

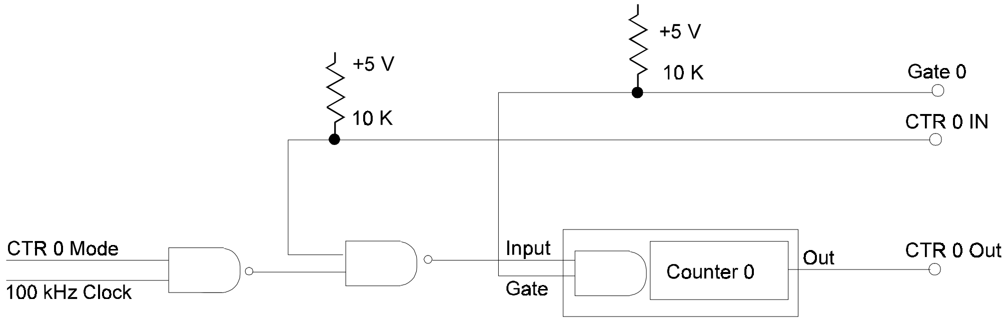
## tbkClose

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

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

# tbkConfCntr0

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



**`tbkConfCntr0`** programs the control register of Counter 0 in one of six modes. Counter 0 is a general purpose counter with input, gate and output lines. The input of counter 0 can be configured using the **`ctr0mode`** parameters of the **`tbkSetTrig`** command.

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

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

Mode 2, rate generator, acts as a divide-by-N counter. The output will be high until the counter value decrements to 1, when the output goes low for 1 clock pulse before going high again.

Mode 3, square wave generator, is similar to mode 2 except for the duty-cycle. The output will be high for half of the count value, and low for the other half. If the count value is odd, the output will remain high for the extra clock pulse.

Mode 4, software triggered strobe, will strobe each time the count value is loaded. The output is initially high. After the count value is written and has decremented to 1, the output will go low for one clock pulse before going high again.

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

## tbkDefaultHandler

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

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

## tbkGetBackStat

| | |
|---|---|
| **DLL Function** | `int tbkGetBackStat(uchar *active, ulong *count);` |
| **C** | `tbkGetBackStat(unsigned char _far *active, unsigned long _far *count);` |
| **QuickBASIC** | `BtbkGetBackStat%(active%, count&)` |
| **Visual Basic** | `VBtbkGetBackStat% (active%, count&)` |
| **Turbo Pascal** | `tbkGetBackStat( active:ByteP; count:LongP ):integer;` |
| **Parameters** | |
| `uchar *active` | A flag which will be returned non-zero if a background transfer is in progress, or 0 if not |
| `ulong *count` | The number of scans acquired by the last or current background transfer |
| **Returns** | `TerrOverrun` - Internal data buffer overrun<br>`TerrFIFOFull` - ADC FIFO Overrun<br>`TerrNoError` - No error |
| **See Also** | `tbkRdNBack, tbkStopBack` |
| **Program References** | None |

**`tbkGetBackStat`** determines if a background operation is still in progress. It also reads the number of bytes acquired by the last or current background operation initiated by the **`tbkRdNBack`** function.

**`tbkGetBackStat`** can return two possible error codes. **`TerrFIFOFull`** is returned if the data FIFO in the TempBook/66 is filled before the user can read the data out. In which case, the data read may be invalid. If the **`tbkRdNBack`** is called with the cycle flag enabled, a **`TerrOverrun`** can be returned. This means that the software is just fast enough to read one buffer of data. If this error occurs, the amount of data available (specified by 'count') is valid, but the transfer was stopped.

## tbkGetFreq

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

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

## tbkGetProtocol

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

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

## tbkGetScan

<table>
<tr><td><b>DLL Function</b></td><td><code>int tbkGetScan(uint *chans, uchar *gains, uchar *polarities, uint *count);</code></td></tr>
<tr><td><b>C</b></td><td><code>tbkGetScan(unsigned *chans, unsigned char *gains, unsigned char *polarity, unsigned *count);</code></td></tr>
<tr><td><b>QuickBASIC</b></td><td><code>BtbkGetScan% (chans%, gains%, polarity%, ByVal count%)</code></td></tr>
<tr><td><b>Visual Basic</b></td><td><code>VBtbkGetScan% (chans%(), gains%(), polarity%(), count%)</code></td></tr>
<tr><td><b>Turbo Pascal</b></td><td><code>tbkGetScan( chans:WordP; gains:ByteP; polarity:ByteP; count:word ):integer;</code></td></tr>
<tr><td colspan="2"><b>Parameters</b></td></tr>
<tr><td><code>uint *chans</code></td><td>An array to hold up to 512 channel numbers or 0 if the channel information is not desired. See table at end of chapter for valid values.</td></tr>
<tr><td><code>uchar *gains</code></td><td>An array to hold up to 512 gain values or 0 if the channel gain information is not desired. See table at end of chapter for valid values.</td></tr>
<tr><td><code>uchar polarity</code></td><td>Zero value causes TempBook to default to Unipolar mode. Non-zero value causes default Bipolar mode. All ADC conversions except those started with <code>tbkSetScan</code> will use the default polarity.</td></tr>
<tr><td><code>uint count</code></td><td>A variable to hold the number of values returned in the chans and gains arrays.</td></tr>
<tr><td><b>Returns</b></td><td><code>TerrNoError</code> - No error</td></tr>
<tr><td><b>See Also</b></td><td><code>tbkSetScan, tbkSetMux</code></td></tr>
<tr><td><b>Program References</b></td><td>None</td></tr>
</table>

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

## tbkInit

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

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

`tbkInit` will perform the following tasks:

- Stop any current acquisition
- Set the scan group to channel 1 with a gain of 1
- Set the pacer clock to 100 kHz
- Reset the counter/timers

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

## tbkRd

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

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

## tbkRdBit

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

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

## tbkRdByte

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

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

## tbkRdCntr0

| | |
|---|---|
| **DLL Function** | `int tbkRdCntr0(uint *cntr0, uchar latch);` |
| **C** | `tbkRdCntr0(unsigned _far *cntr0, unsigned char mode);` |
| **QuickBASIC** | `BtbkRdCntr0% (cntr0%, ByVal latch%)` |
| **Visual Basic** | `VBtbkRdCntr0% (cntr0%, latch%)` |
| **Turbo Pascal** | `tbkRdCntr0( cntr0:WordP; mode:Byte ):integer;` |
| **Parameters** | |
| `uint *cntr0` | The value read back from the Counter 0 hold register<br>Valid values: 0 - 65535 |
| `uchar latch` | If latch is non-zero, the count register will be latched into the hold register before reading.<br>If latch is zero, the count register will be read directly.<br>Direct reading should only be performed when no clock pulses are present. |
| **Returns** | `TerrNoError` - No error |
| **See Also** | `tbkConfCntr0, tbkWtCntr0` |
| **Program References** | None |

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

## tbkRdFore

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

**tbkRdFore** will read one sample from the A/D data FIFO. This function, unlike the **tbkRd** function, will not configure the trigger source. It assumes that the A/D converter has already been configured to acquire data.

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

## tbkRdN

| | |
|---|---|
| **DLL Function** | int tbkRdN(uint chan, uint *buf, uint count, uchar  trigger, uchar oneShot, float freq, uchar gain); |
| **C** | tbkRdN(unsigned chan, unsigned *buf, unsigned count, unsigned char trigger, unsigned char oneShot, float freq, unsigned char gain); |
| **QuickBASIC** | BtbkRdN% (ByVal chan%, buf%, ByVal count%, ByVal trigger%, ByVal oneShot%, ByVal freq!, ByVal gain%) |
| **Visual Basic** | VBtbkRdN% (chan%, buf%(), count%, trigger%, oneShot%, freq!, gain%) |
| **Turbo Pascal** | tbkRdN( chan:word; buf:WordP; count:word; trigger:byte; oneShot:byte; freq:real; gain:byte ):integer; |
| **Parameters** | |
| unit  chan | A single channel number (see table at end of chapter for valid values) |
| uint *buf | An array where the A/D scans will be returned |
| uint count | The number of scans to be taken<br>Valid values: 1 - 32767 |
| uchar trigger | The trigger source (see table at end of chapter for valid values) |
| uchar one shot | A flag that if non-zero enables one-shot trigger mode, otherwise enables continuous mode. |
| float freq | The sampling frequency in Hz (100000.0 to 0.0002) |
| uchar gain | The channel gain (see table at end of chapter for valid values) |
| **Returns** | TerrFIFOFull - Buffer overrun<br>TerrInvGain - Invalid gain<br>TerrIncChan - Invalid channel<br>TerrInvTrigSource - Invalid trigger<br>TerrInvLevel - Invalid level |
| **See Also** | tbkRd, tbkRdScan, tbkRdScanN, tbkRdNFore, tbkSetFreq, tbkSetMux, tbkSetClk, tbkSetTrig |
| **Program References** | None |

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

## tbkRdNBack

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

**tbkRdNBack** reads multiple A/D scans in the background using interrupts. This function will return control back to the user's program after initiating the background transfer. The user can then monitor the status of the background transfer with the **tbkGetBackStat** function or stop the transfer with the **tbkBackStop** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes that the A/D acquisition has already been setup.

If the cycle flag is true, the background transfer will run continuously looping back to the beginning of 'buf' after 'count' scans have been read. This allows the user to read large amounts of data without calling **tbkRdNBack** multiple times. As long as the user monitors how much data is in the buffer and processes the data before it gets overwritten, the background transfer can run indefinitely. In this mode, the user should get the total number of scans written into 'buf' using **tbkGetBackStat** and keep track of the total number of scans processed in a variable. The difference between these two totals is the number of unprocessed valid scans in 'buf' that the user can process.

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

The **updateSingle** flag allows the user to control whether the TempBook/66 updates 'buf' one sample at a time or in blocks of 256 scans. Enabling **updateSingle** allows the user to read A/D data during slow acquisitions as the data is acquired. Because the **updateSingle** flag is directly tied to the number of interrupts that will be generated on the computer, the flag should not be enabled if the acquisition rate is greater than roughly 500 scans per second (sampling rate * # of channels). For example, an acquisition running at 1 Hz might enable the **updateSingle** flag so that the data can be read each second rather than waiting for 256 seconds. An acquisition running at 10,000 Hz would disable the flag so the computer does not hang.

## tbkRdNBackPreT

| DLL Function | `int tbkRdNBackPreT(uint *buf, uint count, uchar cycle);` |
|---|---|
| C | `tbkRdNBackPreT(unsigned int _far *buf, unsigned int count, unsigned char cycle);` |
| QuickBASIC | `BtbkRdNBackPreT% (buf%, ByVal count%, ByVal cycle%)` |
| Visual Basic | `VBtbkRdNBackPreT% (buf%(), count%, cycle%)` |
| Turbo Pascal | `tbkRdNBackPreT(buf:WordP; count:word; cycle:byte):integer;` |
| **Parameters** | |
| `uint *buf` | An array where the A/D scans will be placed. |
| `uint count` | The number of scans to be taken (1-32767) |
| `uchar cycle` | A flag that if non-zero will enable continuous operation, or if 0 will disable it |
| Returns | `TerrMultBackXfer` - Background read already in progress<br>`TerrNoError` - No error |
| See Also | `tbkGetBackStat, tbkBackStop, tbkSetTag, tbkSetTrigPreT` |
| Program References | PRETEX3 (all languages) |

**tbkRdNBackPreT** reads multiple A/D scans, initiated by the **tbkSetTrigPreT** command, in the background. This function will return control to the user's program after initiating the background transfer. The user can then monitor the status of the background transfer with the **tbkGetBackStat** function or stop the transfer with the **tbkBackStop** function. Because the transfer occurs in the background, the user can perform other tasks in the foreground. This function assumes that the pre-trigger acquisition has already been setup using the **tbkSetTrigPreT** command.

If the 'cycle' flag is true, the background transfer will run continuously looping back to the beginning of 'buf' after 'count' scans have been read. Under this mode, the background transfer will continue until the acquisition completes. This allows the user to collect large amounts of data without calling **tbkRdNBackPreT** several times. As long as the user monitors how much data is in the buffer and processes the data before it gets overwritten, the background transfer can run until the acquisition completes. In this mode, the user should get the total number of scans written into 'buf' using the **tbkGetBackStat** function and keep track of the total number of scans processed in a variable. The difference between these two totals is the number of unprocessed valid scans in 'buf' that the user can process.

If, however, the 'cycle' flag is false, the background transfer will only collect the number of scans specified in 'count'. If this is the case, a number of **tbkRdNBack** calls may be necessary to read all the data collected during the pre-trigger mode acquisition.

## tbkRdNFore

| DLL Function | `int tbkRdNFore(uint *buf, uint count);` |
|---|---|
| C | `tbkRdNFore(unsigned _far *buf, unsigned count);` |
| QuickBASIC | `BtbkRdNFore% (buf%, ByVal count%)` |
| Visual Basic | `VBtbkRdNFore% (buf%(), count%)` |
| Turbo Pascal | `tbkRdNFore( buf:WordP; count:word ):integer;` |
| **Parameters** | |
| `uint *buf` | An array where the A/D samples will be placed |
| `uint count` | The number of scans to be taken<br>Valid values: 1 - 32767 |
| Returns | `TerrFIFOFull` - Buffer overrun<br>`TerrNoError` - No error |
| See Also | `tbkSetClk, tbkSetTrig` |
| Program References | |

**tbkRdNFore** reads multiple A/D scans in the foreground. Unlike **tbkRdNBack**, this function does not use interrupts and does not return control immediately to the program. It will return only when ‘count’ scans have been read. This function will not configure the A/D acquisition and assumes that the A/D converter has already been configured to acquire data.

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

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

**`tbkRdNForePreT`** reads multiple A/D scans, initiated by the **`tbkSetTrigPreT`** command, in the foreground.  Unlike the **`tbkRdNBackPreT`** command, this function does not use interrupts and does not return control immediately to the application program.  It will only return when either the specified count has been satisfied or the acquisition completes.

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

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

# tbkRdNForePreTWait

| | |
|---|---|
| **DLL Function** | `int tbkRdNForePreTWait(uint *buf, uint count, uint *retcount);` |
| **C** | `tbkRdNForePreTWait(unsigned int _far *buf, unsigned int count, unsigned int _far *retcount);` |
| **QuickBASIC** | `BtbkRdNForePreTWait% (buf%, ByVal count%, retcount%)` |
| **Visual Basic** | `VBtbkRdNForePreTWait% (buf%(), count%, retcount%)` |
| **Turbo Pascal** | `tbkRdNForePreTWait( buf:WordP; count:word; retcount:WordP):integer;` |
| **Parameters** | |
| `uint *buf` | An array where the A/D samples will be placed |
| `uint count` | The number of scans to be taken<br>Valid values:  1 - 32767 |
| `uint *retcount` | Pointer to an integer representing the number of scans actually taken |
| **Returns** | `TerrFIFOFull` - Buffer overrun<br>`TerrNoError` - No error |
| **See Also** | `tbkSetTrigPreT, tbkRdNForePreT, tbkRdNBackPreT` |
| **Program References** | PRETEX2 (all languages) |

**`tbkRdNForePreTWait`** reads multiple A/D scans, initiated by the **`tbkSetTrigPreT`** command, in the foreground.  Unlike the **`tbkRdNForePreT`** command, this function will not return until the acquisition completes.  It will only return when the specified trigger event has occurred and the specified post trigger count has been satisfied.

This function may be called subsequent to configuring a pre-trigger acquisition using the **`tbkSetTrigPreT`** command.  Once this command has been called, it will return only when the trigger has been detected and the acquisition has completed.  The amount specified in the 'count' parameter specifies the length of the supplied buffer in scans.  Unlike **`tbkAdcRdNForePreT`** this command will not return when 'count' is satisfied; instead, it will continue acquiring by wrapping the scans to the beginning of the buffer until the final post-trigger scan is collected and the acquisition completes.

When the acquisition completes, control will be returned to the application program along with the actual number of scans collected in the 'retcount' parameter.

**Note**: If the A/D converter has not been configured to acquire data or the trigger event never occurs, this function may wait indefinitely, hanging the computer.

# tbkRdScan

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

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

## tbkRdScanN

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

`tbkRdScanN` reads multiple scans from multiple A/D channels.  This function will configure the pacer clock, arm the trigger and acquire 'count' scans consisting of each channel starting with 'startChan' and ending with 'endChan'.

## tbkRdTemp

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

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

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

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

## tbkRdTempN

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

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

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

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

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

## tbkRdTempScan

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

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

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

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

## tbkRdTempScanN

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

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

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

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

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

## tbkReadCalFile

| | |
|---|---|
| **DLL Function** | `tbkReadCalFile(char *calfile);` |
| **C** | `tbkReadCalFile(char *calfile);` |
| **QuickBASIC** | `BtbkReadCalFile% (ByVal calfile$)` |
| **Visual Basic** | `VBtbkReadCalFile% (ByVal calfile$)` |
| **Turbo Pascal** | `tbkReadCalFile(calfile : string) : integer;` |
| **Parameters** | |
| `char *calfile` | The file name with optional path information of the calibration file. If calfile is NULL or empty (""), the default calibration file TEMPBOOK .CAL will be read. |
| **Returns** | `TerrNoError` - No error<br>`TerrInvCalfile` - Error occurred while opening or reading calibration file |
| **See Also** | `tbkCalSetup, tbkCalConvert, tbkCalSetupConvert` |
| **Program References** | None |

**`tbkReadCalFile`** is the initialization function for reading in the calibration constants from the calibration text file. This function, which is usually called once at the beginning of a program, will read all the calibration constants from the specified file. If calibration constants for a specific gain setting are not contained in the file, ideal calibration constants will be used, essentially performing no calibration for that channel. If an error occurs while trying to open the calibration file, ideal calibration constants will be used for all channels and a non-zero error code will be returned by the **`tbkReadCalFile`** function.

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

## tbkSelectPort

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

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

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

## tbkSetClk

| | |
|---|---|
| **DLL Function** | `int tbkSetClk(uint ctr1, uint ctr2);` |
| **C** | `tbkSetClk(unsigned ctr1, unsigned ctr2);` |
| **QuickBASIC** | `BtbkSetClk% (ByVal ctr1%, ByVal ctr2%)` |
| **Visual Basic** | `VBtbkSetClk% (ctr1%, ctr2%)` |
| **Turbo Pascal** | `tbkSetClk( ctr1:word; ctr2:word ):integer;` |
| **Parameters** | |
| `uint ctr1` | The value of the counter 1 divisor<br>Valid values: 0 - 65535 |
| `uint ctr2` | The value of the counter 2 divisor<br>Valid values: 0 - 65535 |
| **Returns** | `TerrInvClock` - Invalid clock<br>`TerrNoError` - No error |
| **See Also** | `tbkSetFreq, tbkGetFreq` |
| **Program References** | |

**`tbkSetClk`** sets the frequency of the pacer clock using the two specified counter values. The pacer clock can be used to control the sampling rate of the A/D converter. The frequency is defined to be xtal/ctr1*ctr2) where xtal is the frequency of the board crystal (either 1 MHz or 100 kHz).

## tbkSetErrHandler

| | |
|---|---|
| **DLL Function** | `tbkSetErrHandler(tbkSetErrHandlerFTP tbkErrorHandler);` |
| **C** | `tbkSetErrHandler(tbkErrorHandlerFPT tbkErrorHandler);` |
| **QuickBASIC** | `BtbkSetErrHandler% (ByVal tbkErrorHandler&)` |
| **Visual Basic** | `VBtbkSetErrHandler% (tbkErrorHandler%)` |
| **Turbo Pascal** | `tbkSetErrHandler( tbkErrorHandler:ErrorFuncT ):integer;` |
| **Parameters** | |
| `tbkErrHandler` | This is a function that takes an integer (error code) and returns nothing, or NULL to disable. |
| **Returns** | `TerrNoError` - No error, or an error number |
| **See Also** | `tbkDefaultHandler` |
| **Program References** | |

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

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

## tbkSetFreq

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

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

## tbkSetMode

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

**`tbkSetMode`** is used to program the gain amp for single-ended or differential operation and to set the default polarity.

