BCNTDRV

Windows NT™ Drivers
for PCI Data Acquisition Cards

User Manual

BCTNTDRV

User Manual

Document Part N°	0127_1008
Document Reference	127-1008.doc
Document Issue Level	4.1
Manual covers Drivers identified	v1.0, 2.0, 3.0, 4.0 & 5.0

Amendment History

Issue Level	Issue Date	Author	Amendment Details
1.0	24.02.98	AJP	First Release
2.0	20.04.98	AJP	Added details of error log messages and a description of InitIsoDigModes
3.0	31.07.98	AJP	Added more detail of application programming and alphbetised the function order. Corrected analogue function gain definitions.
4.0	14.09.98	AJP	Added information for VB users regarding INPUT and OUTPUT used as direction parameters.

OUTLINE DESCRIPTION

The Windows NT™ driver for PCI Data Acquisition Cards ("the Windows NT driver") provide a simple programming interface to the supported range of PCI data acquisition cards for application programmers using Windows NT™ as their operating system.

The drivers provide the user with an application programming interface (API) that gives access to the most commonly used features of the PCI data acquisition boards. Not all of the hardware functionality of the PCI data acquisition cards is supported by the driver.

If you require additional functionality, contact your supplier to see if a later version of driver is available.

1.0 IMPORTANT INFORMATION

These drivers remain the property of the supplier. Please ensure that you have read the license agreement printed on the disk envelope and agree to it, and agree to be bound by the conditions laid down in it prior to opening the envelope.

Opening the envelope is taken as agreement to the terms laid down in the agreement printed upon it.

For up to date information regarding the drivers including any limitations, etc. please consult the READ.ME file contained on the installation diskette.

2.0 INSTALLATION

To install the Windows NT™ Drivers for PCI Data Acquisition Cards insert disk 1 into the floppy drive and from Windows Explorer select the appropriate floppy drive and run SETUP.EXE.

This will start the InstallShield wizard which will guide you through the set up process.

On accepting the license agreement as displayed, the installation will prompt for a directory to be specified. The header files for application programmers along with the sample code used for calling the library and drivers will be copied into this directory. Other files will be copied into the Windows NT™ system directories, as appropriate.

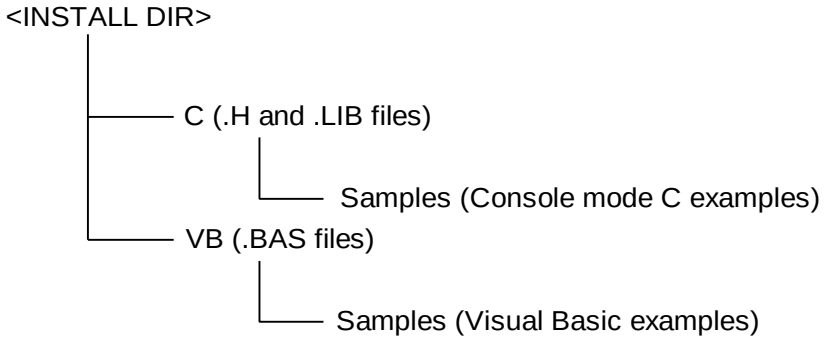
2.1 FILES INSTALLED

The following files will be copied to the hard disk drive during the installation of the Windows NT driver:

BCTNTDRV.SYS	The kernel mode device driver for the PCI data acquisition cards. This file is copied to the directory \WINNT\SYSTEM32\DRIVERS.

BCDLL32.DLL	The dynamic link library that provides API functions to the application programmer and interfaces them to the kernel mode driver. This file is copied to the directory \WINNT\SYSTEM32
BLUECHIP.H	C and C++ header file containing the function prototypes C and C++ applications.
BCTYPES.H	C and C++ header file containing constant definitions and API specific types.
BCDLL32.LIB	C and C++ link library for compiler type checking.
BLUECHIP.BAS	This is the equivalent file to the two .H header files and gives constants and function prototypes for Visual Basic users.

These files are installed into the following directory structure from the install directory specified:



3.0 USING THIS PRODUCT

To use the drivers and the associated library files, you will require some experience of developing Windows applications. Experienced programmers may choose to develop their applications using C or C++. Alternatively, the DLL and the underlying driver may also be called from development tools such as Visual Basic, Delphi, C++ Builder, etc. In any of these cases you will need to possess and be familiar with the appropriate software development kit.

3.1 USING THE DRIVERS FROM C OR C++

3.1.1 The BLUECHIP.H header file

At the top of any source module which includes a call to one of the functions provided by the DLL you should include the header file BLUECHIP.H

The header file contains:

Function prototypes for each function in the library. This is used by the 'C' compiler for parameter checking when a library function is called, and ensures that parameters of an incorrect type are not passed into the library.

Symbolic names are used to identify each PCI board type which is supported, along with driver specific structures and error codes. Use these names when making calls to the library and checking error codes returned from library functions.

3.1.2 Compiling and Linking

Once all modules have been coded and compile without errors or warnings, they must be linked with the correct libraries to form the finished executable file. In order for the application to compile correctly the following standard library include statements are required as a minimum:

```
#include <windows.h>
#include <winioctl.h>
#include <time.h>
```

The BCDLL32.DLL is supplied with an import library, BCDLL32.LIB. Although the import library does not contain the actual library functions, it does contain information which will allow each function included in the dynamic link library to be resolved when called from a C program. The import library must therefore be specified as a library module when linking the compiled modules of your C application.

The resulting Window's executable file is then aware that these functions may be found external to the executable file and must be 'linked in dynamically' at run-time.

The standard Windows libraries and DLLs also needed to build your application will be supplied with your software development kit and should be described in the documents supplied with that kit.

3.2 USING THE DRIVERS FROM VISUAL BASIC

Accessing the library from a Visual BASIC program is straight-forward. Simply include the file BLUECHIP.BAS as one of your BASIC program's modules. You may include the module by selecting the 'Add File...' option on Visual BASIC's 'File' menu. Your own modules may then call the library's functions.

BLUECHIP.BAS tells your program what functions are available and where they may be found when called i.e. in the file BCDLL32.DLL. It also allows Visual BASIC to check the parameters you pass to the functions and to provide any parameter type conversion required.

BLUECHIP.BAS also contains the symbolic names used for the Library's constants e.g. the board types, the function return codes, status codes etc. These are described in later chapters and should always be used in you own code to improve readability.

NOTE: The C header file defines two port direction constants as INPUT and OUTPUT for 8 bit port directions. These are both reserved works in Visual Basic so the constant expressions in the BLUECHIP.BAS file are INPUT8 and OUTPUT8.

3.3 IDENTIFYING BOARDS

It is not possible to identify, in advance, the order in which a particular PCI Subsystem will enumerate the PCI devices. This means that it is not possible to be sure that the physical order of Boards in a system is the same as the enumerated PCI order.

The effect of this is that when there is more than one board of a given type in the system it is not possible to relate the device driver's numbering scheme to the physical location of the boards in the system. For any particular system, the order remains the same providing the PCI subsystem (bridge chips and BIOS) remain unchanged.

To provide application developers with a method for uniquely identifying particular boards the following mechanism has been provided:

1. Each of the PCI cards to which the drivers apply has a unique serial number programmed into it and also attached to the board on the write on label on the printed circuit board. This serial number is used to positively identify one of the PCI cards. This serial number is accessible by both the device driver and the user.

2. The DLL provides a function BCTGetBoardId that will open a Board by providing a board type and a board number (zero based). An additional function is provided, BCTFindSerialNo, that will search all the boards known to the driver for one matching the serial number. The function returns a suitable Board Identifier and Number for use in a subsequent call to BCTGetBoardId. If the serial number cannot be found or the driver is not loaded then an error is returned.

