WDMDRV

WDM Drivers
for PCI Data Acquisition Cards

User Manual

WDMDRV

User Manual

Document Part N°	0127-1033
Document Reference	01271033.doc
Document Issue Level	2.0
Manual covers Drivers identified	v6.0

All rights reserved. No part of this publication may be reproduced, stored in any retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopied, recorded or otherwise, without the prior permission, in writing, from the publisher. For permission in the UK contact your supplier.

Information offered in this manual is correct at the time of printing. The supplier accepts no responsibility for any inaccuracies. This information is subject to change without notice.

All trademarks and registered names acknowledged.

Amendment History

Issue Level	Issue Date	Author	Amendment Details
1.0	01/08/2000	AJP	First Release
2.0	15/09/2005	MJF	Added XP references to supported operating systems. Correction to 3.4.5 Using the ports, - Handle for writing ports is nOutoutHandle. Correction to BCTReadBlockAin function description, - change PCIADC_AIN_AUTOSEL to PCIADC_AIN_MULTI_CHANNEL Add text relating to support for VB.NET header.
3.0	30/01/2007	DPR	Add text relating to PCI Relay card

OUTLINE DESCRIPTION 1
1.0 IMPORTANT INFORMATION 2
2.0 INSTALLATION 3
2.1 FILES INSTALLED 5
2.2 SAMPLE APPLICATIONS 6
2.2.1 PCI-PIO sample programs *7*
2.2.2 PCI-DIO sample programs *8*
2.2.3 PCI-ADC sample programs *9*
2.2.4 PCI-WDT sample programs *10*
2.2.5 PCI-RLY sample programs *10*
2.2.6 Miscellaneous sample programs *13*
3.0 USING THIS PRODUCT 14
3.1 USING THE DRIVERS FROM C OR C++ 14
3.1.1 The BLUECHIP.H header file *14*
3.1.2 Compiling and Linking *14*
3.2 USING THE DRIVERS FROM VISUAL BASIC 15
3.2.1 Visual Basic *15*
3.2.2 VB.Net *15*
3.3 IDENTIFYING BOARDS 15
3.4 APPLICATION DEVELOPMENT 16
3.4.1 Determine the board IDs and handles required. *16*
3.4.2 Initialise the handles and board ID structure *18*
3.4.3 Initialise the port directions in the 8255 .... *18*
3.4.4 Assign each allocated handle to a port *19*
3.4.5 Using the ports *19*
3.4.6 Closing all open handles *20*
3.5 ASYNCHRONOUS OPERATION 21
3.6 USING THE COUNTER TIMERS 21
3.7 PACING 22
3.8 USING DIGITAL INPUT AND OUTPUT 25
3.8.1 Ports A and B *25*
3.8.2 Split Port C *25*
3.8.3 Isolated Digital Output *25*
3.9 WATCHDOG TIMER 26
4.0 DRIVER API FUNCTIONS 27
4.1 Function Overview *27*
4.2 Function Descriptions *28*
4.2.1 Management Functions *28*
4.2.2 Digital Functions *34*
4.2.3 Analogue Functions *39*
4.2.4 Counter Functions *42*
4.2.5 Pacer Functions *43*
4.2.6 Watchdog timer functions *45*
5.0 EVENT LOG MESSAGES 47
5.1 ERROR CODES 48
5.2 BCTGETLASTERROR 56
A.0 LIBRARY DEFINED TYPES 57
A.1 Platform Independent Data Types *57*
A.2 Enumerated Types *57*
A.3 Structure Definitions *58*

OUTLINE DESCRIPTION

The Windows WDM driver for PCI Data Acquisition Cards ("the drivers") provide a simple programming interface to the supported range of PCI data acquisition cards for application programmers using either Windows 98, Windows 98SE, Windows Millennium Edition, Windows 2000 or Windows XP 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 that has the functionality that you require.