Single-ended operation measures the voltage of the selected channel referred to analog ground. Differential operation measures differences in voltage between a pair of selected channels.

Polarity is unipolar or bipolar:

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

## tbkSetMux

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

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

# tbkSetProtocol

| | |
|---|---|
| **DLL Function** | `int tbkSetProtocol( int  protocol)` |
| **C** | `tbkSetProtocol(int protocol);` |
| **QuickBASIC** | `BtbkSetProtocol% (ByVal protocol%)` |
| **Visual Basic** | `VBtbkSetProtocol% (protocol%)` |
| **Turbo Pascal** | `tbkSetProtocol( protocol:integer ):integer;` |
| **Parameters** | |
| `protocol` | One of the predefined protocol codes listed below (additional protocol codes may be described in the README file). |

| Name | Description | Value |
|---|---|---|
| `TbkProtocol8` | 8-bit I/O | 1 |
| `TbkProtocol4` | 4-bit I/O | 2 |
| `TbkProtocolFPort` | Far Point F/Port EPP Interface | 10 |
| `TbkProtocolSL` | 82360 SL EPP Interface | 20 |
| `TbkProtocolSMC666` | SMC 37C666 EPP mode | 30 |
| `TbkProtocolEPPBIOS` | EPP bios mode | 40 |
| `TbkProtocolPastEPP` | WBK20/21 Fast EPP mode | 50 |

| | |
|---|---|
| **Returns** | An error number, or 0 if no error. |
| **See Also** | `tbkInit, tbkGetProtocol` |
| **Program References** | |

**`tbkSetProtocol`** specifies to the TempBook/66 driver the type of parallel-port implementation and protocol that is available on the computer. The driver then attempts to configure the computer and the TempBook/66 to communicate using the specified protocol. Since establishing the protocol may affect the settings of the TempBook, **`tbkSetProtocol`** should only be invoked immediately after **`tbkInit`** has established communications with and reset the TempBook. Switching protocols during normal TempBook/66 operation is not recommended.

Two types of parallel port implementations are supported by the TempBook: Standard and enhanced. Standard parallel ports, using the TempBook/66 manufacturer's proprietary protocols, are capable of receiving data either 4 or 8 bits at a time. When possible, 8-bit operation is preferred (it is much faster), but not all standard parallel ports support 8-bit data reception.

Enhanced parallel ports (EPP) include extra hardware that increases the rate of data transfer to 3 to 10 times the rate of a standard parallel port. Unfortunately, not every computer includes EPP capability and attempting to use EPP on an incompatible computer may cause the TempBook/66 driver to access I/O locations which are not part of the printer port interface. Such accesses may interfere with other operations and cause the computer to operate incorrectly. For this reason, EPP operation must be explicitly requested by the program.

When the TempBook/66 is initialized by **`tbkInit`**, it is initially configured for a standard parallel port protocol: either 8-bit, if possible, or the slower 4-bit protocol. After **`tbkInit`** has completed, **`tbkSetProtocol`** may be used to switch to another supported protocol.

If **`tbkSetProtocol`** is unable to establish communications using the specified protocol, then it will try to establish communications using the standard port protocols, first 8-bit, then the slower 4-bit. In such an event, **`tbkSetProtocol`** will not return an error indication unless it is unable to establish any protocol.

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

# tbkSetScan

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

`tbkSetScan` configures a scan sequence consisting of multiple channels, polarities and corresponding gains. As many as 512 entries can be made in the scan configuration. Any analog input channel at any gain can be included in the scan. Channels can be entered multiple times at the same or different gain. The high-speed digital I/O port can also be included although its gain value will be ignored.

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

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

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

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

An external TTL pulse can be used to initiate a scan or start an acquisition when using the external TTL rising or falling edge trigger source. The external TTL pulse should be applied to the trig input. The pulse will initiate a single scan in one-shot mode and a continuous acquisition at the pacer clock frequency in continuous mode.

Setting the counter 0 mode flag true will enable an onboard 100 kHz clock to be ANDed with the counter 0 input to produce the input to counter 0. If nothing is connected to counter 0 input, the line will float high essentially clocking counter 0 off the 100KHz clock. If this flag is false, counter 0 can only be clocked from the counter 0 input pin. Counter 0 can be used as an alternative trigger source by connecting the counter 0 output to the trig input and choosing an external TTL trigger. Counter 0 can also be used for general counter applications.

The pacer mode flag enables/disables operation of the pacer clock. If this flag is non-zero, the pacer clock will be gated with the trig input. If it is zero, the pacer clock will be enabled.

## tbkSetTrigPreT

| | |
|---|---|
| **DLL Function** | `int tbkSetTrigPreT(uchar trigger, uint channel, uint level, uint precount, uint postcount);` |
| **C** | `tbkSetTrigPreT(unsigned char source, unsigned int channel, unsigned int level, unsigned int preCount, unsigned int postCount);` |
| **QuickBASIC** | `BtbkSetTrigPreT% (ByVal source%, ByVal channel%, ByVal level%, ByVal preCount%, ByVal postCount%)` |
| **Visual Basic** | `VBtbkSetTrigPreT% (source%, channels%, level%, preCount%, postCount%)` |
| **Turbo Pascal** | `tbkSetTrigPreT( source:byte; channel:word; level:word; preCount:word; postCount:word):integer;` |
| **Parameters** | |
| `uchar trigger` | The analog trigger source - `DtsAnalogRisePos, DtsAnalogFallPos, DtsAnalogRisNeg, DtsAnalogFallNeg` |
| `uint channel` | The channel in the current scan group to trigger on |
| `uint level` | The level for the specified channel at which to detect the trigger (0-4095) |
| `uint precount` | The number of pre-trigger scans to collect before arming the trigger (1-32767) |
| `uint postcount` | The number of post-trigger scans to collect after the occurrence of the trigger (1-32767) |
| **Returns** | `TerrInvTrigSource` - Invalid trigger<br>`TerrInvLevel` - Invalid level<br>`TerrNoError` - No error |
| **See Also** | `tbkSetFreq, tbkSetClk, tbkRdNForePreT, tbkRdNForePreTWait, tbkRdNBackPreT` |
| **Program References** | PRETEX1, PRETEX2, PRETEX3 (ALL LANGUAGES) |

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

This command allows the configuration of a data acquisition that includes both pre-trigger and post-trigger data. The specified pre-trigger amount indicates the number of pre-trigger scans to be collected before the trigger is armed. The trigger event will only be recognized after the specified pre-trigger amount has been satisfied and the trigger is armed. This means that the specified pre-trigger amount represents the minimum amount of pre-trigger data which will actually be collected. The specified post-trigger amount represents the number of scans taken after the detection of the trigger event. This amount represents the exact number of scans taken subsequent to the detection of the trigger event.

The pacer clock may be used to set up the sampling rate for the acquisition. The sampling rate can best be set by using the **`tbkSetClk`** or **`tbkSetFreq`** commands.

The four analog trigger sources, rising or falling slope with either a positive or negative level, can be used with any one of the channels in the currently defined scan group. This channel parameter represents the relative channel within the scan group. It does not necessarily represent the actual physical channel number.

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

## tbkSoftTrig

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

**tbkSoftTrig** is used to send a software trigger command to the TempBook. This software trigger can be used to initiate a scan or an acquisition from a program after configuring the software trigger as the trigger source.

## tbkStopBack

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

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

## tbkTCAutoZero

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

The **tbkTCAutoZero** function will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the TempBook. When enabled, the thermocouple conversion functions will require a CJC zero reading and a TC zero reading to precede the actual CJC and TC reading.

## tbkTCConvert

| | |
|---|---|
| **DLL Function** | `tbkTcConvert (uint *counts, uint  scans ,int *temp, uint ntemp);` |
| **C** | `tbkTCConvert(unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);` |
| **QuickBASIC** | `BtbkTCConvert% (counts%, ByVal scans%, temp%, ByVal ntemp%)` |
| **Visual Basic** | `VBtbkTCConvert% (counts%(), ByVal scans%, temp%(), ByVal ntemp%)` |
| **Turbo Pascal** | `tbkTCConvert(counts:WordP; scans:word; temp:DataP; ntemp:word):integer;` |
| **Parameters** | |
| `uint *counts` | Raw A/D data from one or more scans |
| `uint scans` | Number of scans of raw data in counts |
| `int * temp` | Variable array to hold converted temperatures |
| `uint ntemp` | Size of temperature array |
| **Returns** | **`TerrTCE_NOSETUP`** - Setup was not called<br>**`TerrTCE_PARAM`** - Param out of range<br>**`TerrNoError`** - No Error |
| **See Also** | `tbkTCSetup, tbkTCSetupConvert` |
| **Program References** | None |

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

## tbkTCSetup

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

**`tbkTCSetup`** sets up parameters for subsequent temperature conversions.

## tbkTCSetupConvert

| | |
|---|---|
| **DLL Function** | `int tbkTCSetupConvert(uint nscan,uint cjcPosition, uint ntc,  uint tcType,uchar bipolar,uint avg, uint *counts, uint scans, int *temp, );` |
| **C** | `tbkTCSetupConvert(unsigned nscan, unsigned cjcPosition, unsigned ntc, unsigned tcType, unsigned char bipolar, unsigned avg, unsigned _far *counts, unsigned scans, int _far *temp, unsigned ntemp);` |
| **QuickBASIC** | `BtbkTCSetupConvert% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%, counts%, ByVal scans%, temp%, ByVal ntemp%)` |
| **Visual Basic** | `VBtbkTCSetupConvert% (ByVal nscan%, ByVal cjcPosition%, ByVal ntc%, ByVal tcType%, ByVal bipolar%, ByVal avg%, counts%(), ByVal scans%, temp%(), ByVal ntemp%)` |
| **Turbo Pascal** | `tbkTCSetupConvert(nscan, cjcPosition, ntc, tcType:word; bipolar:byte; avg:word; counts:WordP; scans:word; temp:DataP; ntemp:word):integer;` |
| **Parameters** | |
| `uint nscan` | The number of readings in a single scan.<br>Valid range: 1- 512 |
| `uint cjcPosition` | The position of the CJC reading within the scan.<br>Valid range: 0 - (nscan-1)<br>2 -(nscan-1), if auto-zeroing is used. |
| `uint ntc` | The number of thermocouple readings that immediately follow the CJC reading within the scan.<br>Valid range: 1 - (nscan-cjcposition-1) |
| `uint tcType` | The type of thermocouples being measured (see table at end of chapter for valid types) |
| `uint bipolar` | Non-zero if the TempBook/66 is configured for bipolar readings. |
| `uint avg` | The type of averaging to be performed: block, none or moving. |
| `uint *counts` | The raw data from one or more scans. |
| `uint scans` | The number of scans of raw data in counts. |
| `int *temp` | The converted temperatures in tenths of a degree C. |
| `uint ntemp` | The number of elements provided in the temp array (for error checking). |
| **Returns** | `TerrTCE_PARAM` - Parameter out of range<br>`TerrTCE_TYPE` - Invalid thermocouple type<br>`TerrNoError` - No Error |
| **See Also** | `tbkTCSetup, tbkTCConvert` |
| **Program References** | None |

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

## tbkWtBit

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

`tbkWtBit` sets or clears a single digital output bit.

## tbkWtByte

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

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

## tbkWtCntr0

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

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

## tbkZeroConvert

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

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

## tbkZeroSetup

| DLL Function | tbkZeroSetup(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings); |
|---|---|
| C | tbkZeroSetup(unsigned nscan, unsigned zeroPos, unsigned readingsPos, unsigned nReadings); |
| QuickBASIC | BtbkZeroSetup% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%) |
| Visual Basic | VBtbkZeroSetup% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%) |
| Turbo Pascal | tbkZeroSetup( nscan:word; zeroPos:word; readingsPos:word; nReadings:word):integer; |
| **Parameters** | |
| uint nscan | The number of readings in a single scan. |
| uint zeroPosition | The position of the zero reading within the scan |
| uint readingsPosition | The position of the readings to be zeroed within the scan. |
| uint nReadings | The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading. |
| **Returns** | TerrZCInvParam - Invalid parameter value<br>TerrNoError - No error |
| **See Also** | tbkZeroConvert, tbkZeroSetupConvert, tbkAutoZero |
| **Program References** | None |

The **tbkZeroSetup** function configures the location of the shorted channel and the channels to be zeroed within a scan, the size of the scan and the number of readings to zero. This function does not do the conversion. A non-zero return value indicates an invalid parameter error. See the Software Calibration and Zero Compensation chapter for a complete description of zero compensation.

## tbkZeroSetupConvert

| DLL Function | tbkZeroSetupConvert(uint nscan, uint ZeroPosition, uint readingsPosition, uint nReadings, uint *counts, uint scans); |
|---|---|
| C | tbkZeroSetupConvert(unsigned nscan, unsigned zeroPos, unsigned readingsPos, unsigned nReadings, unsigned *counts, unsigned scans); |
| QuickBASIC | BtbkZeroSetupConvert% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%, counts%, ByVal scans%) |
| Visual Basic | VBtbkZeroSetupConvert% (ByVal nscan%, ByVal zeroPosition%, ByVal readingsPos%, ByVal nReadings%, counts%(), ByVal scans%) |
| Turbo Pascal | tbkZeroSetupConvert(nscan:word;zeroPos:word;readingsPos:word;nReadings:word; counts:WordP;scans:word):integer; |
| **Parameters** | |
| uint nscan | The number of readings in a single scan. |
| uint zeroPosition | The position of the zero reading within the scan |
| uint readingsPosition | The position of the readings to be zeroed within the scan. |
| uint nReadings | The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading. |
| uint *counts | The raw data from one or more scans. |
| uint scans | The number of scans of raw data in the counts array. |
| **Returns** | TerrZCInvParam - Invalid parameter value<br>TerrNoError - No error |
| **See Also** | tbkZeroSetup, tbkZeroConvert, tbkAutoZero |
| **Program References** | None |

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

# API Reference Tables

These tables provide information for programming with the TempBook/66 Application Programming Interface. The tables are organized as follows:

**API Parameter Reference Tables**

### *A/D Channel Descriptions*

| A/D Channel | Source |
|---|---|
| 0 to 15* | Analog input channels 0 to 15 |
| 16 | CJC Channel |
| 18 | Shorted Channel |
| 272 | High-speed digital Inputs |
| *Note: In differential mode only channels 0 to 7 are valid. | |

### *A/D Gain Definitions*

| BASE UNIT | |
|---|---|
| **Description** | **Value** |
| TgainX1 | 0x00 |
| TgainX2 | 0x10 |
| TgainX5 | 0x20 |
| TgainX10 | 0x30 |
| TgainX20 | 0x11 |
| TgainX50 | 0x21 |
| TgainX100 | 0x31 |
| TgainX200 | 0x32 |
| TbkBiCJC | 0x21 |
| TbkBiTypeJ | 0x31 |
| TbkBiTypeK | 0x31 |
| TbkBiTypeT | 0x32 |
| TbkBiTypeE | 0x21 |
| TbkBiTypeN28 | 0x31 |
| TbkBiTypeN14 | 0x31 |
| TbkBiTypeS | 0x32 |
| TbkBiTypeR | 0x32 |
| TbkBiTypeB | 0x32 |
| TbkUniCJC | 0x31 |
| TbkUniTypeJ | 0x32 |
| TbkUniTypeK | 0x32 |
| TbkUniTypeT | 0x32 |
| TbkUniTypeE | 0x31 |
| TbkUniTypeN28 | 0x32 |
| TbkUniTypeN14 | 0x32 |
| TbkUniTypeS | 0x32 |
| TbkUniTypeR | 0x32 |
| TbkUniTypeB | 0x32 |

### *A/D Trigger Source Definitions*

| Definition | Value | Trigger |
|---|---|---|
| `TtsPacerClock` | 0x00 | 8254 Pacer Clock |
| `TtsSoftware` | 0x10 | Software |
| `TtsTTLFall` | 0x20 | External TTL falling edge |
| `TtsTTLRise` | 0x30 | External TTL rising edge |

### *Pretrigger Functions Trigger Source Definitions*

| Definition | Value | Trigger |
|---|---|---|
| `TtsAnalogFallNeg` | 0x40 | Falling below a negative setpoint |
| `TtsAnalogRiseNeg` | 0x50 | Rising above a negative setpoint |
| `TtsAnalogRisePos` | 0x60 | Rising above a positive setpoint |
| `TDtsAnalogFallPos` | 0x70 | Falling below positive setpoint |

### *Thermocouple Types*

| Description | Value |
|---|---|
| `TbkTypeJ` | 18 |
| `TbkTypeK` | 19 |
| `TbkTypeT` | 20 |
| `TbkTypeE` | 21 |
| `TbkTypeN28` | 22 |
| `TbkTypeN14` | 23 |
| `TbkTypeS` | 24 |
| `TbkTypeR` | 25 |
| `TbkTypeB` | 26 |

### *API Error Codes - C Languages*

| Error Name | Error Code | Description |
|---|---|---|
| `TerrNoError` | 0x00 | No error |
| `TerrBadChannel` | 0x01 | Specified LPT channel was out-of-range |
| `TerrNotOnLine` | 0x02 | Requested TempBook is not on-line |
| `TerrNoTempBook` | 0x03 | TempBook is not on the requested channel |
| `TerrBadAddress` | 0x04 | Bad function address |
| `TerrFIFOFull` | 0x05 | FIFO Full detected, possible data corruption |
| `TerrInvChan` | 0x10 | Invalid analog input channel |
| `TerrInvCount` | 0x11 | Invalid count parameter |
| `TerrInvTrigSource` | 0x12 | Invalid trigger source parameter |
| `TerrInvGain` | 0x14 | Invalid channel gain parameter |
| `TerrInvPort` | 0x17 | Invalid port parameter |
| `TerrInvChip` | 0x18 | Invalid chip parameter |
| `TerrInvBitNum` | 0x1A | Invalid bit number parameter |
| `TerrInvClock` | 0x1B | Invalid clock parameter |
| `TerrInvTod` | 0x1C | Invalid time-of-day parameter |
| `TerrInvGateCtrl` | 0x20 | Invalid gate control parameter |
| `TerrInvOutputCtrl` | 0x21 | Invalid output control parameter |
| `TerrInvInterval` | 0x22 | Invalid interval parameter |
| `TerrTypeConflict` | 0x23 | An integer was passed to a function requiring a character |
| `TerrMultBackXfer` | 0x24 | A second background transfer was requested |
| `TerrInvDiv` | 0x25 | Invalid Fout divisor |
| `TerrTCE_TYPE` | 0x26 | TC type out-of-range |
| `TerrTCE_TRANGE` | 0x27 | Temperature out-of-CJC-range |
| `TerrTCE_VRANGE` | 0x28 | Voltage out-of-TC-range |
| `TerrTCE_PARAM` | 0x29 | Unspecified parameter value error |
| `TerrTCE_NOSETUP` | 0x2A | `tbkTCConvert` called before `tbkTCSetup` |
| `TerrOverrun` | 0x2C | A buffer overrun occurred |
| `TerrZCInvParam` | 0x2D | Invalid zero compensation parameter |
| `TerrZCNoSetup` | 0x2E | `tbkZeroConvert` called before `tbkZeroSetup` |
| `TerrInvCalFile` | 0x2F | Cannot open the specified calibration file |
| `TerrMemLock` | 0x30 | Cannot lock allocated memory from Windows |
| `TerrMemHandle` | 0x31 | Cannot get a memory handle from Windows |
| `TerrNoPreTActive` | 0x32 | No pre-trigger configured |