3.4 APPLICATION DEVELOPMENT

No matter which development language is being used the steps for developing an application is the same:

1. Allocate and initialise handles for each device being used
2. Create and initialise a valid board ID structure
3. Initialise the directions of ports
4. Assign each handle to a port
5. Run required functions
6. Close all handles prior to terminating

If we take as an example writing an application that uses the two of the ports on the PCI_PIO (first 8255 ports A and B) one (port A) for input and the second (port B) for output we would need to use the following process

3.4.1 Determine the board IDs and handles required.

As we are only using 1 PCI data acquisition card we will need only 1 board ID structure, however as we plan to use two ports on the PCI_PIO we will need two handles. These can be defined as:

```
// Define a board ID structure for the PCI_PIO.  We
// don't fill any of the data only reference it by
// name.
BCT_BOARD_ID nPCIPIOBoardID;

// Define two handles one for input and one for
// output
BCT_HANDLE nOutputHandle;
BCT_HANDLE nInputHandle;
```

3.4.2 Initialise the handles and board ID structure

Now we have declared the variables required for the board ID and the handles we need to make the appropriate calls to the driver to get NT to complete the initialisation of the structures. Each time we make a call to the driver we should check the return code from the function and act as appropriate. In this example the error checking is only shown for the first call to the driver but all calls should be checked.

```
BCT_DWORD nStatus;
char *sErrTxt;

// Get the board id structure completed by the
// driver
nStatus = BCTGetBoardId(&nPCIPIOBoardID, PCI_PIO, 0);
if (nStatus != BCT_OK)
{
      BCTErr2Txt(nStatus, sErrTxt);
      printf("Status: %s\n", sErrTxt);
      return(nStatus);
}

// Get the handles initialised by the driver
nStatus = BCTInitHandle(&nOutputHandle);
// Check nStatus returned
nStatus = BCTInitHandle(&InputHandle);
// Check nStatus returned
```

3.4.3 Initialise the port directions in the 8255

We are using two ports on the first PIO (PIO 0) these are port A as input and port B as output. If we were using both PIOs on the PCI_PIO we would need to call BCTInit8255Modes twice, once for each PIO.

```
// Init the 8255 using the board ID structure
// declared and initialised.  0 is PIO 0, using
// MODE 0, port A is input, port B is output, port
// C is NOCARE as not used.

nStatus = BCTInit8255Modes(&nPCIPIOBoardID,
            0, MODE_0, INPUT, OUTPUT, NOCARE);
// Check nStatus returned
```

3.4.4 Assign each allocated handle to a port

Now that the 8255 has been set up we can assign each of the handles to represent a specific port within the device.

```
// nInputHandle is for port A
nStatus = BCTOpen(&nInputHandle, &nPCIPIOBoardID,
                          BCT_8255, 0, BCT_PORT_A);
// Check nStatus returned

// nOutputHandle is for port B
nStatus = BCTOpen(&nOutputHandle, &nPCIPIOBoardID,
                          BCT_8255, 0, BCT_PORT_B);
// Check nStatus returned
```

3.4.5 Using the ports

If all the initialisation detailed above has completed succesfully we can use the handles to input and output values to and from the PCI_PIO. In this example we are doing all the IO directly from the application without using the pacer so we make calls to the functions BCTReadPort and BCTWritePort. If we get a status return of BCT_IO_PENDING there is already an IO operation pending on the port we specified and we use BCTWait to specify how long to wait for this previous IO operation to complete.

```
// Variable to store result
BCT_BYTE nVal;
char *sErrTxt;

// To read a port…
nStatus = BCTReadPort(&nInputHandle, &nVal);
if (nStatus == BCT_IO_PENDING)
{
      nStatus = BCTWait(&nInputHandle, INFINITE);
}
if ((nStatus != BCT_OK) && (nStatus !=
BCT_IO_PENDING))
{
      BCTErr2Txt(nStatus, sErrTxt);
      printf("Status: %s\n", sErrTxt);
      return(nStatus);
}

// To write a port…
nStatus = BCTWritePort(&nOutputHandle, 0x55);
if (nStatus == BCT_IO_PENDING)
{
      nStatus = BCTWait(&nOutputHandle, INFINITE);
}
if ((nStatus != BCT_OK) && (nStatus !=
BCT_IO_PENDING))
{
      BCTErr2Txt(nStatus, sErrTxt);
      printf("Status: %s\n", sErrTxt);
      return(nStatus);
}
```

3.4.6 Closing all open handles

When the application is to be terminated we must close all open handles and release the board id structure.

```
// Close the handles
nStatus = BCTClose(&nInputHandle);
// Check nStatus returned
nStatus = BCTClose(&nOutputHandle);
// Check nStatus returned
nStatus = BCTReleaseBoardId(&nPCIPIOBoardId);
// Check nStatus returned
```

When all the open handles have been closed and all board ID structures released it is safe to terminate the application. If these close calls are not made the handles will remain allocated and may result in strange behaviour of Windows.

3.5 ASYNCHRONOUS OPERATION

Many of the functions provided are asynchronous using the standard Win32 OVERLAPPED operations. This means that an operation could return with a status of BCT_IO_PENDING. If this occurs then **before** using any functions that will access the same device it is **imperative** that the application waits until the operation has finished. This can be achieved by using the BCTWait function which allows a time from 0ms to INFINITE to be specified for how long the function will wait for the I/O to complete. If the operation is not complete it will again return BCT_IO_PENDING. This permits the application to continue processing and periodically test for whether I/O is complete OR to block and wait until the I/O is complete before proceeding any further.

3.6 USING THE COUNTER TIMERS

The Counter Timers available on the PCI_PIO, PCI_DIO and PCI_ADC can be used in a number of ways. They are always programmed to use 16-bit counters. The following options are available:

1. When pacing (one of the timers free running to give a periodic interrupt), counters 0 and 1 are reserved to control the pacing functions. As one pacer can control functions on devices on multiple boards, this would leave counter 0 and 1 free on any other available boards in the system.

2. When pacing analogue input on the PCI_ADC, counter 2 is used. This must be the counter 2 on the ADC board (unlike the case with the generic pacing clock)

3. A particular counter (if available) can be programmed to count external events. Most of the counters allow 2 external pins to be connected and either of these can be used to clock the relevant counter – see the BCTProgramCounter function.

4. Any of the counters can be read so long as the program has a valid handle for the relevant device. In general this means that the pacing clocks cannot be read whilst they are being used for pacing as these are owned by the library and not by the application. The counter will have been started using BCTProgramCounter. As it is initially programmed with $0ffff_{16}$, counts down to 0 then wraps back to $0ffff_{16}$, any use of the counter will have to take account of the fact that the counter counts down. As the counters are 16-bit values they are read using the BCTReadPort16 function.

Note: The counters do not actually load their starting value $0ffff_{16}$ until they receive the first clock input, this means that any value read from the counter before it has started counting returns an undefined value.

3.7 PACING

In order to support transfer of blocks of data under control of the Counter/Timers (8254s) on the PCI_PIO, PCI_DIO and PCI_ADC boards, a number of pacer functions are provided. The following limitations will be placed on the hardware by the driver:

1. Clock 0 and 1 will always be cascaded together to give a total of 32bits for the Counter/Timer value

2. The input to this will always be the on board oscillator – 4MHz

3. The minimum time between I/Os is 1ms, the maximum is that achievable with 32 Bits and a clock rate of 4MHz (approximately 17.89 minutes).

To use pacer input / output the following sequence of events needs to take place:

1. Acquire the Pacer Clocks

2. Specify the operation to be synchronised with the Pacer Clock, this step should be repeated for as many operations as are required

3. Start the Pacer operation, specifying the time between I/Os in milliseconds

4. At the end, stop pacer - this will complete all the outstanding operations tied to the pacer and it will be necessary to return to step 1.

Each step must be carried out in sequence otherwise the routines will report an error reflecting the missed step.

Some of the Pacing Functions support double buffering where it is possible to have the Driver Reading/Writing to/from one buffer whilst the application is processing the other buffer. In order to support this, ALL data buffers passed to the pacer functions contain not only the actual buffer for the data but also a semaphore used to synchronise the use of the buffers.

These buffers, declared as type **BCT_BUFFER**, must be initialised and released using the BCTAllocate and BCTRelease functions which are similar in use to the standard C "malloc" and "free" functions.

If a buffer has been declared as follows:

```
BCT_BUFFER pBuf;
```

Then to access the actual data use

`pBuf.Buffer`, which is declared as an array of BCT_BYTE. To access the semaphore use `pBuf.Sema`, which is declared as a 32-bit unsigned word (BCT_DWORD).

In use the semaphore should be initialised to zero before calling the driver. The driver will use the first buffer, set its semaphore to non zero and switch to the other buffer. Each time the driver switches buffers it will set the semaphore for that buffer to a non zero value.

An application program should test the semaphore and only process the buffer when it has a non zero semaphore value and set the semaphore back to zero when it has finished processing it.

The pacing functions allow a single Pacer clock to be used to trigger multiple events. Note the following:

1. The driver will process each event in turn as the pacer interrupt occurs, the first to be processed will be the first added to the event list. If there are too many operations added to the pacer clock then it is possible that they will not be finished before the next pacer interrupt occurs. If this happens then that pacer interrupt will be ignored however, the currently active I/O will continue until the event list is completed.
2. The Pacer Clock does not need to be on the same board as the devices being paced.

3. Some of the functions are continuous, these will only be removed from the pacer queue when their associated device is closed or the Pacer is released. Releasing the Pacer Clock stops ALL of the activity paced on that clock, whereas closing the device only stops pacing for that particular device.