These drivers have been written to maintain backward compatibility with the API of the Windows NT drivers for the PCI range of data acquisition cards. As such applications written for Windows NT v4.0 should continue to execute correctly on any of the platforms supported by the WDM drivers. There is a possibility that in some cases in order to make these applications work correctly that they need to be recompiled under the new operating system.

Where new functionality has been added to correct problems or to simplify the programming model exposed by the API these new functions should be used when developing new applications.

1.0 IMPORTANT INFORMATION

These drivers remain the property of the supplier and are provided under the non-exclusive license agreement printed on the envelope in which they were delivered.

The most up to date information can be found in the READ.ME file on the installation CD.

If the CD that has been supplied is faulty then please contact your supplier for a replacement.

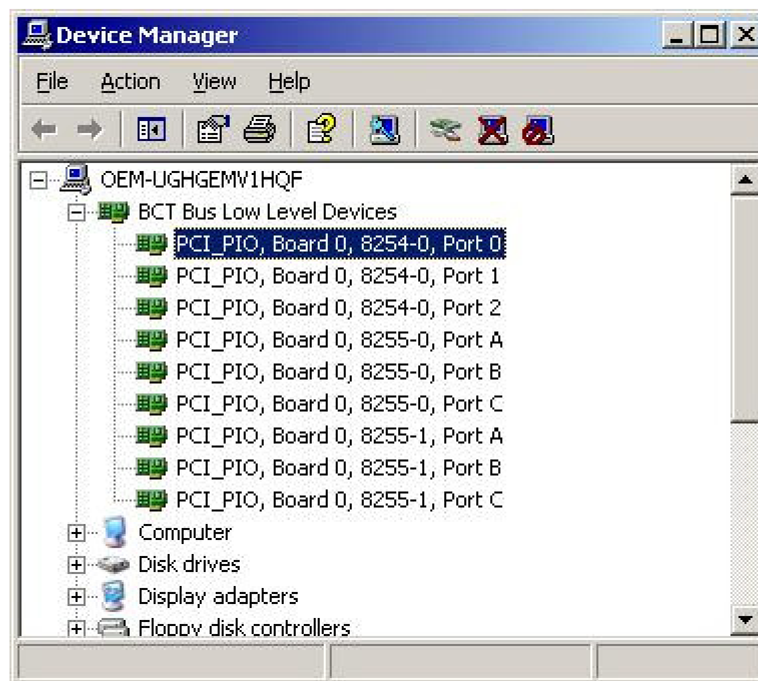
2.0 INSTALLATION

There are two scenarios under which the drivers could be installed. Firstly, if the PCI card was found in the system when the operating system was installed then the PCI data acquisition cards will be found listed as a “PCI Device” under “Other devices” in the device manager. Secondly, if the data acquisition cards have been added after the operating system was installed then card would be auto-detected and the operating system will prompt for a driver to be installed.

If the PCI data acquisition cards are shown in device manager as “PCI Device” under “Other devices” then right click on the device and choose “properties” from the menu. Click on the driver tab and click the “Update driver” button. Allow Windows to search for a driver automatically and ensure that the “CD-ROM” box is ticked. Windows will go and search for the .INF file and return with a location that it has found it from. Choose finish and allow Windows to copy the driver files that are required.

If the cards have been added after the operating system installation then on the reboot after installing the PCI data acquisition card, Windows will detect that new hardware was installed and will prompt for the driver CD. Insert the driver CD in an appropriate drive and specify that windows should “Install the software automatically”. The Windows driver installation procedure will copy the driver files to the appropriate directories on the Windows system.

During the installation of device drivers for data acquisition cards, Windows will detect further low level hardware for each card found, and will install drivers for each of these low level devices. These low level devices appear in device manager under “BCT Bus Low Level Devices”. An example of how device manager will find and display such devices for a PCI_PIO card can be seen below.



The specific low level devices found while installing PCI data acquisition cards is card specific. The table below shows what hardware will be detected for each type of PCI data acquisition card supported by the driver.