### *API Error Codes - QuickBASIC*

| Error Name | Error Code | Description |
|---|---|---|
| `CONST TerrNoError%` | &H00 | No error |
| `CONST TerrBadChannel%` | &H01 | Specified LPT channel was out-of-range |
| `CONST TerrNotOnLine%` | &H02 | Requested TempBook is not on-line |
| `CONST TerrNoTempBook%` | &H03 | TempBook is not on the requested channel |
| `CONST TerrBadAddress%` | &H04 | Bad function address |
| `CONST TerrFIFOFull%` | &H05 | FIFO Full detected, possible data corruption |
| `CONST TerrInvChan%` | &H10 | Invalid |
| `CONST TerrInvCount%` | &H11 | Invalid count parameter |
| `CONST TerrInvTrigSource%` | &H12 | Invalid trigger source parameter |
| `CONST TerrInvGain%` | &H14 | Invalid channel gain parameter |
| `CONST TerrInvPort%` | &H17 | Invalid port parameter |
| `CONST TerrInvChip%` | &H18 | Invalid chip parameter |
| `CONST TerrInvBitNum%` | &H1A | Invalid bit number parameter |
| `CONST TerrInvClock%` | &H1B | Invalid clock parameter |
| `CONST TerrInvTod%` | &H1C | Invalid time-of-day parameter |
| `CONST TerrInvGateCtrl%` | &H20 | Invalid gate control parameter |
| `CONST TerrInvOutputCtrl%` | &H21 | Invalid output control parameter |
| `CONST TerrInvInterval%` | &H22 | Invalid interval parameter |
| `CONST TerrTypeConflict%` | &H23 | An integer was passed to a function requiring a character |
| `CONST TerrMultBackXfer%` | &H24 | A second background transfer |
| `CONSTTerrInvDiv%` | &H25 | Invalid Fout divisor |
| `CONST TerrTCE.TYPE` | &H26 | TC type out of range |
| `CONST TerrTCE.TRANGE` | &H27 | Temperature out-of-CJC-range |
| `CONST TerrTCE.VRANGE` | &H28 | Voltage out-of-TC-range |
| `CONST TerrTCE.PARAM` | &H29 | Unspecified parameter value error |
| `CONST TerrTCE.NOSETUP` | &H2A | `tbkTCConvert` called before `tbkTCSetup` |
| `CONST TerrOverrun%` | &H2C | A buffer overrun occurred |
| `CONST TerrZCInvParam` | &H2D | Invalid zero compensation parameter |
| `CONST TerrZCNoSetup` | &H2E | `tbkZeroConvert` called before `tbkZeroSetup` |
| `CONST TerrInvCalFile` | &H2F | Cannot open the specified calibration file |
| `CONST TerrMemLock` | &H30 | Cannot lock allocated memory from Windows |
| `CONST TerrMemHandle` | &H31 | Cannot get a memory handle from Windows |
| `CONST TerrNoPreTActive` | &H32 | No pre-trigger configured |

### *API Error Codes - Turbo Pascal*

| Error Name | Error Code | Description |
|---|---|---|
| `TerrNoError` | 0 | No error |
| `TerrBadChannel` | 1 | Specified LPT channel was out-of-range |
| `TerrNotOnLine` | 2 | Requested TempBook is not on-line |
| `TerrNoTempBook` | 3 | TempBook is not on the requested channel |
| `TerrBadAddress` | 4 | Bad function address |
| `TerrFIFOFull` | 5 | FIFO Full detected, possible data corruption |
| `TerrInvChan` | 16 | Invalid |
| `TerrInvCount` | 17 | Invalid count parameter |
| `TerrInvTrigSource` | 18 | Invalid trigger source parameter |
| `TerrInvGain` | 20 | Invalid channel gain parameter |
| `TerrInvPort` | 23 | Invalid port parameter |
| `TerrInvChip` | 24 | Invalid chip parameter |
| `TerrInvBitNum` | 26 | Invalid bit number parameter |
| `TerrInvClock` | 27 | Invalid clock parameter |
| `TerrInvTod` | 28 | Invalid time-of-day parameter |
| `TerrInvGateCtrl` | 32 | Invalid gate control parameter |
| `TerrInvOutputCtrl` | 33 | Invalid output control parameter |
| `TerrInvInterval` | 34 | Invalid interval parameter |
| `TerrTypeConflict` | 35 | An integer was passed to a function requiring a character |
| `TerrMultBackXfer` | 36 | A second background transfer was requested |
| `TerrInvDiv` | 37 | Invalid Fout divisor |
| `TerrTCE_TYPE` | 38 | TC type out-of-range |
| `TerrTCE_TRANGE` | 39 | Temperature out-of-CJC-range |
| `TerrTCE_VRANGE` | 40 | Voltage out-of-TC-range |
| `TerrTCE_PARAM` | 41 | Unspecified parameter value error |
| `TerrTCE_NOSETUP` | 42 | `tbkTCConvert` called before `tbkTCSetup` |
| `TerrNot Capable` | 43 | TempBook not capable of function |
| `TerrOverrun` | 44 | A buffer overrun occurred |
| `TerrZCInvParam` | 45 | Invalid zero compensation parameter |
| `TerrZCNoSetup` | 46 | `tbkZeroConvert` called before `tbkZeroSetup` |
| `TerrInvCalFile` | 47 | Cannot open the specified calibration file |
| `TerrMemLock` | 48 | Cannot lock allocated memory from Windows |
| `TerrMemHandle` | 49 | Cannot get a memory handle from Windows |
| `TerrNoPreTActive` | 50 | No pre-trigger configured |

## *API Error Codes - Visual Basic*

| Error Name | Error Code | Description |
|---|---|---|
| `Global Const TerrNoError%` | &H00 | No error |
| `Global Const TerrBadChannel%` | &H01 | Specified LPT channel was out-of-range |
| `Global Const TerrNotOnLine%` | &H02 | Requested TempBook is not on-line |
| `Global Const TerrNoTempBook%` | &H03 | TempBook is not on the requested channel |
| `Global Const TerrBadAddress%` | &H04 | Bad function address |
| `Global Const TerrFIFOFull%` | &H05 | FIFO Full detected, possible data corruption |
| `Global Const TerrInvChan%` | &H10 | Invalid |
| `Global Const TerrInvCount%` | &H11 | Invalid count parameter |
| `Global Const TerrInvTrigSource%` | &H12 | Invalid trigger source parameter |
| `Global Const TerrInvGain%` | &H14 | Invalid channel gain parameter |
| `Global Const TerrInvPort%` | &H17 | Invalid port parameter |
| `Global Const TerrInvChip%` | &H18 | Invalid chip parameter |
| `Global Const TerrInvBitNum%` | &H1A | Invalid bit number parameter |
| `Global Const TerrInvClock%` | &H1B | Invalid clock parameter |
| `Global Const TerrInvTod%` | &H1C | Invalid time-of-day parameter |
| `Global Const TerrInvGateCtrl%` | &H20 | Invalid gate control parameter |
| `Global Const TerrInvOutputCtrl%` | &H21 | Invalid output control parameter |
| `Global Const TerrInvInterval%` | &H22 | Invalid interval parameter |
| `Global Const TerrTypeConflict%` | &H23 | An integer was passed to a function requiring a character |
| `Global Const TerrMultBackXfer%` | &H24 | A second background transfer |
| `Global Const TerrInvDiv%` | &H25 | Invalid Fout divisor |
| `Global Const TerrTCE_TYPE%` | &H26 | TC type out-of-range |
| `Global Const TerrTCE_TRANGE%` | &H27 | Temperature out-of-CJC-range |
| `Global Const TerrTCE_VRANGE%` | &H28 | Voltage out-of-TC-range |
| `Global Const TerrTCE_PARAM%` | &H29 | Unspecified parameter value error |
| `Global Const TerrTCE_NOSETUP%` | &H2A | `tbkTCConvert` called before `tbkTCSetup` |
| `Global Const TerrOverrun%` | &H2C | A buffer overrun occurred |
| `Global Const TerrZCInvParam` | &H2D | Invalid zero compensation parameter |
| `Global Const TerrZCNoSetup` | &H2E | `tbkZeroConvert` called before `tbkZeroSetup` |
| `Global Const TerrInvCalFile` | &H2F | Cannot open the specified calibration file |
| `Global Const TerrMemLock` | &H30 | Cannot lock allocated memory from Windows |
| `Global Const TerrMemHandle` | &H31 | Cannot get a memory handle from Windows |
| `Global Const TerrNoPreTActive` | &H32 | No pre-trigger configured |

# 32-Bit API Programming Models for TempBook 10

## Overview

The 32-bit Application Programming Interface (API) allows you to create custom software to satisfy your TempBook data acquisition requirements. Two chapters explain the 32-bit API: this chapter gives you the basic concepts needed to write effective programs, and chapter 11 describes the API functions in detail. This chapter explains *how to combine the API functions into useful routines* and is divided into three parts:

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

**Note**: The TempBook 32-bit API is a subset of the `DaqX` API which provides a common interface for 32-bit data acquisition applications (TempBook, WaveBook, DaqBook, DaqBoard, Daq PC-Card, etc). This document describes the commands that pertain to the TempBook.

## Data Acquisition Environment

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

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

### Application Programming Interface (API)

The API includes all the software functions needed for building a data acquisition system with the hardware described in this manual. Chapter 11 (*daqCommand Reference—32-bit 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.

### 32-bit vs 16-bit API

Major differences between the 32-bit and 16-bit APIs were described in the introductory chapter (*Programmer's Guide*). Language support varies as follows:

- The **32-bit** API accommodates C, Visual Basic, and Delphi.
- The **16-bit** API accommodates C, QuickBASIC, Visual Basic, and Turbo Pascal 7.

**Coding for the 32-bit and 16-bit API cannot be used together; 32-bit and 16-bit models are slightly different (this chapter is for the 32-bit API models; chapters 6 to 8 demonstrate examples using the 16-bit API).**

### Hardware Capabilities and Constraints

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

## Signal Environment

Several data acquisition concepts are listed here; you must apply these concepts as needed in your situation. These concepts include, but are not limited to:

- **Device and parameter identification**. Refer to the related reference tables in chapter 11.
- **Scan rates and sequencing**. With multiple scans, the time between scans becomes a parameter. This time can be a constant or can be dependent upon a trigger.
- **Triggering options**. Triggering starts the A/D conversion. The trigger can be an external analog or TTL trigger or a program-controlled software trigger. Refer to the trigger functions in chapter 11.
- **Foreground/background**. Foreground transfer routines require the entire transfer to occur before returning control to the application program. Background routines start the A/D acquisition and return control to the application program before the transfer occurs. Data is transferred while the application program is running. Data will be transferred to the user memory buffer during program execution in 1 sample or 2048 sample blocks, depending on the configuration. The programmer must determine what tasks can proceed in the background while other tasks perform in the foreground and how often the status of the background operations should be checked.

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

# Basic Models

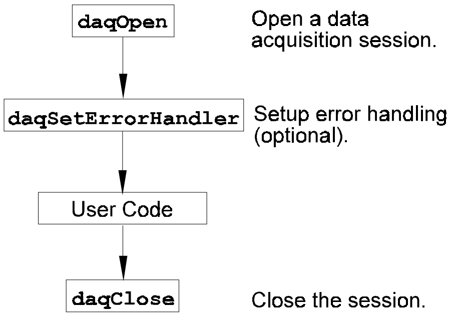
This section outlines basic programming steps commonly used for data acquisition. Consider the models as building blocks that can be put together in different ways or modified as needed. As a general tutorial, these examples use Visual Basic since most programmers know BASIC and can translate to other languages as needed. The 32-bit API programming models discussed in this chapter include:

| Model Type | Model Name | Page |
|---|---|---|
| **Configuration** | Initialization and Error Handling | 10-3 |
| **Acquisition** | Foreground Acquisition with One-Step Commands | 10-4 |
| | Temperature Acquisition Using One-Step Commands | 10-6 |
| | Counted Acquisition Using Linear Buffers | 10-8 |
| | Indefinite Acquisition, Direct-To-Disk Using Circular Buffers | 10-10 |
| | Multiple Hardware Scans, Software Triggering | 10-13 |
| | Background Acquisition | 10-15 |
| | Temperature Acquisition Using TC Conversion Functions | 10-17 |
| **Data Handling** | Double Buffering | 10-20 |
| | Direct-to-Disk Transfers | 10-22 |
| | Transfers With Driver-Allocated Buffers | 10-25 |

## 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&)`**


daqOpen
Open a data acquisition session.
daqSetErrorHandler
Setup error handling (optional).
User Code
daqClose
Close the session.

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

```
handle& = VBdaqOpen&(“TempBook0”)
ret& = VBdaqClose&(handle&)
```

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

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

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

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

```
If (VBdaqOpen&(“TempBook0”) <  0) Then
   “Cannot open “TempBook0”
```

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

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

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

```
handle& = VBdaqOpen&(“TempBook0”)
ret& = VBdaqSetErrorHandler&(handle&, 100)

    On Error GoTo ErrorHandler
```

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

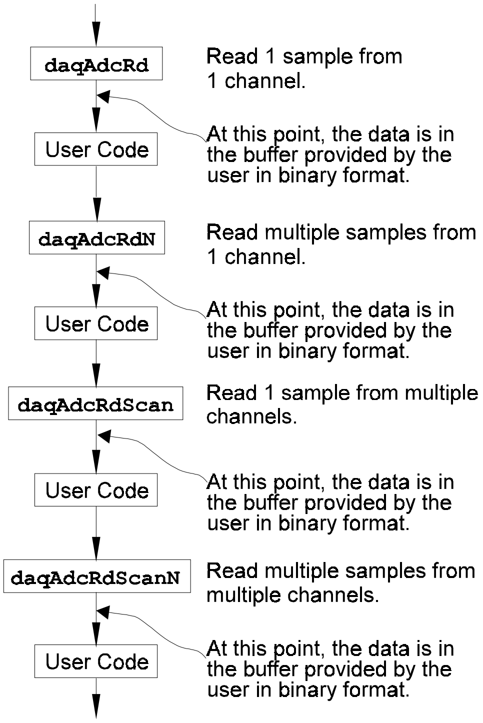
Daq* errors will send the program into the error handling routine. This is the error handler. Program control is sent here on error.

```
ErrorHandler:

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

   MsgBox errorString$, , "Error!"

End Sub
```

## Foreground Acquisition with One-Step Commands

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

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


daqAdcRd
Read 1 sample from 1 channel.
User Code
At this point, the data is in the buffer provided by the user in binary format.
daqAdcRdN
Read multiple samples from 1 channel.
User Code
At this point, the data is in the buffer provided by the user in binary format.
daqAdcRdScan
Read 1 sample from multiple channels.
User Code
At this point, the data is in the buffer provided by the user in binary format.
daqAdcRdScanN
Read multiple samples from multiple channels.
User Code
At this point, the data is in the buffer provided by the user in binary format.

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 TempBook 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
```

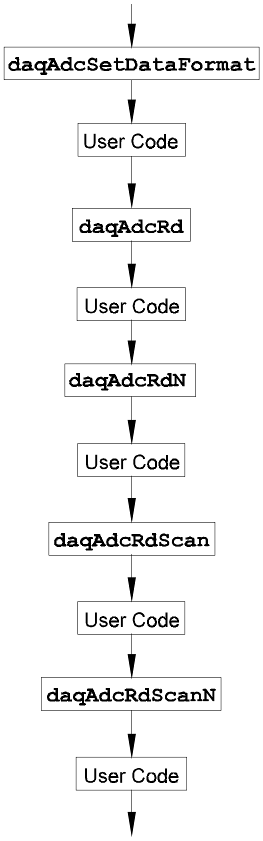
Finally, the program will collect 9 scans from channels 1 through 7 with an immediate trigger, then display the results.

```
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&)
```

## Temperature Acquisition Using One-Step Commands



This example demonstrates the 32-bit API's high-level one-step routines to read thermocouple input from the TempBook/66 device. These functions combine scan sequencer setup, ADC data collection, and thermocouple linearization. Functions used include:

- `VBdaqSetDefaultHandler&(handler&)`
- `VBdaqGetDeviceCount&(deviceCount&)`
- `VBdaqGetDeviceList&(deviceList&, deviceProps)`
- `VBdaqOpen&(daqName$)`
- `VBdaqAdcSetDataFormat&(handle&, rawFormat&, postProcFormat&)`
- `VBdaqAdcRd&(handle&,chan&, sample%, gain&, flags&)`
- `VBdaqAdcRdScan&( handle&,startChan&, endChan&, buf%, gain&, flags&)`
- `VBdaqAdcRdScanN&( handle&,chan&, buf%, scanCount&, triggerSource&, rising&, level%, freq!, gain&, flags&)`
- `VBdaqAdcRdN&(handle&,chan&, startChan&, endChan&, buf%, scanCount&, triggerSource&, rising&, level%, freq!, gain&, flags&)`
- `VBdaqClose&(handle&,daqEvent&)`

```
Const Scans& =  5000
Const Level% = 0
Const Rising& = 0
Const Start& = 0
Const End& = 0
Const Freq! = 6000.0
Const Gain& = TbkBiTypeJ&
Const Flags& =  DafBipolar& + DafDifferential& + DafTcTypeJ&
Const Chans& =  End& - Start&+1

Dim buf%(80), handle&, ret&, flags&
Dim I&, j&, deviceCount&, handle&, deviceIndex&
Dim temp%, temps%(Scans&*Chans&)
Dim sum!, totals!(Chans&)
Dim ret&
```

First, we need to open the TempBook/66 device. This example uses a device named “TempBook0”. The device name must be a valid device name for a configured device.

```
handle& = VBdaqOpen&(“TempBook0”)
```

Set the raw data format to native and the post-processing data format to temperature in tenths of a degree C. The post-processing data format controls the format of the data returned by the one-step acquisition functions such as **daqAdcRd**. In this case, **daqAdcRd** returns temperatures rather than raw ADC readings.

```
ret& = VBdaqAdcSetDataFormat& (handle&, DardfNative&, DappdfTenthsDegC&)
```

The following statement retrieves a single ADC sample from a type J thermocouple on channel **Start&** and converts the reading to a temperature. The temperature is returned in the **sample%** parameter and is a 16-bit quantity which represents tenths of degrees C.

```
ret& = VBdaqAdcRd& (handle&, Start&, temp%, TbkBiTypeJ, Flags&)
```

Now the results for the single-sample read are displayed:

```
Print "Results of daqAdcRd for channel “, Start&, “: (single reading)"
Print "Temperature: “, temp/10.0, “C"
```

The next statement retrieves **Scans&(5000)** number of samples from a type J thermocouple on channel **Start&** and converts the readings to a single temperature using block averaging. The trigger source is set to **DatsImmediate&** so the scans will be acquired immediately. Again, the temperatures are returned as 16-bit words representing tenths of degrees C.

```
ret& = VBdaqAdcRdN(handle&, Start&, temps%(), Scans&, DatsImmediate&,
 Rising&, Level&,  Freq!, TbkBiTypeJ&, Flags&)
```

The averaging of the temperature values into a single temperature value is performed through the following:

```
sum! = 0.0
For I& =0 To Scans&
   sum! = sum! + temps%(I&)
Next I&
sum! = sum!/Scans&
```

The averaged temperature can now be printed out.

```
Print "Results of daqAdcRdN: (",Scans&,"  readings averaged)"
Print "Channel",  Start&, " Temperature: ", sum!/10.0, "C"
```

Next, a single scan will be retrieved for a multiple channel scan configuration. The following statement configures the scan from the **`Start& (0)`** to the **`End& (0)`** and configures each channel as a type J thermocouple. The returned values for each channel will be placed in the **`temps%`** array and will be 16-bit words representing tenths of degrees C.

```
ret&  = VBdaqAdcRdScan& (handle&, Start&, End&, temps%(), TbkBiTypeJ&,
      Flags&)
Print "Result of daqAdcRdScan  (Single Readings) : "
For  I& = Start& To  End&
      Print "Channel ",  I&,"  Temperature ", temps%(I&)/10.0, " C "
Next I&
```

Finally, we will retrieve multiple scans for a multiple-channel scan configuration. The scan will be defined using type J thermocouples for each channel in the scan group configuration. The scan group will start at **`Start& (0)`** and go to **`End&  (0);`** and all channels will be type J thermocouples. The returned values will be placed in the **`temps%`** array and will be 16-bit words representing tenths of degrees C.