To allow different operations on the same pacer to run at different rates, an additional operation is supported where the number of Pacer interrupts to be ignored before carrying out the operation can be specified. For example a value of 0 means carry out the operation on every pacer interrupt. A value of 5 means ignore 5 pacer interrupts and carry out the operation on every 6^{th} interrupt.

Paced input from analogue inputs on the PCI_ADC is handled slightly differently to that for the other devices.

1. It uses only Counter/Timer 2

2. Only a single block of data can be captured (up to 4Gbytes in size!)

3. Capture is either “as quick as possible – as in Software, Level Triggered or Paced – driven by the output of Counter/Timer 2.

4. In the same way as the standard Pacer, if you are using a Counter/Timer it must be first “ Acquired” and then “Released”

3.8 USING DIGITAL INPUT AND OUTPUT

3.8.1 Ports A and B

The 8255 devices on the PCI_PIO and PCI_ADC have three ports that can be configured for input or output and in the case of port C the bits can be split between input and output.

Ports A and B are each configured as a single byte wide port either all as input or all as output in 8255 mode 0 operation. This configuration must be done before calls are made to write or read to the port using BCTInit8255Modes.

3.8.2 Split Port C

The 8255 devices on the PCI_PIO and the PCI_ADC have a Port C that can be programmed so that the low 4 bits and high 4 bits are used for Input or Output independently. This is controlled by the BCTInit8255Modes function, specifying ININ for 8 bit input, OUTOUT for 8 bit output and INOUT or OUTIN for split mode operation.

Apart from initial set up port C is used in the same way as the other two ports (A and B). When using the Bit Setting function BCTWriteBit, this will reject attempts to write to a bit set to input. When using any of the Input or Output functions, BCTWritePort, BCTReadPort or BCTAddPacerBlockIo, these will read or write 8 bit values. If the Port is "split" then,

- On output all 8 bits will be written to Port C but only the appropriate half of the byte will actually be placed on the output pins by the device
- On Input all 8 bits will be read from Port C, but only the appropriate half of the byte will contain valid information, the other half is undefined and no assumption should be made as to its contents.

So, although the Port can be split into two 4 bit halves all access to the port is made using 8-bit values and care should be taken that the data of interest is in the correct half of the 8 bit value read or written.

3.9 WATCHDOG TIMER

The PCI_WDT board operates in a different manner to the other boards supported by the driver. As with the other boards in the range, it is identified using a BCT_BOARD_ID and a handle is obtained using the BCTOpen function. The main difference is that all the functionality on the board is accessed as a single device.

This implementation does not support interrupts and the functionality is limited to:

- Reading and Writing the System Monitor Registers – see BCTSetWdt and BCTReadWdt.

The data that can be written is defined in the BCT_WDT_SETDATA data structure, the data that can be read is defined in the BCT_WDT_READDATA data structure, see data structure definitions is appendix A.

- Control Operations on the Watchdog are all carried out using the BCTWatchdog function.

The Monitor chip on the PCI_WDT does not support fast access and so the BCTSetWdt and BCTReadWdt routines will reject access less than 2 seconds apart. If you are monitoring using a loop of some sort, ensure that the routines are called at more than 2 second intervals.

4.0 DRIVER API FUNCTIONS

In order to take away the complexity of the Windows NT™ kernel mode device driver interface the drivers are supplied with an accompanying 32bit DLL which provides a number of library functions for interfacing to the drivers. All of these functions return an error code as detailed in the 'Error Codes' section of this manual and these should be checked when calls to the library have been made.

The behaviour of the DLL and driver is neither predictable nor supportable if error codes returned from API routines are ignored.

The error codes are detailed in a later section of this manual however it is recommended that the routine **BCTErr2Txt()** is used to translate error codes into their appropriate text forms in order for them to be displayed on the screen.

4.1 Function Overview

The library contains functions which can be divided into the following categories:

Management:	BCTAcquireAPacer
	BCTAllocate
	BCTClose
	BCTErr2Txt
	BCTFindSerialNo
	BCTGetBoardId
	BCTInitHandle
	BCTInitPacer
	BCTOpen
	BCTRelease
	BCTReleaseAPacer
	BCTReleaseBoardId
	BCTWait
Digital Functions:	BCTInit8255Modes
	BCTInitIsoDigModes
	BCTReadPort
	BCTReadPort16
	BCTWriteBit
	BCTWritePort
	BCTWritePort16
Analogue Functions:	BCTAutoCalAin
	BCTInitAOutModes
	BCTReadBlockAin
	BCTReadPortAin

Counter Functions:	BCTProgramCounter
Pacer Functions:	BCTAddPacerBlockIO BCTAddPacerFunction BCTStartPacer BCTStopPacer
Watchdog functions:	BCTReadWdt BCTSetWdt BCTWatchdog

4.2 Function Descriptions

The function prototypes are given below in C notation, using "Hungarian" type prefixes on variable names. This is consistent with the format used by Microsoft in their Windows programming manuals.

4.2.1 Management Functions

BCTAcquireAPacer

```
nError BCTAcquireAPacer(BCT_BOARD_ID
*pBoardId);
```

pBoardId Identifier returned from BCTGetBoardId

Attempts to gain access to Counter/Timer 2 on the board specified. If it fails an error will be returned

BCTAllocate

```
BCT_DWORD BCTAllocate(PBCT_BUFFER pBctBuf,
                      ULONG Length)
```

pBctBuf The address of a buffer structure

Length How many bytes to allocate

This routine is used to allocate the requested amount of memory and store a pointer to the buffer in the structure. This structure also contains the semaphore used to manage Double Buffering. All Buffers passed to the API must be encapsulated in a BCT_BUFFER structure.

BCTClose

```
nError BCTClose(PBCT_HANDLE pHandle)
```

pHandle A BCT specific handle for the board

Releases any resources associated with the device when it was opened

BCTErr2Txt

```
void BCTErr2Txt(BCT_DWORD nCode, char
*lpanTxt)
```

nCode	An error code returned from any of the BCT library routines
lpanTxt	A pointer to an area to copy an ASCII version of the Error Code into.

This function converts the error code returned by the DLL functions into the equivalent text string. These errors are defined in the chapter on error codes.

BCTFindSerialNo

```
nError BCTFindSerialNo(BCT_DWORD nSerial,
                       BCT_BOARDTYPE
                       *nBoardType,
                       BCT_WORD *pBoard)
```

nSerial	The unique serial number of the board to be identified
pBoardType	Address of a variable for the return of the unique identifier for the board type: PCI_PIO, PCI_DIO, PCI_ADC, PCI_WDT
pBoard	Address of a variable for the return of the driver derived board number of this particular type of board, starting from zero.

This function will scan all boards identified by the system during start-up. If it finds a Board with a matching serial number then it will return a BoardType and BoardNumber suitable for use in a subsequent call to BCTGetBoardId.

This routine is typically only used in those circumstances where more than one board of a given type will be installed in the system. The exact mechanism by which the serial numbers are provided to the application is implementation dependent. It could, for example, be achieved using the Registry, an initialisation file or even as a parameter on the command line to a program.

BCTGetBoardId

```
nError BCTGetBoardId(PBCT_BOARD_ID pBoardId,
                     BCT_BOARDTYPE
nBoardType,          BCT_WORD nBoard)
```

pBoardId	A validated descriptor for use in subsequent operations on this board
nBoardType	Unique identifier for the board type: PCI_PIO, PCI_DIO, PCI_ADC, PCI_WDT
nBoard	Which board of a particular type starting from 0

This function will validate the nBoardType and board number, make sure that such a device is present on the system and return a Handle for use in addressing the board.

This handle cannot be used for any I/O but simply identifies the board. See the comments in the section on identifying boards for the limitations in associating the Driver Based Board Number with the physical order of boards of the same type in any particular system.

BCTInitHandle

```
nError BCTInitHandle(PBCT_HANDLE pHandle)
```

pHandle A BCT specific handle for the board

This must be used to initialise a device handle before use, for example:

```
BCT_HANDLE handle,
BCTInitHandle(&handle);
```

BCTInitPacer

```
nError BCTInitPacer(PBCT_BOARD_ID pBoardId)
```

pBoardId Identifier returned from BCTGetBoardId

This will implicitly open the two clock devices to ensure they are available and prevent use by other parts of the application, they will be released when BCTStopPacer is called.