PCI Card	Low Level Devices Installed
PCI-PIO	8254-0, Port 0 8254-0, Port 1 8254-0, Port 2 8255-0, Port A 8255-0, Port B 8255-0, Port C 8255-1, Port A 8255-1, Port B 8255-1, Port C
PCI-DIO	8254-0, Port 0 8254-0, Port 1 8254-0, Port 2 ISODIG-0, Port 0 ISODIG-0, Port 1 ISODIG-0, Port 2
PCI-RLY	8254-0, Port 0 8254-0, Port 1 8254-0, Port 2 RLY-0, Port 0 RLY-0, Port 1 RLY-0, Port 2
PCI-ADC	8254-0, Port 0 8254-0, Port 1 8254-0, Port 2 AIN-0, Port 0 AOUT-0, port 0 AOUT-0, port 1 AOUT-0, port 2 AOUT-0, port 3
PCI-WDT	WDT-0, Port 0

When installing under Windows 2000 use is made of a co-installer that provides more meaningful names to the devices found under the device manager. Unfortunately at present Windows 98 and Millennium Edition make no use of this co-installer and as such the information shown in the device manager is limited.

NOTE: When altering the Blue Chip Technology card collection installed within a computer (adding cards, removing cards, changing PCI slots), it is recommended that prior to change all Blue Chip technology data acquisition cards are uninstalled in device manager. It is also recommended that after device manager has detected and installed new devices, a system restart be performed.

2.1 FILES INSTALLED

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

BCTENUM.SYS	This is the Blue Chip Technology WDM bus class driver for the PCI data acquisition cards
BCT_8254.SYS	The driver for 8254 counter timer devices found on the PCI data acquisition cards
BCT_8255.SYS	The driver for 8255 programmable IO devices on the PCI data acquisition cards.
BCT_AIN.SYS	The driver for analogue input functions on the PCI-ADC.
BCT_AOUT.SYS	The driver for analogue output functions on the PCI-ADC.
BCT_ISO.SYS	The driver for isolated digital IO functions on the PCI-DIO.
BCT_WDT.SYS	The driver for the PCI-WDT watchdog card.
BCT_RLY.SYS	The driver for the PCI-RLY relay card.
BCTCOINS.DLL	The co-installer library used during installation on Windows 2000
BCDLL32.DLL	The API dynamic link library used to access functions on the PCI data acquisition cards.
BCTENUM.INF BCTDEV.INF	The .INF files used by the operating system to install the appropriate drivers and library files.

NOTE: Even if an area of functionality is not present on the PCI data acquisition card installed in the system then all of the .SYS driver files are installed i.e. if only a PCI-PIO is installed in the system that contains only 8254 and 8255 devices the driver files BCT_AIN.SYS, BCT_AOUT.SYS, BCT_ISO.SYS, BCT_RLY.SYS and BCT_WDT.SYS will also be installed.

2.2 SAMPLE APPLICATIONS

The installation CD that contains the driver files also contains a number of sample applications that can be used to test out the functionality of the PCI data acquisition cards that have been installed in your system. These samples can also be used as the basis for functions that are implemented within your own application for calling the functions within the API DLL.

The sample programs are written using Visual Basic, VB.Net and Visual C++. The applications written using Visual C++ have been developed as console mode applications to remove the complexity of the code that is associated with the development of Windows applications using C or C++.

Each of the applications check the error codes returned from the API functions and display the results on the screen, however, in order to not over complicate the code some checking of data sizes is removed. The actions taken on finding an error returned from a DLL function have also been removed in an effort to make the sample applications appear simple to read.

It is imperative when writing your own applications that all error checking of parameters is performed and the correct action is taken as a result of error codes being returned from the API functions. This is especially important if the data acquisition cards and WDM drivers are being used in a control system where there is a potential danger to human life.