```
ret& = VBdaqAdcRdScanN& (handle& , Start&, End& , temps%, Scans&,
  DatsImmediate&, Rising&, Level&, Freq!, Gain&, Flags&)
```

The following code will average all samples collected for each channel and display the results.

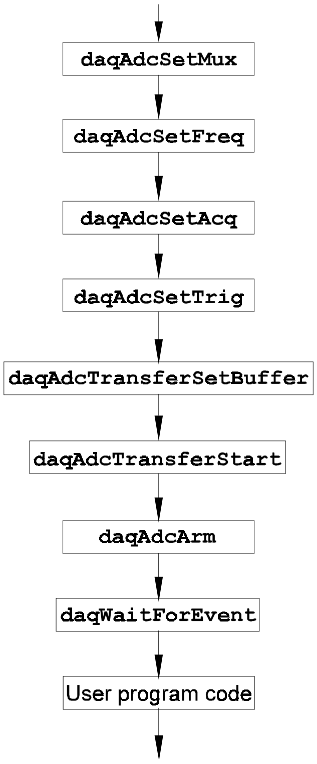
```
' Zero the totals
 For I& = Start& To End&+1
    totals!(I&) = 0.0
 Next I&

 ' Average the temperatures
 For I& = 0 To Scans&
    For j& = Start& To End&
       totals!(j&) = totals!(j&) + temps%(I&*Chans& + j&)
    Next j&
 Next I&

' Divide the totals
 For I& = 0 To Chans&
    totals!(I&) = totals!(I&)/Scans&
 Next I&

 ' Print "Results of daqAdcRdScanN: "
 For I& = 0  To Scans&
    Print "Channel", I&, " Temperature: ", totals!(I&)/10.0, "C"
 Next I&

 ' Close the TempBook/66
 ret& = VBdaqClose&(handle&)
```

## Counted Acquisitions Using Linear Buffers

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

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

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



Define a channel scan group.

Set the sampling frequency.

Configure a counted acquisition for 10 post-trigger scans.

Set the trigger event to be immediate.

Configure an ADC transfer data buffer to be 10 scans long and terminate once the end of the buffer is reached.

Initiate a transfer into the configured buffer.

Arm the acquisition. Since trigger source is immediate, the acquisition begins now.

Wait for the acquisition to complete.

Process the data.

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 TempBook/Daq* device to the defined user buffer which will begin after the trigger event is satisfied (upon the completion of the **daqAdcSoftTrig()** function call).

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

Now the trigger will start the transfer:

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

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

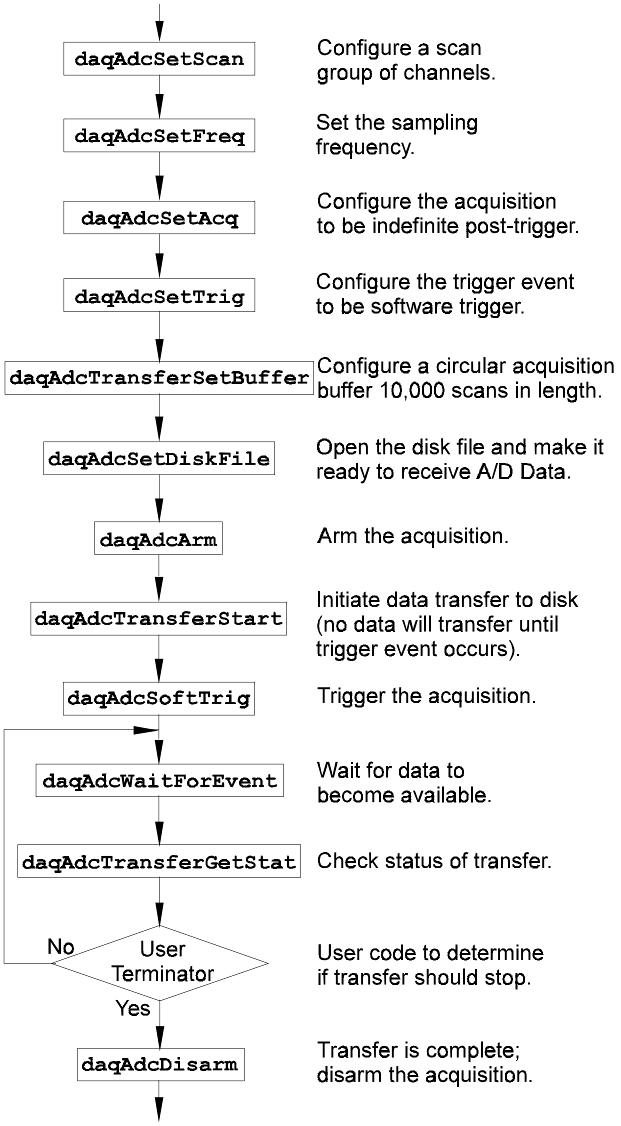
```
ret& = VBdaqWaitForEvent(handle&,DteAdcDone&)
```

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

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

## Indefinite Acquisition, Direct-To-Disk Using Circular Buffers

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


daqAdcSetScan
Configure a scan group of channels.
daqAdcSetFreq
Set the sampling frequency.
daqAdcSetAcq
Configure the acquisition to be indefinite post-trigger.
daqAdcSetTrig
Configure the trigger event to be software trigger.
daqAdcTransferSetBuffer
Configure a circular acquisition buffer 10,000 scans in length.
daqAdcSetDiskFile
Open the disk file and make it ready to receive A/D Data.
daqAdcArm
Arm the acquisition.
daqAdcTransferStart
Initiate data transfer to disk (no data will transfer until trigger event occurs).
daqAdcSoftTrig
Trigger the acquisition.
daqAdcWaitForEvent
Wait for data to become available.
daqAdcTransferGetStat
Check status of transfer.
No
User Terminator
Yes
User code to determine if transfer should stop.
daqAdcDisarm
Transfer is complete; disarm the acquisition.

Unlike the 16-bit API, the 32-bit API has built-in direct-to-disk functionality. Therefore, very little needs to be done by the application to configure direct-to-disk operations.

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

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

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

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

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

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

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

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

Next, set the internal sample rate to 3 kHz.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Loop While acqTermination& = 0
```

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

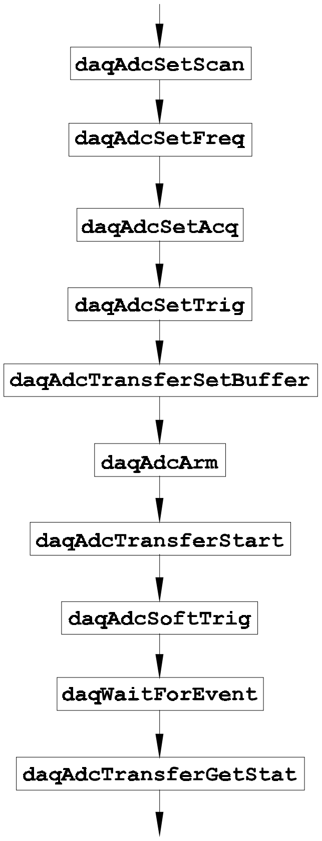
```
ret& = VBdaqAdcDisarm&(handle&)
```

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

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

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

## Multiple Hardware Scans, Software Triggering



This model takes multiple scans from several channels. The functions used here are of a lower level than the one-step functions, and more control is allowed over the acquisition. This program exemplifies this flexibility by individually configuring the channels and by explicitly setting up the transfer buffer.

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

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

This example will retrieve 10 scans for channels 0, 5, and 8, triggered via software with a 3000 Hz sampling frequency and unity gain. Functions used include:

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

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

```
Const freq! = 3000
Const scans& = 10
Const channels& = 3
Dim buf%(scans& * channels&)
Dim chans&(channels&), gains&(channels&), flags&(channels&)
```

Now set up the desired channels and their individual gains and flags.

```
chans&(0) = 0        ' high speed digital channel
chans&(1) = 5        ' analog channel 5
chans&(2) = 8        ' analog channel 8
   ' Channel gains and flags setting
   For i& = 0 To channels& - 1
      gains&(i&) = DgainX1&  ' unity gain
      flags&(i&) = DafAnalog& + DafSingleEnded& + DafUnipolar&
   Next i&
```

Open the device, and set up the error handler. For simplicity, the error handler is not defined explicitly. Refer to Example 1 for more information.

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC3
```

Now set the scan configuration:

```
ret& = VBdaqAdcSetScan&(handle&,chans&(), gains&(), 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&)
```

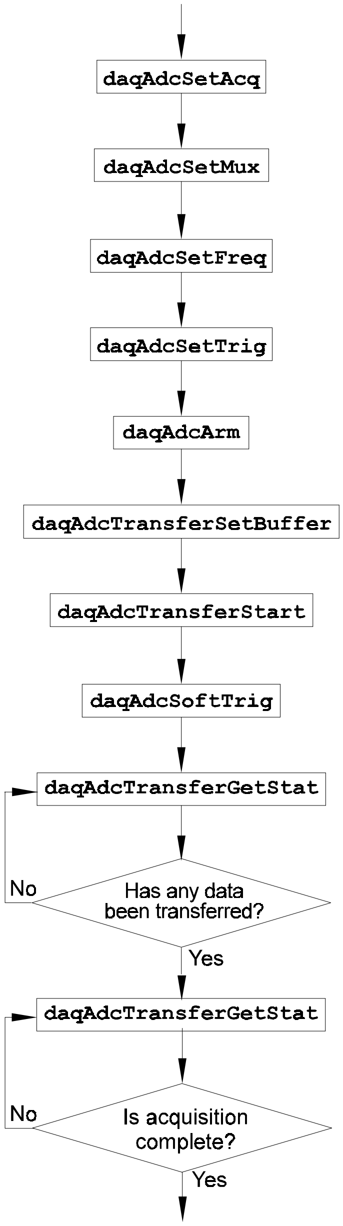
Finally, display the results and close the device.

```
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("TempBook0")`**
- **`VBdaqSetErrorHandler(handle&, 100)`**



The constants used are defined as follows:

```
Const channels& = 8
Const scans& = 9
Const freq# = 200
```

As usual, the device is opened and the error handler set up:

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC4
```

The acquisition is configured for 9 post-trigger scans and **`Nshot`** mode:

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

Set up the scan configuration for channels 1 to 9 with a gain of ×1:

```
ret& = VBdaqAdcSetMux&(handle&, 1, channels&,
  DgainX1&, 1)
```

Set the post-trigger scan rates:

```
ret& = VBdaqAdcSetFreq&(handle&, freq#)
```

Set the trigger source to a software trigger command; the other trigger parameters are not needed with a software trigger.

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

Arm the acquisition:

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

Now to set up the buffer for a background acquisition in update-single mode with cycle-mode off:

```
ret& = VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& +
  DatmUpdateSingle&)
```

Start the transfer, and trigger to begin transferring data:

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

These next few lines wait for the first data to be received, by checking the **`retCount`** value after calling **`daqAdcTransferGetStat()`**:

```
retCount& = 0
   While retCount& = 0
      ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
   Wend
```

With the same function, wait for the acquisition to complete:

```
While active& <> 0
      ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
Wend
Print "Acquisition complete:"; retCount&; "scans acquired."
```

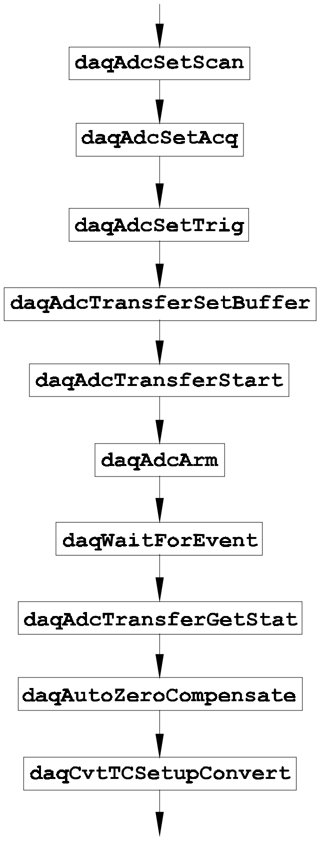
Now the data can be displayed or manipulated:

```
Print "Data acquired:"
   For i& = 0 To channels& - 1
      Print "Channel"; i& + 1; "Data:";
      For j& = 0 To scans& - 1
        Print Tab(j& * 7 + 17); buf%(j& * channels& + i&);
      Next j&
      Print
   Next i&
```

Finally, close the device:

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

## Temperature Acquisition Using TC Conversion Functions



This example demonstrates the general-purpose data-transfer functions coupled with TC-specific conversion routines. This method first configures the channel scan group for thermocouple input, then acquires the raw A/D data from the thermocouple input, and finally, converts the raw A/D data to temperature units in degrees C.

- `VBdaqOpen&(daqName$)`
- `VBdaqAdcSetFreq& (handle&, freq!)`
- `VBdaqAdcSetAcq& (handle&, mode&, preTrigCount&, postTrigCount&)`
- `VBdaqAdcSetScan& (handle&,   channels&(), gains&(),  flags&(),  chanCount&)`
- `VBdaqAdcSetClockSource&(handle&, clockSource&)`
- `VBdaqAdcSetTrig& (handle&, triggerSource&, rising&, level%, hysteresis%, channel&)`
- `VBdaqAdcTransferSetBuffer&(handle&, buf%(), scanCount&, transferMask&)`
- `VBdaqAdcTransferStart& (handle&)`
- `VBdaqAdcArm&(handle&)`
- `VBdaqWaitForEvent(handle&, event&)`
- `VBdaqAdcTransferGetStat& (handle&, active&, retCount&)`
- `VBdaqCvtTCSetupConvert(nscan&, cjcPosition&, ntc&, tcType&, bipolar&, avg&, counts%(), scans&, temp%(), ntemp&)`
- `VBdaqClose&(handle&)`

The following list defines the necessary constants and variables for temperature acquisition and conversion.

```
Const Scans& =  10
Const Level% = 0
Const Rising& = 0
Const Start& = 0
Const End& = 7
Const NumTcChans& = End& - Start& + 1
Const TotalChans& = NumTcChans + 3
Const TcGain& = TbkBiTypeJ&
Const CjcGain& = TbkBiCJC&
Const TcType& =  TbkTCTypeJ&
Const Freq! = 1000.0
Const AvgType& = 0
Const Gain& = TbkBiTypeJ&
Const Flag& =  DafUnsigned& + DafDifferential& + DafBiPolar&
Const Chans& =  End& - Start&+1

Dim       buf%(Scans& *  TotalScans&)
Dim       temp%(NumTcChans&)
Dim handle&
Dim       I&,j&, active&, retCount&
Dim Gains&(TotalChans&)
Dim Chans&(TotalChans&)
Dim Flags&(TotalChans&)
```

First, we need to open the TempBook/66 device. This example uses a device named “TempBook0”. The device name must be for a valid configured device.

```
 ‘ Open TempBook/66 device
handle& = VBdaqOpen&(“TempBook0”)
```

Next, build the channel scan group configuration. The following code sets up the configuration arrays that define the channel scan group. The channels are configured with the appropriate gain, mode and polarity definitions.

```
‘ Configure CJC and shorted channel
Chans[0] = 18    ‘ Shorted channel
Chans[1] = 18          ‘ Shorted channel
Chans[2] = 16          ‘ CJC channel

Gains[0] = TbkBiCJC&          ‘ Shorted channel at CJC gain
Gains[1] = TC_GAIN;      ‘  Shorted channel at thermocouple gain
Gains[2] = TbkBiCJC;          ‘ CJC channel reading at CJC gain

‘ All channels bipolar
Flags&[0] = Flag&
Flags&[1] = Flag&
Flags&[2] = Flag&

‘ Configure thermocouple channels after shorted and CJC channels in scan
For I& =0  To NumTcChans
 Chans&(I&+3) = Start& + I&;
 Gains&(I&+3) = TcGain&
 Flags&(i+3)  = Flag&
Next I&

‘ Configure the TempBook with the scan sequence
ret& = VbdaqAdcSetScan& (handle&, chans&(), gains&(), flags&(), TotalChans&)
```

Now the acquisition itself needs to be configured. This acquisition specifies a counted acquisition (**DaamNShot&**) with no pre-trigger and 10 post-trigger (**Scans&**) scans.

```
‘ Set the acquisition mode and clock source
ret& = VBdaqAdcSetAcq& (handle&, DaamNShot&, 0, Scans&)
ret& = VBdaqAdcSetClockSource& (handle&, DacsAdcClock&)
```

The acquisition trigger is set to immediate triggering (**DatsImmediate&**); however, the acquisition will not be started until it is armed later on.

```
  ‘  Configure to trigger immediately after armed
ret& = VBdaqAdcSetTrig& (handle&, DatsImmediate&, True, 0, 0, 0)
```

The scan frequency is now set to 1000 Hz (**Freq!**).

```
‘ Set the scan frequency
ret& = VBdaqAdcSetFreq& (handle&, Freq!)
```

Now that the acquisition has been configured, the transfer buffer for the raw data must be defined. The following routines will configure the **buf%** array as the raw data array with a length of 10 (**Scans&**) scans. Also, the buffer is set to linear mode by specifying the **DatmCycleOff&** flag. The **DatmUpdateSingle&** flag indicates that the buffer should be updated while each sample is acquired. After the buffer has been defined, the raw data transfer will be started with the **daqAdcTransferStart** routine. However, data transfer into the raw data buffer will not really begin until the acquisition has been triggered.

```
 ‘ Set up and start the transfer
ret& = VBdaqAdcTransferSetBuffer& (handle&, buf%(), Scans&, DatmCycleOff& +
DatmUpdateSingle&)
ret& = VBdaqAdcTransferStart& (handle&)
```

The acquisition is now configured, and the raw data buffer is ready to receive transferred data. To initiate the transfer, the acquisition needs to be armed. Once armed, the data transfer will begin immediately since the trigger source was configured as **DatsImmediate&**.

```
 ‘ Arm the acquisition
VBdaqAdcArm& (handle&)
```

The acquisition and transfer of the raw data is now active. The following statement can be used to wait for the termination of the transfer. Once the transfer is terminated, the raw data will then be present in the buffer.

```
‘ Wait until the acquisition is complete
ret& = VBdaqWaitForEvent& (handle&, DteAdcDone&)
‘ Get the number of scans acquired
ret& = VBdaqAdcTransferGetStat& (handle&, active&, retCount&)
Print retCount&,  “ Scans acquired. Converting data... "
```

The raw data is now available to be converted to temperature readings. The following code performs the conversion process on the **`buf%()`** raw data buffer. When complete, the temperature values will be available in the **`temps%()`** array. The **`VBdaqCvtTCSetupConvert`** function is used to perform the conversion. Here, the conversion is configured with the number of readings per scan (**`TotalChans&=11`**), CJC position of 2, the number of thermocouple channels (**`NumTcChans&=8`**), the thermocouple type (**`TcType&= TbkTCTypeJ&`**), bipolar raw data, no averaging (**`AvgType&=0`**), the raw data buffer (**`buf%()`**), the total number of scans (**`Scans&=10`**) and the target array for the converted temperature data in tenths of degrees C (**`temp%()`**).