BCTOpen

```
nError BCTOpen(PBCT_HANDLE pHandle,
               PBCT_BOARD_ID pBoardId,
               BCT_DEVICETYPE nDevType,
               BCT_WORD nDev
               BCT_PORTTYPE nPort)
```

pHandle A BCT specific handle for the device returned by an implicit use of the CreateFile function

pBoardId Identifier returned from BCTGetBoardId, used to identify on which Board the device is located

nDevType Specifies an individual device type:

BCT_8255,	A digital I/O chip
BCT_8254,	A counter timer chip
BCT_ISODIG,	Isolated Digital I/O
BCT_AIN,	Analogue Input
BCT_AOUT	Analogue Output

nDev — Specifies which device on the Board is being addressed. It is a zero based number, for example, there are two 8255 devices on a PCI_PIO board these are numbered 0 and 1.

nPort — Specifies an individual port within the device, this can either be done using an integer, or one of the constants provided. All ports / channels are numbered in zero based sequence, 0, 1, 2, …. The following constants allow a more clearer description:

BCT_PORT_A,	Port A on an 8255
BCT_PORT_B,	Port B on an 8255
BCT_PORT_C,	Port C on an 8255
BCT_CLK_0,	Counter 0 on an 8254
BCT_CLK_1,	Counter 1 on an 8254
BCT_CLK_2,	Counter 2 on an 8254
ISO_DIG_LOW8,	Low 8-bits of I/O on a PCI_DIO
ISO_DIG_HIGH8,	High 8-bits of I/O on a PCI_DIO
ISO_DIG_ALL16,	16 bits of I/O on a PCI_DIO

BCT_CHAN_0,	Analogue Out, Channel 0
BCT_CHAN_1,	Analogue Out, Channel 1
BCT_CHAN_2,	Analogue Out, Channel 2
BCT_CHAN_3	Analogue Out, Channel 3

BCTRelease

```
BCT_DWORD BCTRelease(PBCT_BUFFER pBctBuf)
```

pBctBuf The address of a buffer structure

Release the memory allocated to the buffer in the BCT_BUFFER structure, note this does NOT release the structure only the buffer memory allocated to it.

BCTReleaseAPacer

```
nError BCTReleaseAPacer(BCT_BOARD_ID
*pBoardId);
```

pBoardId Identifier returned from BCTGetBoardId

Release the Counter Timer 2 acquired by BCTAcquireAPacer()

BCTReleaseBoardId

```
nError BCTReleaseBoardId(PBCT_BOARD_ID
pBoardId)
```

pBoardId Identifier returned from BCTGetBoardId

This function will release any resources reserved by the call to BCTGetBoardId.

BCTWait

```
nError BCTWait(PBCT_HANDLE pHandle,
               BCT_DWORD nDelay)
```

pHandle A BCT specific handle for the board

nDelay Time in milliseconds to wait for an I/O operation to complete.

This function is used to wait for any operation that returns BCT_IO_PENDING. If the routine times out before the I/O is complete it will return BCT_IO_PENDING.

The constant 0 for a time-out just tests whether the I/O is complete, a value of INFINITE will not return until the I/O is complete, any value in-between is possible.

4.2.2 Digital Functions

BCTInit8255Modes

```
nError BCTInit8255Modes(
                    PBCT_BOARD_ID pBoardId,
                    BCT_WORD nDev,
                    BCT_8255_MODES nMode,
                    BCT_DIRECTIONS nPortA,
                    BCT_DIRECTIONS nPortB,
                    BCT_DIRECTIONS nPortC)
```

pBoardId — Identifier returned from BCTGetBoardId

nDev — Specifies which of the 8255's on the board to access – e.g. 0, 1, …

nMode — Specifies the mode of operation for the 8255, defined as one of the following:

MODE_0 - PIO port is in mode 0

nPortA — Specifies the direction of the port A within the PIO

INPUT	Port is input
OUTPUT	Port is output
NOCARE	Port is not used

nPortB Specifies the direction of the port B within the PIO

INPUT	Port is input
OUTPUT	Port is output
NOCARE	Port is not used

nPortC Specifies the direction of the port C within the PIO

ININ	Port is all input
OUTOUT	Port is all output
INOUT	Upper nibble is input, lower output
OUTIN	Upper nibble is output, lower input
NOCARE	Port is not used

Each 8255 needs to be initialised to set up the mode, port direction, etc. If all six 8255 ports are required on a PCI_PIO then two calls need to be made to the BCTInit8255Modes function, once for the first 8255 (0) and once for second 8255 (1).

Note: This function is used to initialise the 8255 on ALL the boards, i.e. both the 8255's on a PCI_PIO and the single 8255 on a PCI_ADC.

BCTInitIsoDigModes

```
nError BCTInitIsoDigModes(
                        PBCT_BOARD_ID pBoardId,
                        BCT_WORD nDev,
                        BCT_8255_MODES nMode,
                        BCT_DIRECTIONS nPortA,
                        BCT_DIRECTIONS nPortB,
                        BCT_DIRECTIONS nPortC)
```

This function is used to initialise the isolated digital ports on a PCI_DIO card.

Although the PCI_DIO has fixed inputs and outputs the driver needs to be initialised as though it were an 8255 on a PCI_PIO. This means that a PCI_PIO could be changed for a PCI_DIO with minimal code changes. This conceptual view of the PCI_DIO requires the following parameters to the BCTInitIsoDigModes function call.

pBoardId	Identifier returned from BCTGetBoardId
nDev	Specifies which of the isolated digital inputs or outputs is to be initialised. This should always be 0.
nMode	Specifies the mode of operation for the PCI_DIO. This should always be set as MODE_0.

nPortA Specifies the direction of the port A within the DIO. Port A is the lowest 8 bits of the DIO.

INPUT	Port is input
OUTPUT	Port is output
BIDI	Port is in and output
NOCARE	Port is not used

nPortB Specifies the direction of the port B within the DIO. Port B is the upper 8 bits of the DIO.

INPUT	Port is input
OUTPUT	Port is output
BIDI	Port is in and output
NOCARE	Port is not used

nPortC Specifies the direction of the port C within the DIO. Port C is used to access all 16 bits of the DIO.

INPUT	Port is input
OUTPUT	Port is output
BIDI	Port is in and output
NOCARE	Port is not used

BCTReadPort

```
nError BCTReadPort(PBCT_HANDLE pHandle,
                   BCT_BYTE *pVal)
```

pHandle A BCT specific handle for the board

pVal A pointer to a variable to store the data returned from the port.

This routine can be used to read an 8-bit value from any device that supports 8-bit reads.

BCTReadPort16

```
nError BCTReadPort16(PBCT_HANDLE pHandle,
                     BCT_WORD *pVal)
```

pHandle A BCT specific handle for the board

pVal A pointer to a variable to store the data returned from the port.

This routine can be used to read a 16-bit value from any device that supports 16-bit reads.

BCTWriteBit

```
nError BCTWriteBit(PBCT_HANDLE pHandle,
                   BCT_BYTE nBit,
                   BCT_BYTE nVal)
```

pHandle A BCT specific handle for the board

nBit Specified the bit within the port (0 - 7)

nVal The value to be written to the port

This function can be used on any device that supports setting of individual bits, for example Port C on an 8255 or the Isolated Digital Output on a PCI_DIO.

BCTWritePort

```
nError BCTWritePort(PBCT_HANDLE pHandle,
                    BCT_BYTE nVal)
```

pHandle A BCT specific handle for the board

nVal The value to be written to the port

This routine can be used to write an 8-bit value from any device that supports 8-bit writes.

BCTWritePort16

```
nError BCTWritePort16(PBCT_HANDLE pHandle,
                      BCT_WORD nVal)
```

pHandle A BCT specific handle for the board

nVal The value to be written to the port

This routine can be used to write a 16-bit value from any device that supports 16-bit writes.

4.2.3 Analogue Functions

BCTAutoCalAin

```
nError BCTAutoCalAin(PBCT_HANDLE pHandle,
                     BCT_BYTE Gain,
                     BCT_WORD *lpnMeanZero,
                     BCT_WORD *lpnMeanFsd);
```

pHandle — A BCT specific handle for the board

nGain — The gain at which the Calibration is to be carried out,

PCIADC_AIN_GAIN_1	for gain of 1
PCIADC_AIN_GAIN_10	for gain of 10
PCIADC_AIN_GAIN_100	for gain of 100
PCIADC_AIN_GAIN_1000	for gain of 1000

lpnMeanZero — a pointer to return the calculated Mean Zero Value

lpnMeanFsd — a pointer to return the calculated 90% Full Scale Value

This routine will take 10 samples at the chosen gain with the input set to 0 Volts and return the average value. It will take 10 samples at a gain of 1 with the input value set at 4.0 Volts (80% FSD) and return the average.

Note the MeanFsd value is ALWAYS calculated at a gain of 1. These values allow the sampled data to be re-calibrated. The calibration should be carried out at the same gain setting as the samples.