The sample programs perform the same functions in Visual Basic,VB.Net, and C++. The Visual Basic programs are controlled using buttons and text boxes on screen to get input and to display values. The console mode C applications use menus to get the information that they require and display the output on the screen using calls to printf().

2.2.1 PCI-PIO sample programs

There are two test programs for the PCI-PIO card. The first (PIOTEST) requires a loopback plug with the following connections in order to allow output values to be read back using the same card.

PIN 1 – PIN 9	PIN 5 – PIN 13	PIN 17 – PIN 25	PIN 21 – PIN 29
PIN 2 – PIN 10	PIN 6 – PIN 14	PIN 18 – PIN 26	PIN 22 – PIN 30
PIN 3 – PIN 11	PIN 7 – PIN 15	PIN 19 – PIN 27	PIN 23 – PIN 31
PIN 4 – PIN 12	PIN 8 – PIN 16	PIN 20 – PIN 28	PIN 24 – PIN 32

These pins connections relate to the following ports being connected.

PORT 1	PORT 2
PIO 8255 Device 0 Port A	PIO 8255 Device 0 Port B
PIO 8255 Device 0 Port C	PIO 8255 Device 1 Port A

Having set up the relevant handles and board ID structures for the PCI-PIO as described in section 3.4 it is possible to perform demand driven IO which involves writing an 8 bit value out of PIO 8255 Device 0 Port A and reading a value back from PIO 8255 Device 0 Port B. If the loopback connector is attached the value written out of PIO 8255 Device 0 Port A should match the value read back on port PIO 8255 Device 0 Port B.

Pacer driven IO uses a preloaded buffer to output values to PIO 8255 Device 1 Port A at 30ms intervals using the control of the 8254 pacer device on the PCI-PIO board. The program then sits in a loop reading back the value that appears at the input of PIO 8255 Device 0 Port C and displays it on screen. This should show an incrementing count from 0x00 to 0xFF, which then resets back to 0 and starts to count again.

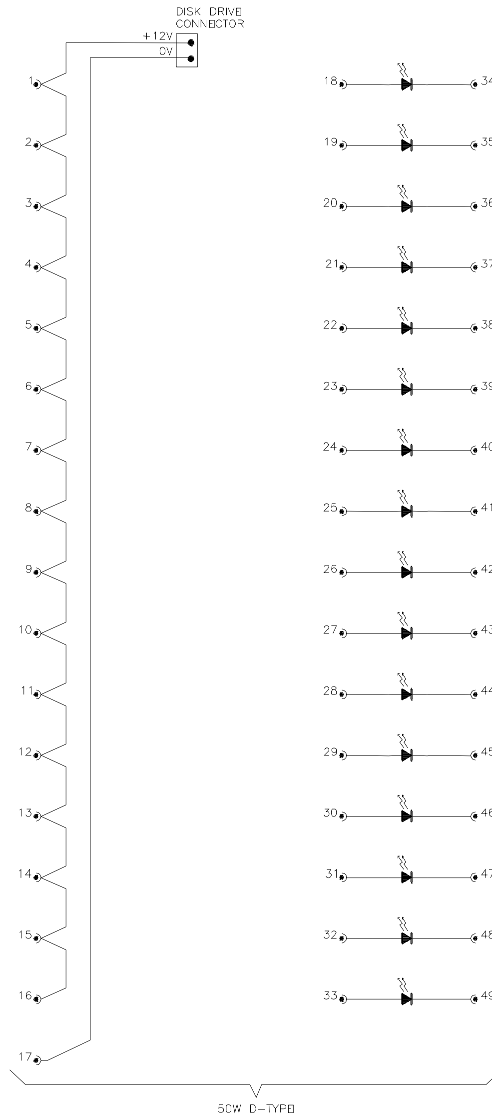
The second test program for the PCI-PIO (PIOCOUNT) uses the function BCTProgramCounter to count down the number of input pulses from the second external input to counter 0. In order to see this test working an input signal (square wave) from a signal generator needs to be input to pin 17 of the connector on the PCI-PIO and a ground connected from the signal generator to pin 50 of the PCI-PIO connector.