```
‘ Configure the TC conversion functions to use zero correction
ret& = VBdaqAutoZeroCompensate& (1)

‘ Configure and Perform Thermocouple Linearization
ret& = VBdaqCvtTCSetupConvert& (TotalChans&, 2, NumTcChans&, TcType&,1,
  AvgType&,
buf%(), Scans&, temps%(), NumTcChans&)

‘ Print a channel column labels
Print "Averaged temperature readings:"

For I& = 0 To NumTcChans&
  Print  “Channel”, I&, “ Temperatue “, temp%(I&)/10.0,” C”
Next I&

‘ Close and exit
ret& = VBdaqClose& (handle&)
```

## Double Buffering

This example demonstrates using double buffering in the background mode, so that data can be read into one buffer while the another buffer can be processed in the foreground. Functions used include:

- VBdaqAdcArm&(handle&)
- VBdaqAdcBufferTransfer(buf1%(0), BLOCK&, 0, 0, 0, tmpActive&, tmpRetCount&)
- VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)
- VBdaqAdcSetFreq&(handle&, freq!)
- VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, DafAnalog&+DafUnipolar&)
- VBdaqAdcSetTrig(handle&, DatsSoftware&, rising&, level%, HYSTERESIS%, 1)
- VBdaqAdcSoftTrig&(handle&)
- VBdaqAdcTransferGetStat(handle&, active&, retCount&)
- VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&, DatmCycleOff& + DatmUpdateSingle&)
- VBdaqAdcTransferStart(handle&)
- VBdaqClose(handle&)
- VBdaqOpen("TempBook0")
- VBdaqSetErrorHandler(handle&, 100)

The following constants define the number of channels and other acquisition parameters:

```
Const channels& = 8
Const scans& = 20000
Const BLOCK& = 1000
Const freq! = 5000#
Const level% = 0
Const HYSTERESIS% = 0
Const rising& = 0
```

Dimension 2 buffers for double buffering:

```
Dim buf0%(channels& * BLOCK&)
Dim buf1%(channels& * BLOCK&)
```

Set error handler and initialize TempBook:

```
 handle& = VBdaqOpen("TempBook0")
 ret& = VBdaqSetErrorHandler(handle&,
100)
 On Error GoTo ErrorHandlerADC7
```

Set the acquisition to **NShot** on trigger and the post-trigger scan count:

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

Set the scan configuration for unity gain, from channels 1 to 8, in analog unipolar mode:

```
ret& = VBdaqAdcSetMux&(handle&, 1,
channels&, DgainX1&,
DafAnalog&+DafUnipolar&)
```

Set the post-trigger scan rate:

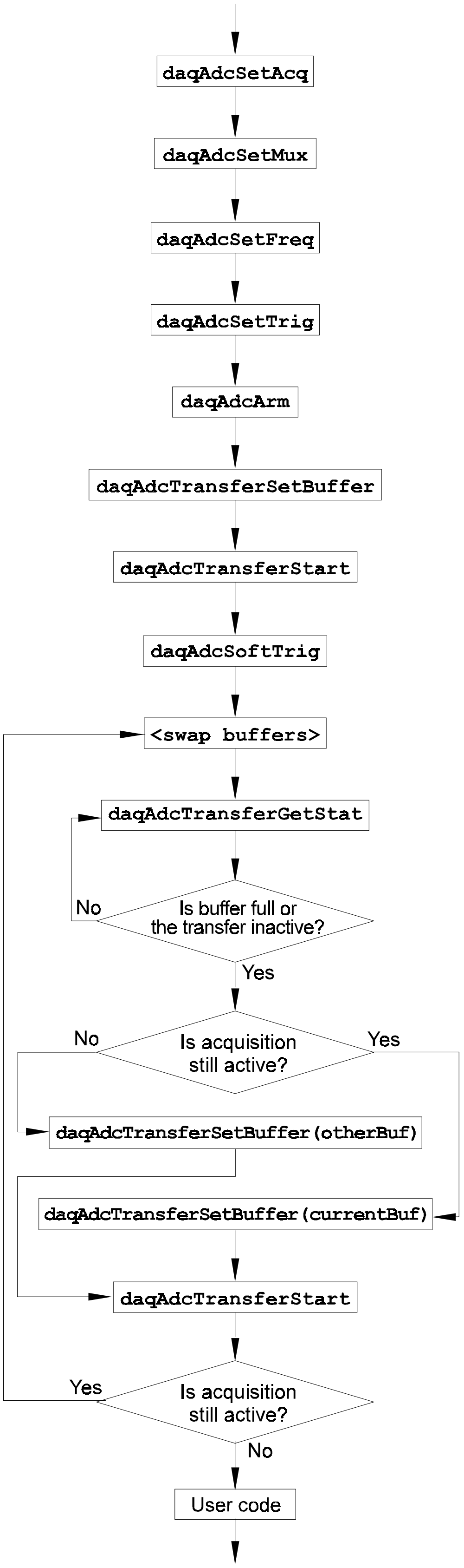
```
ret& = VBdaqAdcSetFreq&(handle&, freq!)
```

Set the trigger source to a software trigger command:

```
ret& = VBdaqAdcSetTrig(handle&,
DatsSoftware&, rising&, level%,
HYSTERESIS%, 1)
```

Arm the acquisition:

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



Set up the first buffer for BLOCK scans, with cycle mode off and update single on:

```
    ret& = VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&, DatmCycleOff& +
    DatmUpdateSingle&)
```

Start the first transfer; the transfer will actually start upon trigger detection. In this case, the following software trigger will start the transfer:

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

Issue a software trigger command to the hardware to trigger the transfer:

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

The next **do** loop swaps the active buffer back and forth from **buf0** to **buf1** and waits for the acquisition to go inactive or the buffer to fill up. Swapping continues until the transfer goes inactive:

```
      whichBuf& = 0
      Do
```

The following line changes the current buffer:

```
         If whichBuf& = 1 Then whichBuf& = 0 Else whichBuf& = 1
```

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

```
         Do
            ret& = VBdaqAdcTransferGetStat(handle&, active&, retCount&)
         Loop While ((active& <> 0) And (retCount& < BLOCK&))
```

If the previous acquisition is still active, start another transfer into the next buffer:

```
         If (active& <> 0) Then
            If whichBuf& = 0 Then
               ret& = VBdaqAdcTransferSetBuffer(handle&, buf0%(), BLOCK&,
    DatmCycleOff& + DatmUpdateSingle&)
               ret& = VBdaqAdcTransferStart(handle&)
```

Otherwise, restart the transfer into the current buffer:

```
     Else
               'ret& = VBdaqAdcBufferTransfer(buf1%(0), BLOCK&, 0, 0, 0,
                    tmpActive&, tmpRetCount&)
               ret& = VBdaqAdcTransferSetBuffer(handle&, buf1%(), BLOCK&,
                    DatmCycleOff& + DatmUpdateSingle&)
               ret& = VBdaqAdcTransferStart(handle&)
             End If
         End If
```

Send the data into the process buffer, **totals()**:

```
         If (retCount& > 0) Then
```

Average the readings in the process buffer and print the results:

```
            For j& = 0 To channels& - 1
               totals&(j&) = 0
            Next j&
            For i& = 0 To retCount& - 1
               For j& = 0 To channels& - 1
```

Decide which buffer to add the data from:

```
                  If whichBuf& = 0 Then
                     totals&(j&) = totals&(j&) + buf1%(i& * channels& + j&)
                  Else
                     totals&(j&) = totals&(j&) + buf0%(i& * channels& + j&)
                  End If
               Next j&
            Next i&
```

Display the averaged results:

```
            Print "Averages:";
            For j& = 0 To channels& - 1
               Print Tab(j& * 7 + 17); Format$((5# / 32768#) * totals&(j&) /
                  retCount&, "#0.000");
            Next j&
            Print
         End If
```

Continue the do..while loop until the acquisition goes inactive:

```
      Loop While (active& <> 0)
```

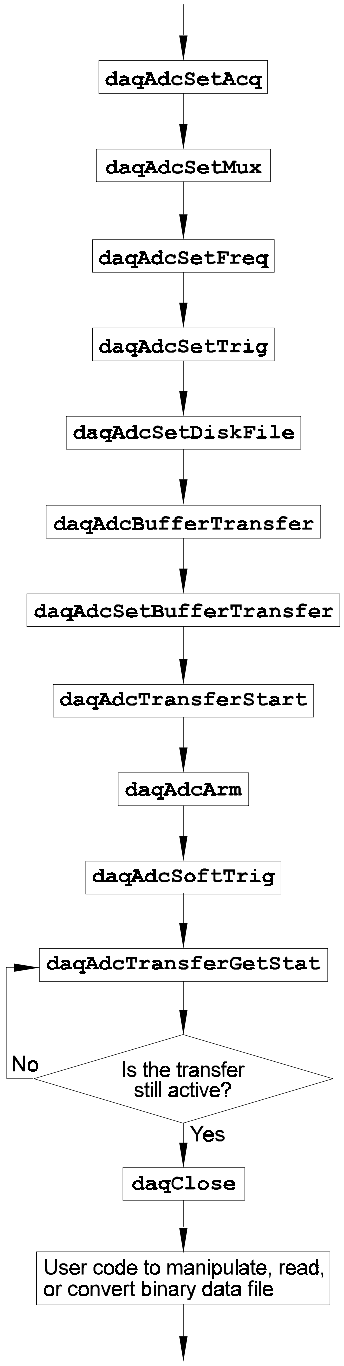
Close the device before exiting:

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

## Direct-to-Disk Transfers

This example takes multiple scans from multiple channels and writes them directly to disk in a packed-data format. Functions used are:

- `VBdaqAdcArm&(handle&)`
- `VBdaqAdcSetAcq&(handle&, DaamNShot, 0, scans&)`
- `VBdaqAdcSetDiskFile&(handle&, "adcex8.bin", DaomAppendFile&, 0)`
- `VBdaqAdcSetFreq&(handle&, freq!)`
- `VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, DafUniPolar&+DafAnalog&)`
- `VBdaqAdcSetTrig&(handle&, DatsSoftware&, DatdRisingEdge&, 0, HYSTERESIS%, 1)`
- `VBdaqAdcSoftTrig&(handle&)`
- `VBdaqAdcTransferGetStat&(handle&, active&, retCount&)`
- `VBdaqAdcTransferSetBuffer&(handle&, buf%(), BLOCK&, DatmCycleOn& + DatmUpdateBlock&)`
- `VBdaqAdcTransferStart&(handle&)`
- `VBdaqClose&(handle&)`
- `VBdaqCvtRawDataFormat&(buf%(), DacaUnpack, BLOCK&, channels&, scanCount&)`
- `VBdaqOpen&("TempBook0")`
- `VBdaqSetErrorHandler(handle&, 100)`



File handling in MS-Windows requires calls to the windows API, so the following constants are defined for use in those calls. For further information, see **`mapiwin.h`**.

```
Const GENERIC_READ& = &H80000000
Const OPEN_EXISTING = 3
Const FILE_ATTRIBUTE_NORMAL& = &H80
Const OPEN_ALWAYS = 4
Const CREATE_ALWAYS = 2
```

Also define the usual constants defining scan parameters and some declarations for file manipulation:

```
Const channels& = 2
Const scans& = 800
Const freq! = 200#
Const BLOCK& = 200     ' CHANNELS& * BLOCK& must be
  a multiple of 4
Const HYSTERESIS% = 0
Dim buf%(channels& * BLOCK&)
Dim fileHandle&
Dim byteCount&, wordCount&, sampleCount&,
  scanCount&
Dim binFile$
```

First set the name of the file to be used for the acquisition:

```
binFile = "adcex8.bin"
```

Open the device, and set the error handler:

```
handle& = VBdaqOpen&("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC8
```

Set the acquisition to **`NShot`** on trigger and the post-trigger scan count:

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

Set the scan configuration for channels 1 to 8 with a gain of ×1 in unipolar analog mode:

```
    ret& = VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&,
  DafUniPolar&+DafAnalog&)
```

Set the post-trigger scan frequency:

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

Set the trigger source to a software trigger command; the rest of the parameters have no effect on a software trigger:

```
      ret& = VBdaqAdcSetTrig&(handle&, DatsSoftware&, DatdRisingEdge&, 0,
    HYSTERESIS%, 1)
```

Set the direct-to-disk filename with no pre-write, in append mode; also available is:

```
      ret& = VBdaqAdcSetDiskFile&(handle&, "adcex8.bin", DaomAppendFile&, 0)
```

Start reading data in the background mode with cycle mode on and updateBlock:

```
   ret& = VBdaqAdcTransferSetBuffer&(handle&, buf%(), BLOCK&, DatmCycleOn& +
    DatmUpdateBlock&)
   ret& = VBdaqAdcTransferStart&(handle&)
   ret& = VBdaqAdcArm&(handle&)
   ret& = VBdaqAdcSoftTrig&(handle&)
```

Monitor the progress of the transfer:

```
   active& = -1
      While active& <> 0
         ret& = VBdaqAdcTransferGetStat&(handle&, active&, retCount&)
      Wend
      Print "Acquisition complete:"; retCount&; "scans acquired."
```

Close the device:

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

Now we convert the binary file to a text file. There is no simple way to do this, so it is necessary to open the file and manipulate it by hand.

First, open the binary file:

```
   Open "adcex8.bin" For Input As 1
```

Next, get a handle for the file; this is one of the windows API calls, **CreateFile** (it doesn't actually create anything, however).

```
   fileHandle& = CreateFile(binFile, GENERIC_READ, &H1, "", CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, "")
```

Now open the text output file where the converted data will be written:

```
   Open "adcex8.txt" For Output As 2
```

Next, actually convert the binary data to text:

```
      Do
```

Convert BLOCK unpacked scans to packed bytes:

```
         scanCount& = BLOCK&
         sampleCount& = scanCount& * channels&
         wordCount& = sampleCount& * 3 / 4
         byteCount& = 2 * wordCount&
```

Read the packed bytes from the input file, and get the number of bytes actually read. The **UBound()** and **Lbound()** functions just return the upper and lower bounds of the buffer. **Get #1** retrieves data from the file and stores it in the **buf()** array.

```
         Dim sz&
         sz& = UBound(buf%) - LBound(buf%)
         For i& = 0 To sz&
           Get #1, i&, buf(i&)
          Next i&
           byteCount& = sz
```

Write the scans read and unpacked to the text file

```
         For i& = 0 To scanCount& - 1
            For j& = 0 To channels& - 1
```

Send a tab between channels and a newline after each scan:

```
               If (j& < channels& - 1) Then
                  termChar$ = Chr$(9)
               Else
                  termChar$ = Chr$(13) + Chr$(10)
               End If
```

Calculate and write out the voltage value:

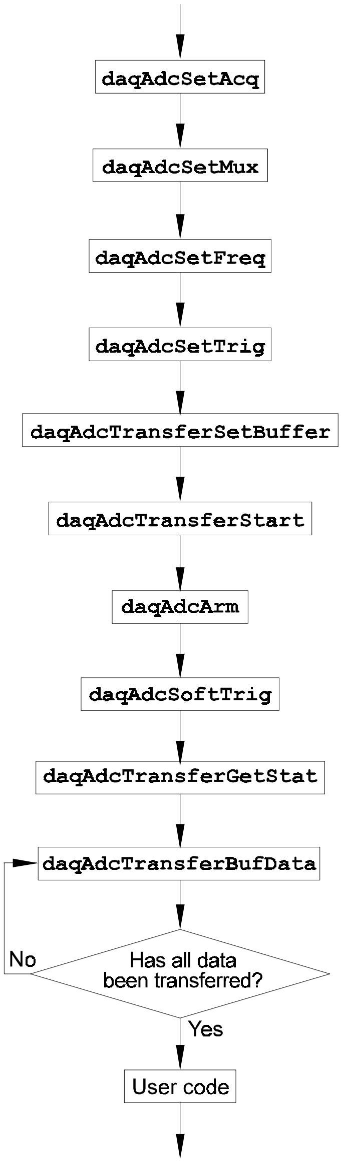
```
               voltage! = buf%(i& * channels& + j&) * 5! / 32768!
               Print #2, Format$(voltage!, ".000") + termChar$;
            Next j&
         Next i&
```

Print something so the program does not appear to be locked:

```
         Print ".";
      Loop While (byteCount& > 0)    ' A byteCount of 0 indicates end-of-file
      ' Close the input and output files
      Close 1
      Close 2
      Print "complete."
```

After program execution: data has been collected directly to disk in a binary file format, the TempBook device closed, the binary file was then opened, the data unpacked, and then written to a text file.

## Transfers With Driver-Allocated Buffers



This example demonstrates the use of the new **daqAdcTransferBufData()** function. The following program reads scans of multiple channels in the background mode and uses a software trigger to start the acquisition. Functions used include:

- `VBdaqAdcArm&(handle&)`
- `VBdaqAdcSetAcq&(handle&, DaamNShot&, 0, scans&)`
- `VBdaqAdcSetFreq&(handle&, freq#)`
- `VBdaqAdcSetMux&(handle&, 1, channels&, DgainX1&, 1)`
- `VBdaqAdcSetTrig&(handle&, DatsSoftware&, 0,0,0,0)`
- `VBdaqAdcSoftTrig&(handle&)`
- `VBdaqAdcTransferBufData(handle&, userBuf(0), 1, DatmWait , retVal)`
- `VBdaqAdcTransferGetStat(handle, active, retCount);`
- `VBdaqAdcTransferSetBuffer(handle&, buf%(), scans&, DatmCycleOff& + DatmUpdateSingle&)`
- `VBdaqAdcTransferStart(handle&)`
- `VBdaqClose(handle&)`
- `VBdaqOpen("TempBook0")`
- `VBdaqSetErrorHandler(handle&, 100)`

The constants used are defined as follows:

```
Const channels& = 8
Const scans& = 9
Const freq# = 200
```

As usual, the device is opened and the error handler is set up:

```
handle& = VBdaqOpen("TempBook0")
ret& = VBdaqSetErrorHandler(handle&, 100)
On Error GoTo ErrorHandlerADC4
```

The acquisition is configured for 9 post-trigger scans and **Nshot** mode:

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

Set up the scan configuration for channels 1 to 9 with a gain of ×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 32-bit API Functions*

The following table organizes the 32-bit API functions by type and includes a brief description.

## Simple One-Step Routines

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

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

## Complex A/D Scan Group Configuration Routines

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

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

## Trigger Options

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

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

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

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

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

## Multiple Scan Timing

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

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

## A/D Acquisition

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

| | |
|---|---|
| `daqAdcArm` | Arm an A/D acquisition using the current configuration. If the trigger source was set to be immediate, the acquisition will be triggered immediately. |
| `daqAdcDisarm` | Disarm the current acquisition if one is active. This command will disarm the current acquisition and terminate any current A/D transfers. |
| `daqAdcSetAcq` | Define the mode of the acquisition and set the pre-trigger and post-trigger acquisition counts, if applicable. |

## A/D Data Transfer

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

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

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

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

Notes

# daqCommand Reference, 32-Bit 11

## Overview

The first part of this chapter describes the TempBook driver commands for Windows95 and WindowsNT in 32-bit mode. Do not confuse 32-bit API with 16-bit API. The first table lists the commands by their function types as defined in the driver header files. Then, the prototype commands are described in alphabetical order as indexed below.

**Note**: The TempBook API is a subset of the Daq* API which also applies to other products; only TempBook-related commands are discussed in this document.

Beginning on page 11-35, several reference tables define parameters for: event-handling definitions, hardware definitions, ADC trigger-source and miscellaneous definitions, WBK card definitions, the API error codes, etc.

| Function | Description | Page |
|---|---|---|
| **Device Initialization Prototypes** | | |
| `daqOpen` | Open a session with the Daq* (including TempBook) | 11-28 |
| `daqClose` | End communication with the Daq* (including TempBook) | 11-20 |
| `daqOnline` | Check online status of the Daq* (including TempBook) | 11-28 |
| `daqGetDeviceCount` | Return the number of currently configured devices | 11-26 |
| `daqGetDeviceList` | Return the list of currently configured devices | 11-26 |
| `daqGetDeviceProperties` | Return the properties of specified device | 11-27 |
| **Error Handler Function Prototypes** | | |
| `daqSetDefaultErrorHandler` | Set the default error handler | 11-29 |
| `daqSetErrorHandler` | Specify a user defined routine to call when an error occurs in any command | 11-29 |
| `daqProcessError` | Process a driver defined error condition | 11-29 |
| `daqGetLastError` | Return the last logged error condition | 11-28 |
| `daqDefaultErrorHandler` | Call the default error handler | 11-25 |
| `daqFormatError` | Return text string for specified error | 11-26 |
| **Event Handling Function Prototypes** | | |
| `daqSetTimeout` | Set the time-out value for the Daq* operation (including TempBook) | 11-30 |
| `daqWaitForEvent` | Wait for specified Daq* device event (including TempBook) | 11-32 |
| `daqWaitForEvents` | Wait for multiple specified Daq* device events (including TempBook) | 11-32 |
| **Utility Function Prototypes** | | |
| `daqGetDriverVersion` | Return the software version | 11-27 |
| `daqGetHardwareInfo` | Return the hardware version | 11-27 |
| **Expansion Configuration Prototypes** | | |
| `daqSetOption` | Set options for a device's channel/signal path configuration | 11-30 |
| **Custom ADC Acquisition Prototypes - Scan Sequence** | | |
| `daqAdcSetMux` | Configure a scan specifying start and end channels | 11-13 |
| `daqAdcSetScan` | Configure up to 256 channels making up an A/D or HS digital input scan | 11-14 |
| `daqAdcGetScan` | Read the current scan configuration | 11-5 |
| **Custom ADC Acquisition Prototypes - Trigger** | | |
| `daqAdcCalcTrig` | Calculate the trigger level and trigger source for an analog trigger | 11-4 |
| `daqAdcSetTrig` | Configure an A/D trigger | 11-15 |
| `daqAdcSoftTrig` | Save a software trigger command to the DaqBook/DaqBoard | 11-16 |
| **Custom ADC Acquisition Prototypes - Scan Rate and Source** | | |
| `dacAdcSetRate` | Configure the ADC scan rate with the `mode` parameter | 11-13 |
| `daqAdcSetFreq` | Configure the pacer clock frequency in Hz | 11-12 |
| `daqAdcGetFreq` | Read the current pacer clock frequency | 11-5 |
| **Custom ADC Acquisition Prototypes - Scan Count, Rate and Source** | | |
| `daqAdcSetAcq` | Set acquisition configuration information | 11-10 |
| **Custom ADC Acquisition Prototypes - Direct-to-Disk** | | |
| `daqAdcSetDiskFile` | Specify the disk file for direct-to-disk transfers | 11-12 |
| **Custom ADC Acquisition Prototypes - Acquisition Control** | | |
| `daqAdcArm` | Arm an acquisition | 11-2 |
| `daqAdcDisarm` | Disarm an acquisition | 11-4 |
| **Custom ADC Acquisition Prototypes - Data Transfer without Buffer Allocation** | | |
| `daqAdcTransferBufData` | Transfer scans from driver-allocated buffer to user-specified buffer | 11-16 |
| `daqAdcTransferSetBuffer` | Setup a destination buffer for an ADC transfer | 11-18 |
| `daqAdcTransferStart` | Start an ADC transfer | 11-19 |

| Function | Description | Page |
|---|---|---|
| `daqAdcTransferGetStat` | Retrieve status of an ADC transfer | 11-17 |
| `daqAdcTransferStop` | Stop an ADC transfer | 11-19 |
| **Custom ADC Acquisition Prototypes - Buffer Manipulation** | | |
| `daqAdcBufferRotate` | Reorganize a circular buffer so that oldest data is oriented towards the front | 11-3 |
| **One-Step ADC Acquisition Prototypes** | | |
| `daqAdcRd` | Configure an A/D acquisition and read one sample from a channel | 11-6 |
| `daqAdcRdScan` | Configure an A/D acquisition and read one scan | 11-8 |
| `daqAdcRdN` | Configure an A/D acquisition and read multiple scans from a channel | 11-7 |
| `daqAdcRdScanN` | Configure an A/D acquisition and read multiple scans | 11-9 |
| **Data Format and Conversion Prototypes** | | |
| `daqAdcSetDataForma` | Set the raw and the post-acquisition data formats | 11-11 |
| `daqCvtRawDataFormat` | Convert raw data to a specified format | 11-20 |
| `daqCvtSetAdcRange` | Set the ADC Voltage Range for the conversion routines | 11-21 |
| **Zero Offset Prototypes** | | |
| `daqZeroSetup` | Configure data for zero compensation | 11-33 |
| `daqZeroConvert` | Perform zero compensation on one or more scans | 11-33 |
| `daqZeroSetupConvert` | Perform both the setup and convert steps with one call | 11-30 |
| `daqAutoZeroCompensate` | Configure the thermocouple linearization functions to automatically perform zero compensation | 11-19 |
| **Thermocouple Conversion Prototypes** | | |
| `daqCvtTCConvert` | Convert raw A/D readings from thermocouples to temperature readings | 11-22 |
| `daqCvtTCSetup` | Set up parameters for subsequent thermocouple temperature conversions | 11-24 |
| `daqCvtTCSetupConvert` | Set up and convert raw A/D readings from thermocouples into temperature readings | 11-25 |
| **Test Prototypes** | | |
| `daqTest` | Perform a specified test on a Daq* device | 11-31 |

## *Commands in Alphabetical Order*

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

### daqAdcArm

| | |
|---|---|
| **DLL Function** | `daqAdcArm(DaqHandleT handle);` |
| **C** | `daqAdcArm(DaqHandleT handle);` |
| **Visual BASIC** | `VBdaqAdcArm&(ByVal handle&)` |
| **Delphi** | `daqAdcArm(handle:DaqHandleT)` |
| **Parameters** | |
| `handle` | Handle to the device to which configured ADC acquisition is to be armed |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcDisarm` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

| | |
|---|---|
| **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 11-39) |
| **See Also** | `daqAdcTransferGetStat, daqAdcTransferSetBuffer` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

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

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

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

**Note: `daqAdcBufferRotate`** only works on unpacked samples.

## daqAdcCalcTrig

| | |
|---|---|
| **DLL Function** | `daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);` |
| **C** | `daqAdcCalcTrig(DaqHandleT handle, BOOL bipolar, FLOAT gainVal, FLOAT voltageLevel, PWORD triggerLevel);` |
| **Visual BASIC** | `VBdaqAdcCalcTrig&(ByVal handle&, ByVal bipolar&, ByVal gainVal!, ByVal voltageLevel!, triggerLevel%)` |
| **Delphi** | `daqAdcCalcTrig(handle:DaqHandleT; bipolar:longbool; gainVal:single; voltageLevel:single; var triggerLevel:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which the trigger level is to be calculated |
| `bipolar` | A flag that should be non-zero if the trigger channel is bipolar, or zero if it is unipolar |
| `gainVal` | A gain value of the trigger channel |
| `voltageLevel` | Voltage level to trigger at. |
| `triggerLevel` | Returned count to program the trigger using the daqAdcSetTrig function |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetTrig` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

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

The **`voltageLevel`** defines the analog voltage at which the Daq* will trigger. The setpoint must be within the valid input range of the trigger channel. For example, the setpoint range for a bipolar channel with unity gain would be 0 to 10 V (for ×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 11-39) |
| **See Also** | `daqAdcArm` |
| **Program References** | None |
| **Used With** | All devices |

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

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

## daqAdcGetFreq

| | |
|---|---|
| **DLL Function** | `daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);` |
| **C** | `daqAdcGetFreq(DaqHandleT handle, PFLOAT freq);` |
| **Visual BASIC** | `VBdaqAdcGetFreq&(ByVal handle&, freq!)` |
| **Delphi** | `daqAdcGetFreq(handle:DaqHandleT; var freq:single)` |
| **Parameters** | |
| `handle` | Handle to the device for which to get the current frequency setting |
| `freq` | A variable to hold the currently defined sampling frequency in Hz<br>Valid values: `100000.0 - 0.0002` |
| **Returns** | `DerrNoError` - No errors (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetFreq, daqAdcSetClock` |
| **Program References** | None |
| **Used With** | All devices |

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

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

## daqAdcGetScan

| | |
|---|---|
| **DLL Function** | `daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);` |
| **C** | `daqAdcGetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, PDWORD chanCount);` |
| **Visual BASIC** | `VBdaqAdcGetScan&(ByVal handle&, channels&(), gains&(), flags&(), chanCount&)` |
| **Delphi** | `daqAdcGetScan( handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:PDWORD )` |
| **Parameters** | |
| `handle` | Handle to the device for which to get the current scan configuration. |
| `channels` | An array to hold up to 512 channel numbers or 0 if the channel information is not desired. |
| `*gains` | An array to hold up to 512 gain values or 0 if the channel gain information is not desired |
| `flags` | Channel configuration flags in the in the form of a bit mask |
| `chanCount` | A variable to hold the number of values returned in the chans and gains arrays |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetScan, daqAdcSetMux` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqAdcRd

| | |
|---|---|
| **DLL Function** | `daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);` |
| **C** | `daqAdcRd(DaqHandleT handle, DWORD chan, PWORD sample, DaqAdcGain gain, DWORD flags);` |
| **Visual BASIC** | `VBdaqAdcRd&(ByVal handle&, ByVal chan&, sample%, ByVal gain&, ByVal flags&)` |
| **Delphi** | `daqAdcRd(handle:DaqHandleT; chan:DWORD; var sample:WORD; const gain:DaqAdcGain; flags:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which the ADC reading is to be acquired |
| `chan` | A single channel number |
| `sample` | A pointer to a value where an A/D sample is stored. Valid values: (See **`daqAdcSetTag`**) |
| `gain` | The channel gain |
| `flags` | Channel configuration flags in the form of a bit mask |
| **Returns** | `DerrFIFOFull` - Buffer Overrun<br>`DerrInvGain` - Invalid gain<br>`DerrInvChan` - Invalid channel<br>`DerrNoError` - No Error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetMux, daqAdcSetTrig, daqAdcSoftTrig` |
| **Program References** | DACEX.PAS (Delphi) |
| **Used With** | All devices |

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

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

| 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<br>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<br>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<br>`DerrInvGain` -Invalid gain<br>`DerrIncChan` - Invalid channel<br>`DerrInvTrigSource` - Invalid trigger<br>`DerrInvLevel` - Invalid level (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetFreq, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

# daqAdcRdScan

| | |
|---|---|
| **DLL Function** | `daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);` |
| **C** | `daqAdcRdScan(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DaqAdcGain gain, DWORD flags);` |
| **Visual BASIC** | `VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)` |
| **Delphi** | `daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device from which the ADC scan is to be acquired |
| `startChan` | The starting channel of the scan group |
| `endChan` | The ending channel of the scan group |
| `buf` | An array where the A/D scans will be placed |
| `gain` | The channel gain |
| `flags` | Channel configuration flags in the form of a bit mask. |
| **Returns** | `DerrInvGain` - Invalid gain<br>`DerrInvChan` -Invalid channel<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcRdNScan, daqAdcSetMux, daqAdcSetClock, daqAdcSetTrig` |
| **Program References** | DACEX.PAS (Delphi) |
| **Used With** | All devices |

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

# daqAdcRdScanN

| | |
|---|---|
| **DLL Function** | `daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);` |
| **C** | `daqAdcRdScanN(DaqHandleT handle, DWORD startChan, DWORD endChan, PWORD buf, DWORD scanCount, DaqAdcTriggerSource triggerSource, BOOL rising, WORD level, FLOAT freq, DaqAdcGain gain, DWORD flags);` |
| **Visual BASIC** | `VBdaqAdcRdScanN&(ByVal handle&, ByVal startChan&, ByVal endChan&, buf%(), ByVal scanCount&, ByVal triggerSource&, ByVal rising&, ByVal level%, ByVal freq!, ByVal gain&, ByVal flags&)` |
| **Delphi** | `daqAdcRdScanN(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; buf:PWORD; scanCount:DWORD; triggerSource:DaqAdcTriggerSource; rising:longbool; level:WORD; freq:single; const gain:DaqAdcGain; flags:DWORD)` |
| **Parameters** | |
| `handle` | Handle  to the device from which ADC scans are to be acquired |
| `startchan` | The starting channel of the scan group  (see table at end of chapter) |
| `endchan` | The ending channel of the scan group  (see table at end of chapter) |
| `buf` | An array where the A/D scans will be placed |
| `scanCount` | The number of scans to be read<br>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<br>Valid values: `0 -4095` |
| `freq` | The sampling frequency in Hz<br>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<br>`DerrInvChan` -Invalid channel<br>`DerrInvTrigSource` - Invalid trigger<br>`DerrInvLevel` - Invalid Level<br>`DerrFIFOFull` -Buffer Overrun<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcRd, daqAdcRdN, daqAdcRdScan,  daqAdcSetClock, daqAdcSetTrig` |
| **Program References** | None |
| **Used With** | All devices |

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

# daqAdcSetAcq

| | |
|---|---|
| **DLL Function** | `daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);` |
| **C** | `daqAdcSetAcq(DaqHandleT handle, DaqAdcAcqMode mode, DWORD preTrigCount, DWORD postTrigCount);` |
| **Visual BASIC** | `VBdaqAdcSetAcq&(ByVal handle&, ByVal mode&, ByVal preTrigCount&, ByVal postTrigCount&)` |
| **Delphi** | `daqAdcSetAcq(handle:DaqHandleT; mode:DaqAdcAcqMode; preTrigCount:DWORD; postTrigCount:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which the ADC acquisition is to be configured |
| `mode` | Selects the mode of the acquisition |
| `preTrigCount` | Number of pre-trigger ADC scans to be collected |
| `postTrigCount` | Number of post-trigger ADC scans to be collected |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcArm, daqAdcDisarm, daqAdcSetTrig` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

Acquisition modes can be defined as follows:

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

| 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 11-39) |
| **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. The TempBook/66 supports this packed format (used with the generic functions of the form **daqAdcTransfer...**).

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

## daqAdcSetDiskFile

| | |
|---|---|
| **DLL Function** | `daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);` |
| **C** | `daqAdcSetDiskFile(DaqHandleT handle, LPSTR filename, DaqAdcOpenMode openMode, DWORD preWrite);` |
| **Visual BASIC** | `VBdaqAdcSetDiskFile&(ByVal handle&, ByVal filename$, ByVal openMode&, ByVal preWrite&)` |
| **Delphi** | `daqAdcSetDiskFile(handle:DaqHandleT; filename:PChar; openMode:DaqAdcOpenMode; preWrite:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which direct to disk ADC acquisition is to be performed. |
| `filename` | String representing the path and name of the file to place the raw ADC acquisition data. |
| `openMode` | Specifies how to open the file for writing |
| `preWrite` | Specifies the amount to pre-write(in bytes) the file |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcTransferGetStat, daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

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

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

## daqAdcSetFreq

| | |
|---|---|
| **DLL Function** | `daqAdcSetFreq(DaqHandleT handle, FLOAT freq);` |
| **C** | `daqAdcSetFreq(DaqHandleT handle, FLOAT freq);` |
| **Visual BASIC** | `VBdaqAdcSetFreq&(ByVal handle&, ByVal freq!)` |
| **Delphi** | `daqAdcSetFreq(handle:DaqHandleT; freq:single)` |
| **Parameters** | |
| `handle` | Handle to the device for which the ADC acquisition frequency is to be set. |
| `freq` | The sampling frequency in Hz<br>Valid values: `100000.0 - 0.0002` |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcGetFreq, daqAdcSetClockSource` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqAdcSetMux

| DLL Function | `daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);` |
|---|---|
| C | `daqAdcSetMux(DaqHandleT handle, DWORD startChan, DWORD endChan, DaqAdcGain gain, DWORD flags);` |
| Visual BASIC | `VBdaqAdcSetMux&(ByVal handle&, ByVal startChan&, ByVal endChan&, ByVal gain&, ByVal flags&)` |
| Delphi | `daqAdcSetMux(handle:DaqHandleT; startChan:DWORD; endChan:DWORD; const gain:DaqAdcGain; flags:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which to configure the ADC channel scan group |
| `startChan` | The starting channel of the scan group |
| `endChan` | The ending channel of the scan group |
| `gain` | The gain value for all channels |
| `flags` | Channel configuration flags in the form of a bit mask |
| **Returns** | `DerrInvGain` - Invalid gain<br>`DerrIncChan` - Invalid channel<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetScan, daqAdcGetScan` |
| **Program References** | DACEX1.C, DAQEX.FRM (VB) |
| **Used With** | All devices |

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

## daqAdcSetRate

| DLL Function | `daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);` |
|---|---|
| C | `daqAdcSetRate(DaqHandleT handle, DaqAdcRateMode mode, DaqAdcAcqState acqState, FLOAT reqRate, PFLOAT actualRate);` |
| Visual BASIC | `VBdaqAdcSetRate(ByVal handle&, ByVal mode&, ByVal acqState&, ByVal reqRate!, actualRate!);` |
| Delphi | `daqAdcSetRate(handle: DaqHandleT; mode: DaqAdcRateMode, acqState: DaqAdcAcqState; reqRate:FLOAT; actualRate:PFLOAT);` |
| **Parameters** | |
| `handle` | Handle to the device for which to set ADC scanning frequency. |
| `mode` | Specifies the rate mode (frequency or period). |
| `acqState` | Specifies the acquisition state to which the rate is to be applied. |
| `reqRate` | Specifies the requested rate. |
| `actualRate` | Returns the actual rate applied. This may be different from the requested rate. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetAcq, daqAdcSetTrig, daqAdcArm, daqAdcSetFreq, daqAdcGetFreq` |
| **Program References** | |
| **Used With** | All devices |

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

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

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

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

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

# daqAdcSetScan

| | |
|---|---|
| **DLL Function** | `daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);` |
| **C** | `daqAdcSetScan(DaqHandleT handle, PDWORD channels, DaqAdcGain *gains, PDWORD flags, DWORD chanCount);` |