BCTInitAOutModes

```
nError BCTInitAOutModes(
                    PBCT_BOARD_ID pBoardId,
                    BCT_WORD nDev,
                    BCT_WORD nChan0,
                    BCT_WORD nChan1,
                    BCT_WORD nChan2,
                    BCT_WORD nChan3);
```

pBoardId	Identifier returned from BCTGetBoardId, currently only the PCI_ADC has an Analogue Out device
nDev	Specifies which of the analogue Out devices to use. The PCI_ADC is considered to have 1 device with 4 channels, so this is typically 0
nChan0	Specifies the whether channel 0 provides voltage current outputs, BCT_AOUT_VOLTAGE or BCT_AOUT_CURRENT
nChan1	Specifies the whether channel 1 provides voltage or current outputs, BCT_AOUT_VOLTAGE or BCT_AOUT_CURRENT
nChan2	Specifies the whether channel 2 provides voltage or current driven, BCT_AOUT_VOLTAGE or BCT_AOUT_CURRENT

nChan3 Specifies the whether channel 3 provides voltage or current driven, BCT_AOUT_VOLTAGE or BCT_AOUT_CURRENT

This routine is used to initialise each of the four channels to a suitable state. Each of the four analogue output channels can provide constant voltage or current outputs.

Note, all four channels are initialised in a single call to the routine

BCTReadBlockAin

```
nError BCTReadBlockAin(
                      BCT_BOARD_ID *pBoardId,
                      PBCT_HANDLE pHandle,
                      BCT_BYTE nChan,
                      BCT_BYTE nGain,
                      BCT_BYTE nMode,
                      PBCT_BUFFER pBuffer,
                      BCT_DWORD nSamples,
                      BCT_WORD nTime);
```

pBoardId Identifier returned from BCTGetBoardId

pHandle A BCT specific handle for the device

nChan The channel to be read, 0-15 for single ended and 0-7 for differential inputs. If reading from multiple channels then this is the maximum channel number to read

nGain — The gain at which the values are to be read,

PCIADC_AIN_GAIN_1 for gain of 1
PCIADC_AIN_GAIN_10 for gain of 10
PCIADC_AIN_GAIN_100 for gain of 100
PCIADC_AIN_GAIN_1000 for gain of 1000

nMode — whether using single ended or differential inputs and whether to collect from a single channel or multiple channels

PCIADC_AIN_MODE_SINGLE, PCIADC_AIN_MODE_DIFFERENTIAL or PCIADC_AIN_AUTOSEL for multi. ports

If more than one option is being used they should be “bitwise or’d” together. e.g.

```
PCIADC_AIN_MODE_DIFFERENTIAL |
PCIADC_AIN_AUTOSEL
```

pBuffer — Where the data should be returned, see the comments on using BCT_BUFFER, only the data buffer itself is used, the semaphore is ignored.

nSamples — Number of samples to be collected. Note each sample is returned as a 16 bit word. The buffer should be allocated with twice as many bytes as samples

nTime	Time in microseconds between samples. A value of 0 means collect the data as fast as it can be converted.

This routine is used to perform paced inputs from the analogue to digital converter on the PCI_ADC. It allows either rapid collection or paced collection as described above.

If a timer of 0 is specified then using multiple channels will not be permitted as it does not allow sufficient settling time as each channel is switched to the converter.

If a non-zero time is specified it will be checked to ensure that it allows sufficient settling time between samples. The times used can be found in the Hardware manual for the PCI_ADC board.

There is a maximum possible time between samples of 16,384 microseconds. If capture is required at a slower rate then the routine BCTReadPortAin should be used repeatedly.

BCTReadPortAin

```
nError BCTReadPortAin(PBCT_HANDLE pHandle,
                      BCT_BYTE nChan,
                      BCT_BYTE nGain,
                      BCT_BYTE nMode,
                      BCT_WORD *pVal)
```

pHandle — A BCT specific handle for the board

nChan — The channel to be read, 0-15 for single ended and 0-7 for differential inputs

nGain — The gain at which the value is to be read,

PCIADC_AIN_GAIN_1 for gain of 1
PCIADC_AIN_GAIN_10 for gain of 10
PCIADC_AIN_GAIN_100 for gain of 100
PCIADC_AIN_GAIN_1000 for gain of 1000

nMode — whether using single ended or differential inputs:

PCIADC_AIN_MODE_SINGLE or
PCIADC_AIN_MODE_DIFFERENTIAL

pVal — A pointer to a variable to store the data returned from the port.

This routine is used to read a 12-bit raw data value from the Analogue In Channels on the PCI_ADC. The actual value returned is 16-bits long, the upper 4-bits are the channel number - see the PCI_ADC hardware manual for further details.

4.2.4 Counter Functions

BCTProgramCounter

```
nError BCTProgramCounter(PBCT_HANDLE pHandle,
                         BCT_BYTE nSource)
```

pHandle	A BCT specific handle for the board
nSource	Which of the external counter inputs to use : BCT_EXTERN_COUNTER_INPUT1 or BCT_EXTERN_COUNTER_INPUT2

This function programs a Counter Timer to $0ffff_{16}$ and starts it counting down. It will count each time the specified input goes low. The nSource parameter will depend on which board is used. The relevant hardware manual indicates what sources may be used as inputs to the counter.

BCT_EXTERN_COUNTER_INPUT1 means the first of the inputs, this will be the first external input entry in the table of inputs for this counter. It is possible that a particular board may not support external counter inputs on all of the counters. For example, the PCI_ADC board only supports external counter inputs on counters 1 and 2.

It is also necessary to ensure that any conditions imposed on the board where the Counter Input pin is shared with another device are met. For example the Counter Input pins on the PCI_PIO are shared with Port

C on the 1st 8255 and Ports B and C on the 2nd 8255, these must be set as inputs to avoid contention.

The relevant Hardware manual MUST be checked and the necessary conditions met.

4.2.5 Pacer Functions

BCTAddPacerBlockIo

```
nError BCTAddPacerBlockIo(PBCT_HANDLE pHandle,
                          PBCT_BOARD_ID
                          pBoardId,
                          BCT_WORD nOperation,
                          PBCT_BUFFER pBuffer1,
                          PBCT_BUFFER pBuffer2,
                          BCT_DWORD nBytes,
                          BCT_DWORD nCount)
```

pHandle	A BCTspecific Handle identifying the device to be used
pBoardId	Identifier returned from BCTGetBoardId() for the board with the Pacer to be used. This may not be the same as the board on which the device is located.
nOperation	BLOCK_SINGLE_READ read single block and then complete I/O. BLOCK_DOUBLE_BUFFER_READ Initiate a continuous double buffered read – see the section on double buffering for how this affects applications programs.

BLOCK_SINGLE_WRITE Write a single block and then complete the I/O.

BLOCK_DOUBLE_BUFFER_WRITE Initiate a continuous double buffered write – see the section on double buffering for how this affects applications programs.

BLOCK_REPEATED_WRITE Write the same block continuously, going back to the start of the block after the last value is written.

The operations will be checked against the current settings for that device and rejected if they do not match - e.g. a BLOCK_WRITE to a port set for input.

pBuffer1	Pointer to buffer containing data and semaphore, 1st buffer when double buffering
pBuffer2	Pointer to 2nd buffer containing data and semaphore when double buffering, ignored for single buffer transfers
nBytes	Size of Buffer in Bytes - when double buffering, both buffers are the same size

nCount	The number of times the Pacer Clock interrupt should be ignored before carrying out the operations. A value of 0 means the operation takes place on every pacer interrupt, 1 means skip interrupt, operate on interrupt, skip interrupt, operate on interrupt, …
	Adds a single operation to be carried out by the driver every time the Pacer Clock interrupts. This routine can be called repeatedly before issuing BCTStartPacer() in order to carry out multiple functions on each Pacer Interrupt. Any attempt to call it once the pacer has started or before the Pacer Clock has been initialised will return an error.

Supported Devices

8255	All of the pacer block IO functions are supported on the 8255 (with the exception that only 8 bit transfers are supported on Port C)
ISODIG	All the functions are supported on the PCI_DIO, Isolated I/O ports with the exception of Port C (all 16 bits) which is not supported at all.

Analogue Out Only the BLOCK_SINGLE_WRITE and BLOCK_CONTINUOUS_WRITE operations are supported on the Analogue Out Channels on the PCI_ADC.

Analogue In None of the functions are supported on the Analogue Input Channels on the PCI_ADC (see the separate Analogue Input, Pacing functions.)

BCTAddPacerFunction

```
nError BCTAddPacerFunction(PBCT_BOARD_ID
                           pBoardId,
                           BCT_HANDLER
                           nOperation, BCT_DWORD
                           *pVal)
```

pBoardId Identifier returned from BCTGetBoardId

nOperation Some simple operation code
READ_PACER reads the current pacer interrupt count.

*pVal Return value from READ_PACER. If *pVal > 0 then wait until *pVal pacer interrupts have occurred before returning the interrupt count.

This routine is used to provide generic pacer functions that can be described by an operation code and an optional 32 bit pointer.

BCTStartPacer

```
nError BCTStartPacer(PBCT_BOARD_ID pBoardId,
                     BCT_DWORD nInterval)
```

pBoardId Identifier returned from BCTGetBoardId

nInterval Specifies the interval in milliseconds, allowable values are 1 to 1,073,000, i.e. maximum interval between samples is approximately 17.89 minutes.

The Onboard clock runs at 4MHz, cascading two 16 bit counters together gives the maximum time interval.

BCTStopPacer

```
nError BCTStopPacer(PBCT_BOARD_ID pBoardId)
```

pBoardId Identifier returned from BCTGetBoardId

Stops the appropriate Counter Timers and releases (closes) the two clock devices opened by the BCTInitPacerClock function. Any pacing operations still outstanding will be terminated at this time.

4.2.6 Watchdog timer functions

BCTReadWdt

```
BCT_DWORD BCTReadWdt(PBCT_HANDLE pHandle,
                     PBCT_WDT_READDATA pData)
```

pHandle — A BCT specific handle for the device to be used

pData — Data returned from the watchdog timer board

This function returns current data from the watchdog timer board.

BCTSetWdt

```
BCT_DWORD BCTSetWdt(PBCT_HANDLE pHandle,
                    PBCT_WDT_SETDATA pData)
```

pHandle — A BCT specific handle for the device to be used

pData — Control Information to send to the watchdog timer Board

This function sets the working parameters for the watchdog timer board.

BCTWatchdog

```
BCT_DWORD BCTWatchdog(PBCT_HANDLE pHandle,
                      WATCHDOG_OPERATION
                      nAction,
                      BCT_BYTE *pData)
```

pHandle A BCT specific handle for the device to be used

nAction Watchdog Operation to carry out:
BCT_WD_WRITE_TIMEOUT
BCT_WD_REFRESH_TIMEOUT
BCT_WD_WRITE_ENABLE_MASK
BCT_WD_WRITE_OUTPUT_MASK

pData Pointer to Byte to read/write to the Watchdog

5.0 EVENT LOG MESSAGES

In the event of the driver failing to load when the system is started up then a message will be logged into the Windows NT event log. This can be viewed using the Event Log viewer found on the administrative tools option on the start menu.

Detailed below are the most common messages and their probable cause. If any other message is logged please call your supplier for more information.

ExAllocPool for Resource List failed:
The non paged memory small is too small for the amount of memory the driver has requested. This can be overcome by adding more system memory or by modifying the memory allocation settings in the registry. NOTE: If making changes to the registry then ensure that all registry files are backed up prior to making changes.

Failed to detect any Supported Boards
The driver could not find any supported PCI data acquisition cards when performing a sweep of PCI space. Ensure that the boards are inserted correctly.

Too many Boards Found
The driver has found too many boards and has been unable to create the controllers for them. There is a maximum of 10 boards that can be supported by the driver.

NT Failed to assign the PCI resources for this card.
There is a conflict between the resources asked for by the board and another driver in the NT system. Use bc_probe and NT diagnostic to resolve the conflict.
Failed to find the Base I/O Address for this Board.
The driver could not get a valid base address for the board. Ensure that the boards are all inserted correctly and are being allocated resources correctly.

Failed to find an IRQ for this board.
The driver could not get a valid hardware interrupt for the board. Ensure that the boards are all inserted correctly and are being allocated resources correctly.

Failed to write to the PCI command register
There is a hardware problem with the PCI data acquisition card. Please contact your supplier for further details.

Failed to find the expected number of I/O Base Address Registers
The driver could not find the required number of base addresses for the card installed. Ensure that the card is functioning correctly.

5.1 ERROR CODES

All functions within the library (except BCTErr2Txt) return an error code to give the status or result of the function call. It is imperative that the application program checks and acts upon these error codes to ensure correct operation.

Each of the error codes and its text based representation are detailed below with details of the cause of the error.

0 - BCT_OK
Function call was successful.

1 - BCTERR_BOARD_BUSY
Not used in this library.

2 - BCTERR_BOARD_NOT_REQUESTED
Not used in this library.

3 - BCTERR_BOARD_ALREADY_REQUESTED
Not used in this library.

4 - BCTERR_NO_GLOBAL_MEMORY
Not used in this library.

5 - BCTERR_TOO_MANY_BOARDS
Not used in this library.

6 - BCTERR_GAINS_NOT_SUPPORTED
Not used in this library.

7 - BCTERR_INVALID_GAIN_VALUE
The gain value specified in a call to BCTReadPortAin or BCTReadBlockAin is not valid. Use the gain constants defined in the header files.

8 - BCTERR_NO_ANALOG_CHANNELS
Not used in this library.

9 - BCTERR_CHANNEL_NOT_ANALOG
Not used in this library.

10 - BCTERR_INVALID_RANGE
Not used in this library.

11 - BCTERR_NO_ANALOG_INPUTS
Not used in this library.

12 - BCTERR_CHANNEL_NOT_ANALOG_INPUT
Not used in this library.

13 - BCTERR_NULL_POINTER
A pointer that is required by a function within the driver is NULL. Check that the pointer has been initialised correctly.

14 - BCTERR_NO_ANALOG_OUTPUTS
Not used in this library.

15 - BCTERR_CHAN_NOT_ANALOG_OUTPUT
Not used in this library.

16 - BCTERR_VALUE_OUT_OF_RANGE
Not used in this library.

17 - BCTERR_ILLEGAL_NUM_CHANS
Not used in this library.

18 - BCTERR_NO_IRQ_AVAILABLE
Not used in this library.

19 - BCTERR_ILLEGAL_CHANS_IN_ARRAY
Not used in this library.

20 - BCTERR_ILLEGAL_NUM_SCANS
Not used in this library.

21 - BCTERR_LOST_DATA_IN_ISR
Not used in this library.

22 - BCTERR_ILLEGAL_FREQ
Not used in this library.

23 - BCTERR_NO_PROG_DIGITAL
Not used in this library.

24 - BCTERR_INVALID_CHARS
Not used in this library.

25 - BCTERR_INVALID_NUM_CHARS
Not used in this library.

26 - BCTERR_PORT_NOT_BIDIRECTIONAL
Not used in this library.

27 - BCTERR_NOT_DIGITAL_INPUT
Not used in this library.

28 - BCTERR_ILLEGAL_PORT
Not used in this library.

29 - BCTERR_PORT_NOT_INPUT
Not used in this library.

30 - BCTERR_ILLEGAL_BIT
Not used in this library.

31 - BCTERR_NOT_DIGITAL_OUTPUT
Not used in this library.

32 - BCTERR_ILLEGAL_DIGOUT_VALUE
Not used in this library.

33 - BCTERR_PORT_NOT_OUTPUT
Not used in this library.

34 - BCTERR_NO_COUNTERS
Not used in this library.

35 - BCTERR_INVALID_COUNTER
A counter specified in a call to BCTProgramCounter is invalid. Check that the counter being requested is supported by the board being used.

36 - BCTERR_INVALID_REF_FREQ
Not used in this library.

37 - BCTERR_INVALID_OUTPUT_FREQ
Not used in this library.

38 - BCTERR_NOT_WATCHDOG
Not used in this library.

39 - BCTERR_UNRECOGNISED_BOARD
The board type requested is not valid. Check that the board type being used is in the valid list in the header files.

40 - BCTERR_INVALID_BASEADDR
Not used in this library.

41 - BCTERR_INVALID_IRQ
Not used in this library.

42 - BCTERR_INVALID_DMACHAN
Not used in this library.

43 - BCTERR_NO_DATA
Not used in this library.

44 - BCTERR_STILL_ACQUIRING
Not used in this library.

45 - BCTERR_NO_DMA_AVAILABLE
Not used in this library.

46 - BCTERR_DMA_IN_USE
Not used in this library.

47 - BCTERR_BUFFER_TOO_SMALL
The buffer size being specified in pacer functions is too small. Check the size of the buffer being requested and increase as necessary.

48 - BCTERR_INVALID_MODE_VALUE
The mode specified for analogue channels is not valid. Check that the mode (single ended or differential) is specified using one of the constants defined in the header file.

49 - BCTERR_HARDWARE_MISSING
Not used in this library.

50 - BCTERR_UNSUPPORTED_OS

The library found that the current operating system is not supported by the driver. The operating system should be Windows NT™ v4.0.