| **Visual BASIC** | `VBdaqAdcSetScan&(ByVal handle&, channels&(), gains&(), flags&(), ByVal chanCount&)` |
| **Delphi** | `daqAdcSetScan(handle:DaqHandleT; channels:PDWORD; gains:DaqAdcGainP; flags:PDWORD; chanCount:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which ADC scan group is to be configured |
| `channels` | An array of up to 512 channel numbers |
| `*gains` | An array of up to 512 gain values |
| `flags` | Channel configuration flags in the form of a bit mask |
| `chanCount` | The number of values in the chans and gains arrays<br>Valid values: `1 -512` |
| **Returns** | `DerrNotCapable` - No high speed digital<br>`DerrInvGain` - Invalid gain<br>`DerrInvChan` - Invalid channel<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcGetScan, daqAdcSetMux` |
| **Program References** | ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

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

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

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

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

| 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 11-39) |
| **See Also** | `daqAdcSetAcq` |
| **Program References** | ADCEX1.C, DACEX1.C, DAQEX.FRM (VB), ADCEX.PAS, ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

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

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

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

## daqAdcSoftTrig

| | |
|---|---|
| **DLL Function** | `daqAdcSoftTrig(DaqHandleT handle);` |
| **C** | `daqAdcSoftTrig(DaqHandleT handle);` |
| **Visual BASIC** | `VBdaqAdcSoftTrig&(ByVal handle&)` |
| **Delphi** | `daqAdcSoftTrig(handle:DaqHandleT)` |
| **Parameters** | |
| `handle` | Handle to the device to which the ADC software trigger is to be applied |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcSetTrig, daqAdcSetAcq` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqAdcTransferBufData

| | |
|---|---|
| **DLL Function** | `daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);` |
| **C** | `daqAdcTransferBufData(DaqHandleT handle, PWORD buf, DWORD scanCount, DaqAdcBufferXferMask bufMask, PDWORD retCount);` |
| **Visual BASIC** | `VBdaqAdcTransferBufData(ByVal handle, buf%, ByVal scanCount&, ByVal bufMask&, retCount&);` |
| **Delphi** | `daqAdcTransferBufData(handle: DaqHandleT; buf : PWORD, scanCount : DWORD, bufMask: DaqAdcBufferXferMask;  retCount: PDWORD);` |
| **Parameters** | |
| `handle` | Handle to the device for which the ADC buffer should be retrieved. |
| `buf` | Pointer to an application-supplied buffer to place the buffered data. |
| `scanCount` | Number of scans to retrieve from the acquisition buffer. |
| `bufMask` | A mask defining operation depending on the current state of the acquisition buffer |
| `retCount` | A pointer to the total number of scans returned, if any. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcTransferSetBuffer, daqAdcTransferGetStat` |
| **Program References** | ADCEX9.C, ADCEX10.C |
| **Used With** | All devices |

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

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

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

## daqAdcTransferGetStat

| | |
|---|---|
| **DLL Function** | `daqAdcTransferGetStat(DaqHandleT handle, PDWORD active, PDWORD retCount);` |
| **C** | `daqAdcTransferGetStat(DaqHandleT handle, PDWORD active, PDWORD retCount);` |
| **Visual BASIC** | `VBdaqAdcTransferGetStat&(ByVal handle&, active&, retCount&)` |
| **Delphi** | `daqAdcTransferGetStat( handle:DaqHandleT; var active:DWORD; var retCount:DWORD )` |
| **Parameters** | |
| `handle` | Handle to the device for which ADC transfer status is to be retrieved |
| `active` | A pointer to the transfer-state flags in the form of a bit mask |
| `retCount` | A pointer to the total number of ADC scans acquired (or available) in the current transfer |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferStop` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| **Used With** | All devices |

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

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

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

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

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

# daqAdcTransferSetBuffer

| | |
|---|---|
| **DLL Function** | `DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);` |
| **C** | `DaqAdcTransferSetBuffer(DaqHandleT handle, PWORD buf, DWORD scanCount, DWORD transferMask);` |
| **Visual BASIC** | `VBdaqAdcTransferSetBufferAllocMem&(ByVal handle&, ByVal scanCount&, ByVal transferMask&)` |
| **Delphi** | `daqAdcTransferSetBufferAllocMem(handle:DaqHandleT; scanCount:DWORD; transferMask:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which an ADC transfer is to be performed. |
| `buf` | Pointer to the buffer for which the acquired data is to be placed. |
| `scanCount` | The total length of the buffer (in scans). |
| `transferMask` | Configures the buffer transfer mode. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcTransferStart, daqAdcTransferStop, daqAdcTransferGetStat, daqAdcSetAcq, daqAdcTransferBufData` |
| **Program References** | ADCEX1.C, DACEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi), ADCEX9.C, ADCEX10.C |
| **Used With** | All devices |

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

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

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

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

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

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

## daqAdcTransferStart

| DLL Function | daqAdcTransferStart(DaqHandleT handle); |
|---|---|
| C | daqAdcTransferStart(DaqHandleT handle); |
| Visual BASIC | VBdaqAdcTransferStart&(ByVal handle&) |
| Delphi | daqAdcTransferStart( handle:DaqHandleT ) |
| **Parameters** | |
| handle | Handle to the device to initiate an ADC transfer |
| Returns | DerrNoError - No error (also, refer to *API Error Codes* on page 11-39) |
| See Also | daqAdcTranferSetBuffer, daqAdcTransferGetStat, daqAdcTransferStop |
| Program References | ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS (Delphi) |
| Used With | All devices |

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

## daqAdcTransferStop

| DLL Function | daqAdcTransferStop(DaqHandleT handle); |
|---|---|
| C | daqAdcTransferStop(DaqHandleT handle); |
| Visual BASIC | VBdaqAdcTransferStop&(ByVal handle&) |
| Delphi | daqAdcTransferStop( handle:DaqHandleT ) |
| **Parameters** | |
| handle | Handle to the device for which the Adc data transfer is to be stopped |
| Returns | DerrNoError - No error (also, refer to *API Error Codes* on page 11-39) |
| See Also | daqAdcTransferSetBuffer, daqAdcTransferStart, daqAdcTransferGetStat |
| Program References | None |
| Used With | All devices |

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

## daqAutoZeroCompensate

| DLL Function | daqAutoZeroCompensate(BOOL zero); |
|---|---|
| C | daqAutoZeroCompensate(BOOL zero); |
| Visual BASIC | VBdaqAutoZeroCompensate&(ByVal zero&) |
| Delphi | daqAutoZeroCompensate(zero:longbool) |
| **Parameters** | |
| zero | If non-zero will enable auto zero compensation in the daqCvtTC... functions |
| Returns | DerrZCInvParam - Invalid parameter value<br>DerrNoError - No error (also, refer to *API Error Codes* on page 11-39) |
| See Also | daqZeroSetup, daqZeroConvert, daqZeroSetupConvert, daqCvtTCSetup, daqCvtTCConvert, daqcvtTcSetupConvert |
| Program References | None |
| Used With | All devices |

**daqAutoZeroCompensate** will configure the thermocouple linearization functions to automatically perform zero compensation. This is the easiest way to use zero compensation with the TempBook. When enabled, the thermocouple conversion functions will require a CJC zero reading and a TC zero reading to precede the actual CJC and TC reading. This can easily be done by configuring the scan group to read:

- channel 18 using the TempBook CJC gain code (CJC zero)
- channel 18 using the gain code of the connected TC (TC zero)
- channel 16 using the TempBook CJC gain code (CJC)
- and finally, the thermocouple channels using the gain code of the connected thermocouples.

**Note**: the offset of the real CJC value should be specified (not the offset of the CJC zero) when calling the thermocouple linearization setup functions.

## daqClose

| DLL Function | `daqClose(DaqHandleT handle);` |
|---|---|
| C | `daqClose(DaqHandleT handle);` |
| Visual BASIC | `VBdaqClose&(ByVal handle&)` |
| Delphi | `daqClose(handle:DaqHandleT)` |
| **Parameters** | |
| `handle` | Handle to the device to be closed |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | daqOpen |
| **Program References** | ADCEX1.C, DACEX1.C, DIGEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ADCEX.PAS, ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqCvtRawDataFormat

| DLL Function | `daqCvtRawDataFormat(PWORD buf, DaqAdcCvtAction` **action, DWORD lastRetCount, DWORD scanCount, DWORD chanCount);** |
|---|---|
| C | `daqCvtRawDataFormat(PWORD buf, DaqAdcCvtAction` **action, DWORD lastRetCount, DWORD scanCount, DWORD chanCount);** |
| Visual BASIC | `VBdaqCvtRawDataFormat&(buf%, ByVal action&, ByVal lastRetCount&,ByVal scanCount&, ByVal chanCount&)` |
| Delphi | `daqCvtRawDataFormat(PWORD buf, action:DaqAdcCvtAction; lastRetCount:DWORD; scanCount:DWORD: chanCount:DWORD);` |
| **Parameters** | |
| `buf` | Pointer to the buffer containing the raw data |
| `action` | The type of conversion action to perform on the raw data |
| `lastRetCount` | The last retCount returned from **daqAdcTransferGetStat** |
| `scanCount` | The length of the raw data buffer in scans |
| `chanCount` | The number of channels per scan in the raw data buffer |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | **daqAdcSetDataFormat** |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

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

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

## daqCvtSetAdcRange

| DLL Function | `daqCvtSetAdcRange(FLOAT Admin, FLOAT Admax);` |
|---|---|
| C | `daqCvtSetAdcRange(FLOAT Admin, FLOAT Admax);` |
| Visual BASIC | `VBdaqCvtSetAdcRange&(ByVal ADmin!, ByVal ADmax!)` |
| Delphi | `daqCvtSetAdcRange(Admin:single; Admax:single)` |
| **Parameters** | |
| `Admin` | A/D minimum voltage range |
| `Admax` | A/D maximum voltage range |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | |
| **Program References** | None |
| **Used With** | All devices |

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

## daqCvtTCConvert

| | |
|---|---|
| **DLL Function** | `daqCvtTCConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);` |
| **C** | `daqCvtTCConvert(PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);` |
| **Visual BASIC** | `VBdaqCvtTCConvert&(counts%(), ByVal scans&, temp%(), ByVal ntemp&)` |
| **Delphi** | `daqCvtTCConvert(counts:PWORD; scans:DWORD; temp:PWORD; ntemp:DWORD)` |
| **Parameters** | |
| `counts` | An array of one or more scans of raw data as received from the device. The ADC data bits are in the 12 most significant bits of the 16-bit integers, and the tag bits (which are discarded) are in the 4 least-significant bits.<br>Valid range: Each raw data item may be any 16-bit value. |
| `scans` | The number of scans of data in `counts` array.<br>Valid range: 1 to 32768/nscan (counts is limited to 64 Kbytes). |
| `temp` | Variable array to hold converted temperature results. The integer values are 10 times the temperatures in °C. For example, 50°C would be represented as 500 and -10°C would be -100.<br>Valid range: Results range from -2000 (-200°C) to +13720 (+1372°C) depending on the thermocouple type. |
| `ntemp` | The number of entries in the temperature array. This value is checked by the functions to avoid writing past the end of the array.<br>Valid range: If avg is 0, then ntc or greater. If avg is non-zero, then scans * ntc or greater. |
| **Returns** | `DerrTCE_NOSETUP` - Setup was not called<br>`DerrTCE_PARAM` - Parameter out of range<br>`DerrNoError` - No Error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `DaqCvtTCSetup, DaqCvtTCSetupConvert` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| Gain Codes | | | | |
|---|---|---|---|---|
| Type | Unipolar Gain Code | Unipolar Gain | Bipolar Gain Code | Bipolar Gain |
| CJC | `TbkUniCJC` | `TgainX100` | `TbkBiCJC` | `TgainX50` |
| J | `TbkUniTypeJ` | `TgainX200` | `TbkBiTypeJ` | `TgainX100` |
| K | `TbkUniTypeK` | `TgainX200` | `TbkBiTypeK` | `TgainX100` |
| T | `TbkUniTypeT` | `TgainX200` | `TbkBiTypeT` | `TgainX200` |
| E | `TbkUniTypeE` | `TgainX100` | `TbkBiTypeE` | `TgainX50` |
| N28 | `TbkUniTypeN28` | `TgainX200` | `TbkBiTypeN28` | `TgainX100` |
| N14 | `TbkUniTypeN14` | `TgainX200` | `TbkBiTypeN14` | `TgainX100` |
| S | `TbkUniTypeS` | `TgainX200` | `TbkBiTypeS` | `TgainX200` |
| R | `TbkUniTypeR` | `TgainX200` | `TbkBiTypeR` | `TgainX200` |
| B | `TbkUniTypeB` | `TgainX200` | `TbkBiTypeB` | `TgainX200` |

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

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

# daqCvtTCSetup

| | |
|---|---|
| **DLL Function** | `daqCvtTCSetup(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg);` |
| **C** | `daqCvtTCSetup(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg);` |
| **Visual BASIC** | `VBdaqCvtTCSetup&(ByVal nscan&, ByVal cjcPosition&, ByVal ntc&, ByVal tcType&, ByVal bipolar&, ByVal avg&)` |
| **Delphi** | `daqCvtTCSetup(nscan:DWORD; cjcPosition:DWORD; ntc:DWORD; tcType:TCType; bipolar:longbool; avg:DWORD)` |
| **Parameters** | |
| `nscan` | The number of readings in a single scan of DaqBook/DaqBoard data. The daqCvtTC… functions can convert several consecutive scans worth of data in a single invocation.<br>Valid range: 2 to 512. |
| `cjcPosition` | The position of the actual cold-junction compensation circuit (CJC) reading within each scan (not the CJC zero reading, if any). The first reading of the scan is position 0, and the last reading is nscan -1. Each scan of temperature data must include a reading of the CJC signal on the expansion board to which the thermocouples are attached. The CJC readings must be taken with the gain in the section *Scan Setup*.<br>Valid range: 0 to nscan-2 with no zero compensation; 2 to nscan-2 with zero compensation. |
| `ntc` | The number of thermocouple signals that are to be converted to temperature values. The thermocouple signal readings must immediately follow the CJC reading in the scan data. The first thermocouple signal is at scan position cjcPosition+1,; the next is at cjcPosition+2,; and so on. Valid range: 1 to nscan-1-cjcPosition. |
| `tcType` | The type of thermocouples that generated the measurements. Valid range: One of the pre-defined values, **`TbkTCTypeJ, TbkTCTypeK, TbkTCTypeT, TbkTCTypeE, TbkTCTypeN28, TbkTCTypeN14, TbkTCTypeS, TbkTCTypeR`** or **`TbkTCTypeB.`** |
| `bipolar` | Must be set true (non-zero) if the readings were acquired with the Daq* set for bipolar operation. Must be set false (zero) for unipolar operation. The required gain settings for the CJC and thermocouple channels change depending on the unipolar/bipolar mode. Valid range: 0 for unipolar or any non-zero value for bipolar. |
| `avg` | The type of averaging to be performed. Valid range: any unsigned integer. Since the thermocouple voltage may be small compared to the ambient electrical noise, averaging may be necessary to yield a steady temperature output.<br>0 specifies block averaging in which all of the scans are averaged together to compute a single temperature measurement for each of the ntemp thermocouples.<br>1 specifies no averaging. Each scan's readings are converted into ntemp measured temperatures for a total of scans*ntemp results.<br>2 or more specifies moving average of the specified number of scans. Scan readings are averaged with the avg-1 preceding scans' readings before conversion. The first avg-1 scans are averaged with all of the preceding scans because they do not have enough preceding scans. For example, if avg is 3, then the results from the first scan are not averaged at all, the results from the second scan are averaged with the first scan, the results from the third and subsequent scans are averaged with the preceding two scans as shown in the table. |
| **Returns** | `DerrTCE_PARAM` - Parameter out of range<br>`DerrTCE_TYPE` - Invalid thermocouple type<br>`DerrNoError` - No Error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqCvtTCConvert, daqCvtTCSetupConvert` |
| **Program References** | None |
| **Used With** | All devices |

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

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

## daqCvtTCSetupConvert

| | |
|---|---|
| **DLL Function** | `daqCvtTCSetupConvert(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);` |
| **C** | `daqCvtTCSetupConvert(DWORD nscan, DWORD cjcPosition, DWORD ntc, TCType tcType, BOOL bipolar, DWORD avg, PWORD counts, DWORD scans, PSHORT temp, DWORD ntemp);` |
| **Visual BASIC** | `VBdaqCvtTCSetupConvert&(ByVal nscan&, ByVal cjcPosition&, ByVal ntc&, ByVal tcType&, ByVal bipolar&, ByVal avg&, counts%(), ByVal scans&, temp%(), ByVal ntemp&)` |
| **Delphi** | `daqCvtTCSetupConvert(nscan:DWORD; cjcPosition:DWORD; ntc:DWORD; tcType:TCType; bipolar:longbool; avg:DWORD; counts:PWORD; scans:DWORD; temp:PWORD; ntemp:DWORD)` |
| **Parameters** | |
| `nscan` | The number of readings in a single scan.<br>Valid range: 1- 512 |
| `cjcPosition` | The position of the CJC reading within the scan.<br>Valid range:<br>16<br>18, if auto-zeroing is used with the TempBook. |
| `ntc` | The number of thermocouple readings that immediately follow the CJC reading within the scan.<br>Valid range: 1 -(nscan-cjcposition-1) |
| `tcType` | The type of thermocouples being measured. |
| `bipolar` | Non-zero if the DaqBook/DaqBoard is configured for bipolar readings. |
| `avg` | The type of averaging to be performed: block, none or moving. |
| `counts` | The raw data from one or more scans. |
| `scans` | The number of scans of raw data in counts. |
| `temp` | The converted temperatures in tenths of a degree C. |
| `ntemp` | The number of elements provided in the temp array (for error checking). |
| **Returns** | `DerrTCE_PARAM` - Parameter out of range<br>`DerrTCE_TYPE` - Invalid thermocouple type<br>`DerrNoError` - No Error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `DaqCvtTCSetup, daqCvtTCConvert` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqDefaultErrorHandler

| | |
|---|---|
| **DLL Function** | `daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);` |
| **C** | `daqDefaultErrorHandler(DaqHandleT handle, DaqError errCode);` |
| **Visual BASIC** | `VBdaqDefaultErrorHandler(ByVal handle&, ByVal errCode&)` |
| **Delphi** | `daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)` |
| **Parameters** | |
| `handle` | Handle to the device to which the default error handler is to be attached. |
| `ErrCode` | The error code number of the detected error (see table *API Error Codes* at end of this chapter). |
| **Returns** | Nothing (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetLastError, daqProcessError, daqSetDefaultErrorHandler` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqFormatError

| | |
|---|---|
| **DLL Function** | `daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);` |
| **C** | `daqCalSelectInputSignal(DaqHandleT handle, DaqCalInputT input);` |
| **Visual BASIC** | `VBdaqCalSelectInputSignal(ByVal handle&, ByVal input as DaqCalInputT )` |
| **Delphi** | `daqCalSelectInputSignal( handle: DaqHandleT; input: DaqCalInputT)` |
| **Parameters** | |
| `daqError` | Daq* 32-bit 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 11-39) |
| **See Also** | `daqSetDefaultErrorHandler, daqSetErrorHandler, daqProcessError, daqGetLastError, daqDefaultErrorHandler` |
| **Program References** | None |
| **Used With** | All devices |

**daqFormatError** returns the text-string equivalent for the specified error condition. The error condition is specified by the **daqError** parameter. The error text will be returned in the character string pointed to by the **msg** parameter. The character string space must have been previously allocated by the application before calling this function. The allocated character string should be, at minimum, 64 bytes in length. For more information on specific error codes refer to the *API Error Codes* on page 11-39.

## daqGetDeviceCount

| | |
|---|---|
| **DLL Function** | `daqGetDeviceCount(PDWORD deviceCount);` |
| **C** | `daqGetDeviceCount(PDWORD deviceCount);` |
| **Visual BASIC** | `VBdaqGetDevice&(deviceCount&)` |
| **Delphi** | `daqGetDeviceCount( var deviceCount:DWORD )` |
| **Parameters** | |
| `deviceCount` | Pointer to which the device count is to be returned |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetDeviceList, daqGetDeviceProperties` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqGetDeviceList

| | |
|---|---|
| **DLL Function** | `daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);` |
| **C** | `daqGetDeviceList(DaqDeviceListT *deviceList, PDWORD deviceCount);` |
| **Visual BASIC** | `VBdaqGetDeviceList(deviceList as DaqDeviceListT, deviceCount&)` |
| **Delphi** | `daqGetDeviceList( var deviceList: DaqDeviceListT; var deviceCount: DWORD)` |
| **Parameters** | |
| `deviceList` | Pointer to memory location to which the device list is to be returned |
| `deviceCount` | Number of devices returned in the device list |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetDeviceCount, daqGetDeviceProperties, daqOpen,` |
| **Program References** | None |
| **Used With** | All devices |

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

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

## daqGetDeviceProperties

| | |
|---|---|
| **DLL Function** | `daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);` |
| **C** | `daqGetDeviceProperties(LPSTR daqName, DaqDevicePropsT *deviceProps);` |
| **Visual BASIC** | `VBdaqGetDeviceProperties( daqName$, deviceProps as DaqDevicePropsT)` |