51 - BCTERR_GETSERIALNO_FAILED

The driver failed to obtain the serial number from the hardware.

52 - BCTERR_FAILED_RELEASEBOARDID

The call to BCTReleaseBoardId failed.

53 - BCTERR_INVALID_SERIALNO

The serial number specified in a call to BCTFindSerialNo has not been found. Check that the serial number specified is one of the boards in the system by checking the number written on the PCB.

54 - BCTERR_UNRECOGNISED_DEVICECODE

The device code specified in a call to BCTOpen is not valid. Ensure that the code being used is from the valid list in the header file.

55 - BCTERR_UNRECOGNISED_PORTCODE

The port code specified in the call to BCTOpen is not valid for the device type being opened. Ensure that the port code being used is from the valid list in the header file and is supported by the device being opened.

56 - BCTERR_UNRECOGNISED_DEVICE

The device specified in a call to BCTOpen is not valid.

57 - BCTERR_FAILEDCLOSE

The driver failed to close the handle specified in a call to BCTClose. Check that the handle is valid.

58 - BCTERR_READPORT_FAILED

The driver failed to read from a port. Check that the devices have been initialised correctly, that the handles are correct i.e. not reading from a handle assigned to an output port.

59 - BCTERR_WRITEPORT_FAILED

The driver failed to write to a port. Check that the devices have been initialised correctly, that the handles are correct i.e. not writing to a handle assigned to an input port.

60 - BCTERR_INIT8255MODES_FAILED

The driver failed to initialise the 8255 ports. Check that the handles and board id's passed to the BCTInit8255Modes call are correct

61 - BCTERR_NTDRIVER_DEVICESOUTOFORDER

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

62 - BCTERR_DEVICECODENOTFOUND

The driver can not find the specified device. Check that the device specified is valid and is present on the installed hardware.

63 - BCTERR_DEVICEOPEN

The device specified in a BCTOpen call is already open and in use by another process or part of the application. Check that the device being opened is correct or close down the other process.

64 - BCTERR_INVALID8255MODE

A mode specified for the 8255 is not valid. Check that the mode specified is valid for an 8255.

65 - BCTERR_INVALIDIODIR

The direction specified for an I/O operation is not valid. I.e. writing to an input port or reading from and output port. Check the port directions specified and that the handles used for reading and writing are correct.

66 - BCTERR_INVALID_BITNO

The bit specified to the BCTWriteBit function is not valid. Check the number of bits available on the port and ensure that the number specified in the write bit call is valid.

67 - BCTERR_INVALID_BITSET

The value specified for the bit is not valid. Ensure that the value is ‘0’ or ‘1’.

68 - BCTERR_WRITEBIT_FAILED

The call to BCTWriteBit failed. Check that the handle and board ID structure for the board are correct and that the call is to an output not an input port

69 - BCTERR_INVALID_PACERCLOCK
The clock period specified in the call to BCTStartPacer is invalid. Use a valid value for the pacer clock period.

70 - BCTERR_NOPACERCLOCK
The board specified in a call to BCTInitPacer does not have an 8254 present and can not support pacer functions. Use a board that supports pacers.

71 - BCTERR_PACERCLOCKINUSE
The pacer clock specified in a call to BCTInitPacer is already in use. Use a different pacer that is not being used.

72 - BCTERR_INITPACER_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

73 - BCTERR_STOPPACER_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

74 - BCTERR_ENABLEINTERUPT_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

75 - BCTERR_INITCLOCK_FAILED
The call to the BCTInitClock function failed.

76 - BCTERR_UNEXPECTED_CLOCKNO

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

77 - BCTERR_UNSUPPORTED_BOARD

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

78 - BCTERR_ADDPACER_FAILED

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

79 - BCTERR_PACER_CLOCK_HANDLES_INUSE

The handle specified in a call to BCTInitPacer is already in use. Check that the handle being used is correct and is not in use elsewhere.

80 - BCTERR_PACER_NOT_INITIALISED

The pacer being requested for block I/O has not been initialised. Check that the pacer has been initialised with a call to BCTInitPacer.

81 - BCTERR_INVALID_INTERRUPT_INDEX

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

82 - BCTERR_FAILED_ENABLE_INTERRUPT

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

83 - BCTERR_FAILED_ADD_ACTION

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

84 - BCTERR_DISABLEINTERUPT_FAILED

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

85 - BCTERR_CANCELLED

The I/O operation was cancelled by the operating system. Check that all the open files are closed prior to application exit.

86 - BCTERR_FAILED_PROGRAM_CNTRLCTRL

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

87 - BCTERR_INVALID_BYTECOUNT

Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

88 - BCTERR_FAILED_MAP_BUFFER
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

89 - BCTERR_FAILED_LOCK_BUFFER
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

90 - BCTERR_FAILED_MAP_BUFFER_SYS
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

91 - BCTERR_INTERNAL_DRIVER_ERROR
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

92 - BCTERR_FAILED_CREATE_OVERLAP_EVENT
An error has occurred within the DLL / WIN32 environment. Restart the system to correct the problem.

93 - BCTERR_FAILED_RESET_OVERLAP
An error has occurred within the DLL / WIN32 environment. Restart the system to correct the problem.

94 - BCT_IO_PENDING
An I/O operation is still in progress. On detecting this condition use the BCTWait procedure to wait for the I/O to complete

95 - BCTERR_WAIT_FAILED
The call to BCTWait failed.

96 - BCTERR_ALLOCATE_FAILED
The driver or library failed to allocate memory for the requested buffer.

97 - BCTERR_RELEASE_FAILED
The driver or library failed to release the memory used by an allocated buffer.

98 - BCTERR_INVALID_IO_DIRECTION
An port has been set to a direction, INPUT, OUTPUT, or BIDI that the port does not support. Check the port directions specified in the initialisation functions.

99 - BCTERR_INIT_ISODIG_MODES_FAILED
Failed to initialise the digital I/O on a PCI_DIO

100 - BCT_NOTIMPLEMENTED
The function requested is not available within this release of driver.

101 - BCTERR_INVALID_DEVICE_NUMBER
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

102 - BCTERR_HANDLER_ABORTED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

103 - BCTERR_FAILED_CLEAR_HANDLER
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

104 - BCTERR_UNSUPPORTED_8255_MODE
An operation mode specified for the 8255 in BCTInit8255Modes is not supported. The modes allowed are detailed in the manual with the BCTInit8255Modes function definition.

105 - BCTERR_UNSUPPORTED_A_DIRECTION
The direction specified for port A in a call to BCTInit8255Modes or BCTInitIsoDigModes is invalid. Check that the port supports the mode specified.

106 - BCTERR_UNSUPPORTED_B_DIRECTION
The direction specified for port B in a call to BCTInit8255Modes or BCTInitIsoDigModes is invalid. Check that the port supports the mode specified.

107 - BCTERR_UNSUPPORTED_C_DIRECTION
The direction specified for port C in a call to BCTInit8255Modes or BCTInitIsoDigModes is invalid. Check that the port supports the mode specified.

108 - BCTERR_UNSUPPORTED_ISODIG_MODE
An operation mode specified for the PCI_DIO in BCTInitIsoDig Modes is not supported. The modes allowed are detailed in the manual with the BCTInitIsoDigModes function definition.

109 - BCTERR_BLOCKIO_NOT_SUPPORTED
Block I/O operations are not supported on this device.

110 - BCTERR_ILLEGAL_COMBINATION
The combination of settings for an analogue output port is invalid. Check that only one mode setting is used.

111 - BCTERR_MISSING_OPTION
An option required for a function has not been specified. Ensure that all required options are specified in all calls to functions.

112 - BCTERR_INITAOUTMODES_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

113 - BCTERR_INVALID_CHANNEL_NUMBER
The channel number specified in an analogue function call is invalid. Check that the channel numbers specified are valid for the hardware being used.

114 - BCTERR_INPUT_CONVERSION_TIMEOUT
The conversion on the analogue input timed out.

115 - BCTERR_FIFO_ERROR
A problem was encountered with the FIFO. Typically this occurs when continuing to read data from the FIFO when it is no longer being filled.

116 - BCTERR_AUTOSEL_NOT_SUPPORTED
The AUTOSEL option is not valid for the hardware type being addressed. Use a mode that is supported by the hardware.

117 - BCTERR_INVALID_SAMPLE_FREQUENCY
The frequency set for samples is not valid. Check the user documentation for valid values.

118 - BCTERR_READBLOCK_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

119 - BCTERR_PACER_CLOCK_HANDLE_INUSE
The handle specified for a pacer clock is in use. Check the usage of handles within the application and allocate another if necessary.

120 - BCTERR_FIFO_OVERFLOW
The FIFO has overflowed. The application is not reading data from the FIFO fast enough to keep up with the speed it is being added.

121 - BCTERR_AUTOCAL_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

122 - BCTERR_AUTOCAL_NOT_SUPPORTED
Auto calibration is not supported on the board specified.

123 - BCTERR_INVALID_COUNTER_VALUE
The value specified for a counter is not valid. Check the appropriate user documentation for valid values.

124 - BCTERR_INVALID_COUNTER_INPUT
The pin or port specified to be the input to the counter is not valid. Check the user documentation for valid input pins.

125 - BCTERR_SETWDT_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

126 - BCTERR_READWDT_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

127 - BCTERR_WDTACCESS_TOOSOON
The watchdog timer functions are being called more frequently than every two seconds. Reduce the frequency with which the functions are called.

128 - BCTERR_ACCESS_WDT_FAILED
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

129 - BCTERR_INVALID_WDT_OPERATION
Driver internal fatal error. The driver and / or hardware encountered an un-resolvable problem. Check all hardware is valid and restart system.

200 - BCTWARN_OUTPUT_VALUES_CLIPPED
Not used in this library.

300 - BCT_RUNNING
Not used in this library.

301 - BCT_FINISHED
Not used in this library.

302 - BCT_IDLE
Not used in this library.

303 - BCT_OVERRUN
Not used in this library.

5.2 BCTGETLASTERROR

If a function call returns a fatal error it may be possible to extract further information from the driver using the function call BCTGetLastError. This function is defined as follows:

```
BCT_DWORD BCTGetLastError(BCT_HANDLE *pHandle,
                          BCT_DWORD *pErrCode)
```

pHandle The handle used in the function returning the fatal error.

pErrCode The internal Windows NT™ code.

If a function call returns a fatal error then a call should be made to BCTGetLastError and contact Technical Services with the resulting error code. They will then be able to determine the cause of the problem.

NOTE: The last error is only updated when an error occurs and is only supported by the simple I/O functions and not by all of the pacer functions.

A.0 LIBRARY DEFINED TYPES

The library introduces a number of simple data types and structures that are used within the system. The behaviour of the library and driver are neither predictable nor supportable if other data types are used inappropriately.

A.1 Platform Independent Data Types

These platform independent types allow the drivers and application software to be ported easily to systems running Windows NT™ on none Intel™ processor hardware platforms.

BCT_INT8	8-bit Signed Integer
BCT_INT16	16-bit Signed Integer
BCT_INT32	32-bit Signed Integer
BCT_BYTE	8-bit unsigned data item
BCT_WORD	16-bit unsigned data item
BCT_DWORD	32-bit unsigned data item
PBCT_BUFFER	Pointer to a block of 8-bit unsigned data

A.2 Enumerated Types

The BLUECHIP.H file defined a number of enumerated types that contain the constant definitions for board type, ports, etc. In BLUECHIP.BAS these are defined as zero ordered lists of global constants as Visual Basic v4.0 has no support for enumerated types.

These types are defined as follows:

```
typedef enum {
      BCT_8255,
      BCT_8254,
      BCT_ISODIG
} BCT_DEVICETYPE;

typedef enum {
   MODE_0,  // All ports on the PIO are mode 0
   MODE_1,  // Ports A & B are mode 1 IO ports with
            // port C as control
   MODE_2,  // Port A is bi-directional IO with
            // port C as control
   MODE_20, // Ports A and C are mode 2 as MODE_2
            // but port B is used as mode 0 IO.
   MODE_21  // Ports A and C are mode 2 as MODE_2
            // but port B is used as mode 1.
} BCT_8255_MODES;

typedef enum {
   INPUT,      // Port is input
   OUTPUT,     // Port is output
   BIDI,       // Port is bi-directional ie mode 2)
   NOCARE,     // Port is not used in application -
               // don't care how setup
   ININ,       // Port C is all input
   OUTOUT,     // Port C is all output
   INOUT,      // Port C Upper nibble is input,
               // lower nibble is output
   OUTIN,      // Port C Upper nibble is output,
               // lower nibble is input
```

```
} BCT_DIRECTIONS;

typedef enum {
      NO_HANDLER,
      READ_PACER,
      BLOCK_SINGLE_READ,
      BLOCK_DOUBLE_BUFFER_READ,
      BLOCK_SINGLE_WRITE,
      BLOCK_DOUBLE_BUFFER_WRITE,
      BLOCK_REPEATED_WRITE
} BCT_HANDLER;
```

A.3 Structure Definitions

Except where expressly documented these structures should be considered as Opaque and no guarantee is made that their contents will not change in future releases.

BCT_HANDLE structure. A valid handle is required for each function being called on a board. Variables of type BCT_HANDLE will need to be defined within the application program and be passed to functions within the API. However, the application program should never manipulate the contents of the structure.

```
typedef struct _BCT_HANDLE {
      HANDLE            Handle;
      OVERLAPPED        Overlap;
      BCT_DWORD         DeviceType;
} BCT_HANDLE, *PBCT_HANDLE;
```

BCT_BOARD_ID structure. A valid BCT_BOARD_ID structure is required for each PCI data acquisition card being used within the application. Again, the application program should not manipulate the values within the structure.

```
typedef struct _BCT_BOARD_ID {
      HANDLE            Handle;
      BCT_DWORD         SerialNo;
      char              DevName[30];
      char              Clock0Name[30];
      char              Clock1Name[30];
      BCT_HANDLE        hClock0;
      BCT_HANDLE        hClock1;
      BCT_DWORD         Index;
      BCT_DWORD         DeviceType;
} BCT_BOARD_ID;
```

BCT_BUFFER structure. This is a buffer of data passed to the pacer functions. The buffer should be requested using BCTAllocate and then be filled in by the application program. The semaphore flag within the BCT_BUFFER structure is used when using double buffering to show when the buffer is full.

```
typedef struct _BCT_BUFFER {
      ULONG             Length;
      ULONG             Sema;
      BCT_BYTE          Buffer[];
} BCT_BUFFER, *PBCT_BUFFER;
```

The BCT_WDT_SETDATA data structure defines the following values that can be written to the watchdog timer board:

```
typedef struct _BCT_WDT_SETDATA {
  BCT_BYTE  IN0_High_3_3V;     // 3.3V Line High
                               // Limit
  BCT_BYTE  IN0_Low_3_3V;      // 3.3V Line Low
            Limit
  BCT_BYTE  IN1_High_5V;       // 5V Line High Limit
  BCT_BYTE  IN1_Low_5V;        // 5V Line, Low Limit
  BCT_BYTE  IN2_High_12V;      // 12V Line High
                               // Limit
  BCT_BYTE  IN2_Low_12V;       // 12V Line Low Limit
  BCT_BYTE  IN3_High;
  BCT_BYTE  IN3_Low;
  BCT_BYTE  IN4_High;
  BCT_BYTE  IN4_Low;
  BCT_BYTE  IN5_High_Minus12V;// –12V High Limit
  BCT_BYTE  IN5_Low_Minus12V; // –12V Low Limit
  BCT_BYTE  IN6_High;
  BCT_BYTE  IN6_Low;
  BCT_BYTE  OverTempHigh;      // Over Temp. Setting
  BCT_BYTE  TempHysteresisLow;// Temp. hysteresis
                               // val.
  BCT_BYTE  Fan1Count;         // Fan1 Count Limit
  BCT_BYTE  Fan2Count;         // Fan2 Count Limit
  BCT_BYTE  Fan3Count;         // Fan3 Count Limit
  BCT_BYTE  FanRpm;            // Fan/Rpm Control
                               // Byte
}
```

The BCT_WDT_READDATA data structure defines the following values that can be read from the watchdog timer:

```
typedef struct BCT_WDT_READDATA {
  BCT_BYTE  IN0_3_3V;      // Current 3.3V Line
                           // Value
  BCT_BYTE  IN1_5V;        // Current 5V Line Value
  BCT_BYTE  IN2_12V;       // Current 12V Line
                           // Value
  BCT_BYTE  IN3;
  BCT_BYTE  IN4;
  BCT_BYTE  IN5_Minus12V;  // Current –12V Line
                           // Value
  BCT_BYTE  IN6;
  BCT_BYTE  Temperature;   // Current Temp. value
  BCT_BYTE  Fan1Count;     // Current Fan1 Count
  BCT_BYTE  Fan2Count;     // Current Fan2 Count
  BCT_BYTE  Fan3Count;     // Current Fan3 Count
  BCT_BYTE  ExternalInput; // Bit 0 is the current
                           //external Input Value
  BCT_BYTE  InterruptStatus1;
  BCT_BYTE  InterruptStatus2;
}
```