| **Delphi** | `daqGetDeviceProperties( daqName: string; var deviceProps: DaqDevicePropsT)` |
| **Parameters** | |
| `daqName` | Pointer to a character string representing the name of the device for which to retrieve properties |
| `deviceCount` | Number of devices returned in the device list |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetDeviceCount, daqGetDeviceList, daqOpen` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

## daqGetDriverVersion

| | |
|---|---|
| **DLL Function** | `daqGetDriverVersion(PDWORD version);` |
| **C** | `daqGetDriverVersion(PDWORD version);` |
| **Visual BASIC** | `VBdaqGetDriverVersion&(version&)` |
| **Delphi** | `daqGetDriverVersion(var version:DWORD)` |
| **Parameters** | |
| `version` | Pointer to the version number of the current device driver. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetHardwareInfo` |
| **Program References** | ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqGetHardwareInfo

| | |
|---|---|
| **DLL Function** | `daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);` |
| **C** | `daqGetHardwareInfo(DaqHandleT handle, DaqHardwareInfo whichInfo, VOID * info);` |
| **Visual BASIC** | `VBdaqGetHardwareInfo&(ByVal handle&, ByVal whichInfo&, info As Variant)` |
| **Delphi** | `daqGetHardwareInfo(handle:DaqHandleT; whichInfo:DaqHardwareInfo; info:pointer)` |
| **Parameters** | |
| `handle` | Handle to the device |
| `whichInfo` | Specifies what type of device information to retrieve |
| `* info` | Pointer to the returned device information |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqGetDriverVersion, daqOpen` |
| **Program References** | DACEX.PAS, ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqGetLastError

| | |
|---|---|
| **DLL Function** | `daqGetLastError(DaqHandleT handle, DaqError *errCode);` |
| **C** | `daqGetLastError(DaqHandleT handle, DaqError *errCode);` |
| **Visual BASIC** | `VBdaqGetLastError&(ByVal handle&, errCode&)` |
| **Delphi** | `daqGetLastError(handle:DaqHandleT; var errCode:DaqError): DaqError; stdcall; external DAQX_DLL; procedure daqDefaultErrorHandler(handle:DaqHandleT; errCode:DaqError)` |
| **Parameters** | |
| `handle` | Handle to the device |
| `*errCode` | Returned last error code |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqDefaultErrorHandler, daqProcessError, daqSetDefaultErrorHandler` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqOnline

| | |
|---|---|
| **DLL Function** | `daqOnline(DaqHandleT handle, PBOOL online);` |
| **C** | `daqOnline(DaqHandleT handle, PBOOL online);` |
| **Visual BASIC** | `VBdaqOnline&(ByVal handle&, online&)` |
| **Delphi** | `daqOnline(handle: DaqHandleT; var online: longbool)` |
| **Parameters** | |
| `handle` | Handle of the device to test for online |
| `online` | Boolean indicating whether the device is currently online |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqOpen, daqClose` |
| **Program References** | ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqOpen

| | |
|---|---|
| **DLL Function** | `daqOpen(LPSTR daqName);` |
| **C** | `daqOpen(LPSTR daqName);` |
| **Visual BASIC** | `VBdaqOpen&(ByVal daqName$)` |
| **Delphi** | `daqOpen(devName: PChar)` |
| **Parameters** | |
| `daqName` | String representing the name of the device to be opened |
| **Returns** | A handle to the specified device (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqClose, daqOnline` |
| **Program References** | ADCEX1.C, DIGEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS, ADCEX.PAS (Delphi) |
| **Used With** | |

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

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

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

## daqProcessError

| | |
|---|---|
| **DLL Function** | `daqProcessError(DaqHandleT handle, DaqError errCode);` |
| **C** | `daqProcessError(DaqHandleT handle, DaqError errCode);` |
| **Visual BASIC** | `VBdaqProcessError&(ByVal handle&, ByVal errCode&)` |
| **Delphi** | `daqProcessError(handle:DaqHandleT; errCode:DaqError)` |
| **Parameters** | |
| `handle` | Handle to the device for which the specified error is to be processed. |
| `errCode` | Specifies the device error code to process |
| **Returns** | Refer to *API Error Codes* on page 11-39 |
| **See Also** | `daqSetDefaultErrorHandler, daqGetLastError, daqDefaultErrorHandler` |
| **Program References** | None |
| **Used With** | All devices |

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

## daqSetDefaultErrorHandler

| | |
|---|---|
| **DLL Function** | `daqSetDefaultErrorHandler(DaqErrorHandlerFPT handler);` |
| **C** | `daqSetDefaultErrorHandler(DaqErrorHandlerFPT handler);` |
| **Visual BASIC** | `VBdaqSetDefaultErrorHandler&(ByVal handler&)` |
| **Delphi** | `daqSetDefaultErrorHandler(handler:DaqErrorHandlerFPT)` |
| **Parameters** | |
| `handler` | Pointer to a user-defined error handler function. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqDefaultErrorHandler, daqGetLastError, daqProcessError, daqSetErrorHandler` |
| **Program References** | ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

## daqSetErrorHandler

| | |
|---|---|
| **DLL Function** | `daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);` |
| **C** | `daqSetErrorHandler(DaqHandleT handle, DaqErrorHandlerFPT handler);` |
| **Visual BASIC** | `VBdaqSetErrorHandler&(ByVal handle&, ByVal handler&)` |
| **Delphi** | `daqSetErrorHandler(handle:DaqHandleT; handler:DaqErrorHandlerFPT)` |
| **Parameters** | |
| `handle` | Handle to the device to which to attach the specified error handler |
| `handler` | Pointer to a user defined error handler function. |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqSetDefaultErrorHandler, daqDefaultErrorHandler, daqGetLastError, daqProcessError` |
| **Program References** | ADCEX1.C, DYN32ENH.C, DAQEX.FRM (VB), ERREX.PAS (Delphi) |
| **Used With** | All devices |

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

# daqSetOption

| | |
|---|---|
| **DLL Function** | `daqSetOption(DaqHandleT handle, DWORD chan, DWORD flags, DaqOptionType optionType, FLOAT optionValue);` |
| **C** | `daqSetOption(DaqHandleT handle, DWORD chan, DWORD flags, DaqOptionType optionType, FLOAT optionValue);` |
| **Visual BASIC** | `VBdaqSetOption&(ByVal handle&, ByVal chan&, ByVal flags&, ByVal optionType&, ByVal optionValue!)` |
| **Delphi** | `daqSetOption(Handle:DaqHandleT; chan:DWORD; flags:DWORD; optionType:DaqOptionType; optionValue:FLOAT)` |
| **Parameters** | |
| `handle` | The handle to the device for which to set the option |
| `chan` | The channel number on the device for which the option is to be set |
| `flags` | Flags specifying the options to use. |
| `optionType` | Specifies the type of option. |
| `optionValue` | The value of the option to set |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqAdcExpSetChanOption,` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

# daqSetTimeout

| | |
|---|---|
| **DLL Function** | `daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);` |
| **C** | `daqSetTimeout(DaqHandleT handle, DWORD mSecTimeout);` |
| **Visual BASIC** | `VBdaqSetTimeout&(ByVal handle&, ByVal mSecTimeout&)` |
| **Delphi** | `daqSetTimeout(handle:DaqHandleT; mSecTimeout:DWORD)` |
| **Parameters** | |
| `handle` | Handle to the device for which the event time-out is to be set |
| `mSecTimeout` | Specifies time-out ( ms ) for events |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqWaitForEvent, daqWaitForEvents` |
| **Program References** | None |
| **Used With** | All devices |

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

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

| DLL Function | `daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);` |
|---|---|
| C | `daqTest(DaqHandleT handle, DaqTestCommand command, DWORD count, PBOOL cmdAvailable, PDWORD result);` |
| Visual BASIC | `VBdaqTest&(ByVal handle&, ByVal command&, ByVal count&, cmdAvailable&, result&)` |
| Delphi | [not supported] |
| **Parameters** | |
| `handle` | Handle to the device for which the test is to be performed |
| `command` | Specifies the type of test to be run |
| `count` | Optional parameter which specifies the length of the test |
| `cmdAvailable` | Return Boolean indicating the availability of the test for the device |
| `result` | Pointer to the test result field |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqOpen` |
| **Program References** | None |
| **Used With** | All devices |

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

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

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

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

**Performance tests** measure the speed at which certain operations can be performed on the device. In general, the performance test results indicate the maximum rate at which the operation can be performed on the device. The valid performance test types are defined as follows:

- **`DtsAdcFifoInputSpeed`** - This test will determine the maximum rate at which analog input can be acquired and transferred to system memory. Analog input performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsDacFifoOutputSpeed`** - (DaqBoard only) This test will determine the maximum rate at which analog output data can be read from system memory and transferred to the device's DAC FIFO. Analog output performance results will be returned in the **`result`** parameter with units of samples/second.
- **`DtsIOInputSpeed`** - This test will determine the maximum rate at which digital input can be read from the device's DIO port and transferred to system memory. Digital input performance results will be returned in the **`result`** parameter with units of bytes/second.
- **`DtsIOOutputSpeed`** - This test will determine the maximum rate at which digital output can be read from system memory and output to the device's DIO port. Digital output performance results will be returned in the **`result`** parameter with units of bytes/second.

The **`cmdAvailable`** parameter is a pointer to a Boolean value that indicates whether or not the specified test is available for the device.

The **count** parameter can be used to indicate the duration or length of the test. For instance, a resource test will be run **count** times; and if any one iteration of the test fails, it will indicate an overall failure of the test. For a performance test, the **count** parameter will indicate the number of times to run the test, and the test result will be an average of all the tests performed.

## daqWaitForEvent

| | |
|---|---|
| **DLL Function** | `daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);` |
| **C** | `daqWaitForEvent(DaqHandleT handle, DaqTransferEvent daqEvent);` |
| **Visual BASIC** | `VBdaqWaitForEvent&(ByVal handle&, ByVal daqEvent&)` |
| **Delphi** | `daqWaitForEvent(handle:DaqHandleT; daqEvent:DaqTransferEvent)` |
| **Parameters** | |
| `handle` | Handle of the device for which to wait of the specified event |
| `daqEvent` | Specifies the event to wait on |
| **Returns** | `DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **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,`<br>`DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);` |
| **C** | `daqWaitForEvents(DaqHandleT *handles, DaqTransferEvent *daqEvents,`<br>`DWORD eventCount, BOOL *eventSet, DaqWaitMode waitMode);` |
| **Visual BASIC** | `VBdaqWaitForEvents&(handles&(), daqEvents&(), ByVal eventCount&, eventSet&(),`<br>`ByVal waitMode&)` |
| **Delphi** | `daqWaitForEvents(handles:DaqHandlePT; daqEvents:DaqTransferEventP;`<br>`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 11-39) |
| **See Also** | `daqWaitForEvent, daqSetTimeout` |
| **Program References** | None |
| **Used With** | All devices |

**daqWaitForEvents** allows you to wait on specific Daq* events to occur on the specified devices. This function will wait on the specified events and will return based upon the criteria selected with the **waitMode** parameter. A time-out for all events can be specified using the **daqSetTimeout** command.

Events to wait on are specified by passing an array of event definitions in the **events** parameter. The number of events is specified with the **eventCount** parameter. See the *Transfer Event Definitions* table for **events** parameter definitions. Also see the *Transfer Event Wait Mode Definitions* table for **waitMode** parameter definitions.

## daqZeroConvert

| DLL Function | `daqZeroConvert(PWORD counts, DWORD scans);` |
|---|---|
| C | `daqZeroConvert(PWORD counts, DWORD scans);` |
| Visual BASIC | `VBdaqZeroConvert&(counts%(), ByVal scans&)` |
| Delphi | `daqZeroConvert(counts:PWORD; scans:DWORD)` |
| **Parameters** | |
| `counts` | The raw data from one or more scans. |
| `scans` | The number of scans of raw data in the counts array. |
| Returns | `DerrZCInvParam` - Invalid parameter value<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| See Also | `daqZeroSetup, daqZeroSetupConvert, daqZeroDbk19` |
| Program References | None |
| Used With | All devices |

**`daqZeroConvert`** compensates one or more scans according to the previously called **`daqZeroSetup`** function. This function will modify the array of data passed to it.

## daqZeroSetup

| DLL Function | `daqZeroSetup(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings);` |
|---|---|
| C | `daqZeroSetup(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings);` |
| Visual BASIC | `VBdaqZeroSetup&(ByVal nscan&, ByVal zeroPos&, ByVal readingsPos&, ByVal nReadings&)` |
| Delphi | `daqZeroSetup(nscan:DWORD; zeroPos:DWORD; readingsPos:DWORD; nreadings:DWORD)` |
| **Parameters** | |
| `nscan` | The number of readings in a single scan. |
| `zeroPos` | The position of the zero reading within the scan |
| `readingsPos` | The position of the readings to be zeroed within the scan. |
| `nReadings` | The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading. |
| Returns | `DerrZCInvParam` - Invalid parameter value<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| See Also | `daqZeroConvert, daqZeroSetupConvert, daqZeroDbk19` |
| Program References | None |
| Used With | All devices |

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

# daqZeroSetupConvert

| | |
|---|---|
| **DLL Function** | `daqZeroSetupConvert(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings, PWORD counts, DWORD scans);` |
| **C** | `daqZeroSetupConvert(DWORD nscan, DWORD zeroPos, DWORD readingsPos, DWORD nReadings, PWORD counts, DWORD scans);` |
| **Visual BASIC** | `VBdaqZeroSetupConvert&(ByVal nscan&, ByVal zeroPos&, ByVal readingsPos&, ByVal nReadings&, counts%(), ByVal scans&)` |
| **Delphi** | `daqZeroSetupConvert(nscan:DWORD; zeroPos:DWORD; readingsPos:DWORD; nreadings:DWORD; counts:PWORD; scans:DWORD)` |
| **Parameters** | |
| `nscan` | The number of readings in a single scan. |
| `zeroPos` | The position of the zero reading within the scan |
| `readingsPos` | The position of the readings to be zeroed within the scan. |
| `nReadings` | The number of readings immediately following the zero reading that are sampled at the same gain as the zero reading. |
| `counts` | The raw data from one or more scans. |
| `scans` | The number of scans of raw data in the counts array. |
| **Returns** | `DerrZCInvParam` - Invalid parameter value<br>`DerrNoError` - No error (also, refer to *API Error Codes* on page 11-39) |
| **See Also** | `daqZeroSetup, daqZeroConvert, daqZeroDbk19` |
| **Program References** | None |
| **Used With** | All devices |

**`daqZeroSetupConvert`** performs both the setup and convert steps with one call. This is useful when the zero compensation needs to be performed multiple times because data was read from channels at different gains or from different boards.

# API Reference Tables

These tables provide information for programming with the Daq* Application Programming Interface. Information includes channel identification and error codes, as well as valid parameter values and descriptions. The tables are organized as follows:

**API Parameter Reference Tables**

| Table Title | Sub-Title/Parameter/Description | Page |
|---|---|---|
| Daq Device Property Definitions - `daqGetDeviceProperties` | Identifies the format (`DWORD`, `STRING`, or `FLOAT`) for property parameters | 11-36 |
| Event-Handling Definitions | Transfer Event Definitions - `DaqTransferEvent`<br>Transfer Event Wait Mode Definitions - `DaqWaitMode` | 11-36 |
| Hardware Information Definitions | Hardware Information Selector Definitions - `DaqHardwareInfo`<br>Hardware Version Definitions - `DaqHardwareVersion` | 11-36 |
| ADC Trigger Source Definitions | `DaqAdcTriggerSource`<br>`DaqEnhTrigSensT` | 11-37 |
| ADC Miscellaneous Definitions | ADC Flag Definitions - `DaqAdcFlag`<br>Frequency vs Period - `DaqAdcRateMode`<br>ADC Acquisition Mode Definitions - `DaqAdcAcqMode`<br>ADC Transfer Mask Definitions - `DaqAdcTransferMask`<br>ADC Clock Source Definitions - `DaqAdcClockSource`<br>ADC File Open Mode Definitions - `DaqAdcOpenMode`<br>ADC Acquisition/Transfer Active Flag Definitions - `DaqAdcActiveFlag`<br>ADC Acquisition State - `DaqAdcAcqState`<br>ADC BufferTransfer Mask- `DaqAdcBufferXferMask` | 11-37 |
| TempBook Definitions | Unipolar Thermocouple Gain Definitions<br>Bipolar Thermocouple Gain Definitions<br>Thermocouple Definitions<br>Voltage Gain Definitions | 11-38 |
| General I/O Definitions | I/O Operation Code Definitions - `DaqIOOperationCode` | 11-38 |
| DaqTest Command Definitions | `DaqTestCommand` | 11-38 |
| Calibration Input Signal Sources | `DaqCalInputT`<br>`DaqCalTableTypeT` | 11-38 |
| API Error Codes | Identifies API errors by number and description | 11-39 |

## 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 |

## TempBook Definitions

| *Unipolar Thermocouple Gain Definitions* | | *Bipolar Thermocouple Gain Definitions* | |
|---|---|---|---|
| `TbkUniCJC` | `TgainX100` | `TbkBiCJC` | `TgainX50` |
| `TbkUniTypeJ` | `TgainX200` | `TbkBiTypeJ` | `TgainX100` |
| `TbkUniTypeK` | `TgainX200` | `TbkBiTypeK` | `TgainX100` |
| `TbkUniTypeT` | `TgainX200` | `TbkBiTypeT` | `TgainX200` |
| `TbkUniTypeE` | `TgainX100` | `TbkBiTypeE` | `TgainX50` |
| `TbkUniTypeN28` | `TgainX200` | `TbkBiTypeN28` | `TgainX100` |
| `TbkUniTypeN14` | `TgainX200` | `TbkBiTypeN14` | `TgainX100` |
| `TbkUniTypeS` | `TgainX200` | `TbkBiTypeS` | `TgainX200` |
| `TbkUniTypeR` | `TgainX200` | `TbkBiTypeR` | `TgainX200` |
| `TbkUniTypeB` | `TgainX200` | `TbkBiTypeB` | `TgainX200` |

| *Thermocouple Definitions* | | *Voltage Gain Definitions* | |
|---|---|---|---|
| `TbkTCTypeJ` | 18 | `TgainX1` | 0 |
| `TbkTCTypeK` | 19 | `TgainX2` | 1 |
| `TbkTCTypeT` | 20 | `TgainX5` | 2 |
| `TbkTCTypeE` | 21 | `TgainX10` | 3 |
| `TbkTCTypeN28` | 22 | `TgainX20` | 5 |
| `TbkTCTypeN14` | 23 | `TgainX50` | 6 |
| `TbkTCTypeS` | 24 | `TgainX100` | 7 |
| `TbkTCTypeR` | 25 | `TgainX200` | 11 |
| `TbkTCTypeB` | 26 | | |

## General I/O Definitions

| *I/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<br>Name | Code #<br>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` |
| `DerrRtdArraySize` | `3Dh - 61` | Temperature array not large enough |
| `DerrRtdParam` | `3Eh - 62` | Incorrect RTD parameter |

| Error | Code # | |
|---|---|---|
| Name | hex - dec | Description |
| DerrInvBankType | 3Fh - 63 | Invalid bank-type specified |
| DerrBankBoundary | 40h - 64 | Simultaneous writes to DBK cards in different banks not allowed |
| DerrInvFreq | 41h - 65 | Invalid scan frequency specified |
| DerrNoDaq | 42h - 66 | No Daq112B/216B installed |
| DerrInvOptionType | 43h - 67 | Invalid option-type parameter |
| DerrInvOptionValue | 44h - 68 | Invalid option-value parameter |
| **New API Error Codes** | | |
| DerrTooManyHandles | 60h - 96 | No more handles available to open |
| DerrInvLockMask | 61h - 97 | Only a part of the resource is already locked, must be all or none |
| DerrAlreadyLocked | 62h - 98 | All or part of the resource was locked by another application |
| DerrAcqArmed | 63h - 99 | Operation not available while an acquisition is armed |
| DerrParamConflict | 64h - 100 | Each parameter is valid, but the combination is invalid |
| DerrInvMode | 65h - 101 | Invalid acquisition/wait/dac mode |
| DerrInvOpenMode | 66h - 102 | Invalid file-open mode |
| DerrFileOpenError | 67h - 103 | Unable to open file |
| DerrFileWriteError | 68h - 104 | Unable to write file |
| DerrFileReadError | 69h - 105 | Unable to read file |
| DerrInvClockSource | 6Ah - 106 | Invalid acquisition mode |
| DerrInvEvent | 6Bh - 107 | Invalid transfer event |
| DerrTimeout | 6Ch - 108 | Time-out on wait |
| DerrInitFailure | 6Dh - 109 | Unexpected result occurred while initializing the hardware |
| DerrBufTooSmall | 6Eh - 110 | Unexpected result occurred while initializing the hardware |
| DerrInvType | 6Fh - 111 | Invalid acquisition/wait/dac mode |
| DerrArraySize | 70h - 112 | Used as a catch all for arrays not large enough |
| DerrBadAlias | 71h - 113 | Invalid alias names for Vxd lookup |
| DerrInvCommand | 72h - 114 | Invalid command |
| DerrInvHandle | 73h - 115 | Invalid handle |
| DerrNoTransferActive | 74h - 116 | Transfer not active |
| DerrNoAcqActive | 75h - 117 | Acquisition not active |
| DerrInvOpstr | 76h - 118 | Invalid operation string used for enhanced triggering |
| DerrDspCommFailure | 77h - 119 | Device with DSP failed communication |
| DerrEepromCommFailure | 78h - 120 | Device with EEPROM failed communication |
| DerrInvEnhTrig | 79h - 121 | Device using enhanced trigger detected invalid trigger type |
| DerrInvCalConstant | 7Ah - 122 | User calibration constant out of range |
| DerrInvErrorCode | 7Bh - 123 | Invalid error code |
| DerrInvAdcRange | 7Ch - 124 | Invalid analog input voltage range parameter |
| DerrInvCalTableType | 7Dh - 125 | Invalid calibration table type |
| DerrInvCalInput | 7Eh - 126 | Invalid calibration input signal selection |
| DerrInvRawDataFormat | 7Fh - 127 | Invalid raw-data format selection |
| DerrNotImplemented | 80h - 128 | Feature/function not implemented yet |
| DerrInvDioDeviceType | 81h - 129 | Invalid digital I/O device type |
| DerrInvPostDataFormat | 82h - 130 | Invalid post-processing data format selection |