The on screen display will show the counter count down from 0xffff to 0 and when 0 is reached the counter will reset to 0xffff and start to count down again.

2.2.2 PCI-DIO sample programs

There are two test programs for the PCI-DIO card. The first (DIOTEST) performs demand driven 16 bit read and write operations using the functions BCTReadPort16 and BCTWritePort16.

This test requires a loopback plug with the following connections to allow output values to be read back using the same card. This loopback connector also includes LED's that can be used to monitor the values being driven on the output ports.



The second test program for the PCI-DIO (DIOCOUNT) uses the function BCTProgramCounter to count down the number of input pulses from the first external input to counter 0. In order to see this test working an input signal (square wave) from a signal generator needs to be input to pin 1 of the connector on the PCI-DIO and a ground connected from the signal generator to pin 18 of the PCI-DIO connector. In order for this test to work,

sufficient voltage needs to be supplied to the input pins to register the changes in the Opto-isolator.

The on screen display will show the counter count down from 0xffff to 0 and when 0 is reached the counter will reset to 0xffff and start to count down again.

2.2.3 PCI-ADC sample programs

There are two test programs for the PCI-ADC card. The first (ADCTEST) performs a test of each of the functional components of the PCI-ADC data acquisition cards, i.e. digital IO, analogue output, analogue input, auto calibration using direct IO calls and under pacer control.

In order to use the test program a test connector with the following connections is required:

PIN 10 – PIN 26	PIN 13 – PIN 29	PIN 16 – PIN 32	PIN 44 – PIN 48
PIN 11 – PIN 27	PIN 14 – PIN 30	PIN 17 – PIN 33	PIN 45 – PIN 49
PIN 12 – PIN 28	PIN 15 – PIN 31	PIN 43 – PIN 47	PIN 46 – PIN 50

Analogue Input	PIN 1 +ve	PIN 22 –ve
Analogue Output	PIN 24 +ve	PIN 8 –ve

The first part of the test performs demand driven IO activities on all three ports of the 8255 PIO device. Port A is looped back into port B and is used for 8 bit output from port A read back via port B. Port C has the upper 4 bits looped back to the lower four bits to show the use of a split port C. In this sample application the upper nibble is set for output and the lower nibble is used for the input from the looped back signals.

This sample program includes an analogue output test that drives a voltage out between –10V and +10V which can be monitored on pins 24 and 8 using a voltmeter. The analogue input test shows how the function BCTReadBlockAin test works and reads 500 samples over a 5 second period and displays the result on the screen. In the case of the Visual Basic version of ADCTEST this is done in the form of a graph to show a representation of the waveform, in the C example the first 16 values read back are displayed on the screen.

In the final option of the ADCTEST sample application the auto calibrate function of the PCI-ADC is shown to determine the mean zero and mean fsd of the PCI-ADC card that can then be used for scaling and correcting the values obtained when capturing analogue input data.

The second test program for the PCI-ADC (ADCCOUNT) uses the function BCTProgramCounter to count down the number of input pulses from the first external input to counter 1. In order to see this test working an input signal (square wave) from a signal generator needs to be input to pin 50 of the connector on the PCI-ADC and a ground connected from the signal generator to pin 9 of the PCI-ADC connector.

The on screen display will show the counter count down from 0xffff to 0 and when 0 is reached the counter will reset to 0xffff and start to count down again.

2.2.4 PCI-WDT sample programs

The sample application for the PCI-WDT performs two operations on the watchdog card. The first monitors the external inputs and displays the results on the screen. The second runs in a loop to keep the watchdog timeout refreshed. If either part of the sample application returns an error then the watchdog card is tripped. By default this is shown by illuminating the LED on the card.

2.2.5 PCI-RLY sample programs

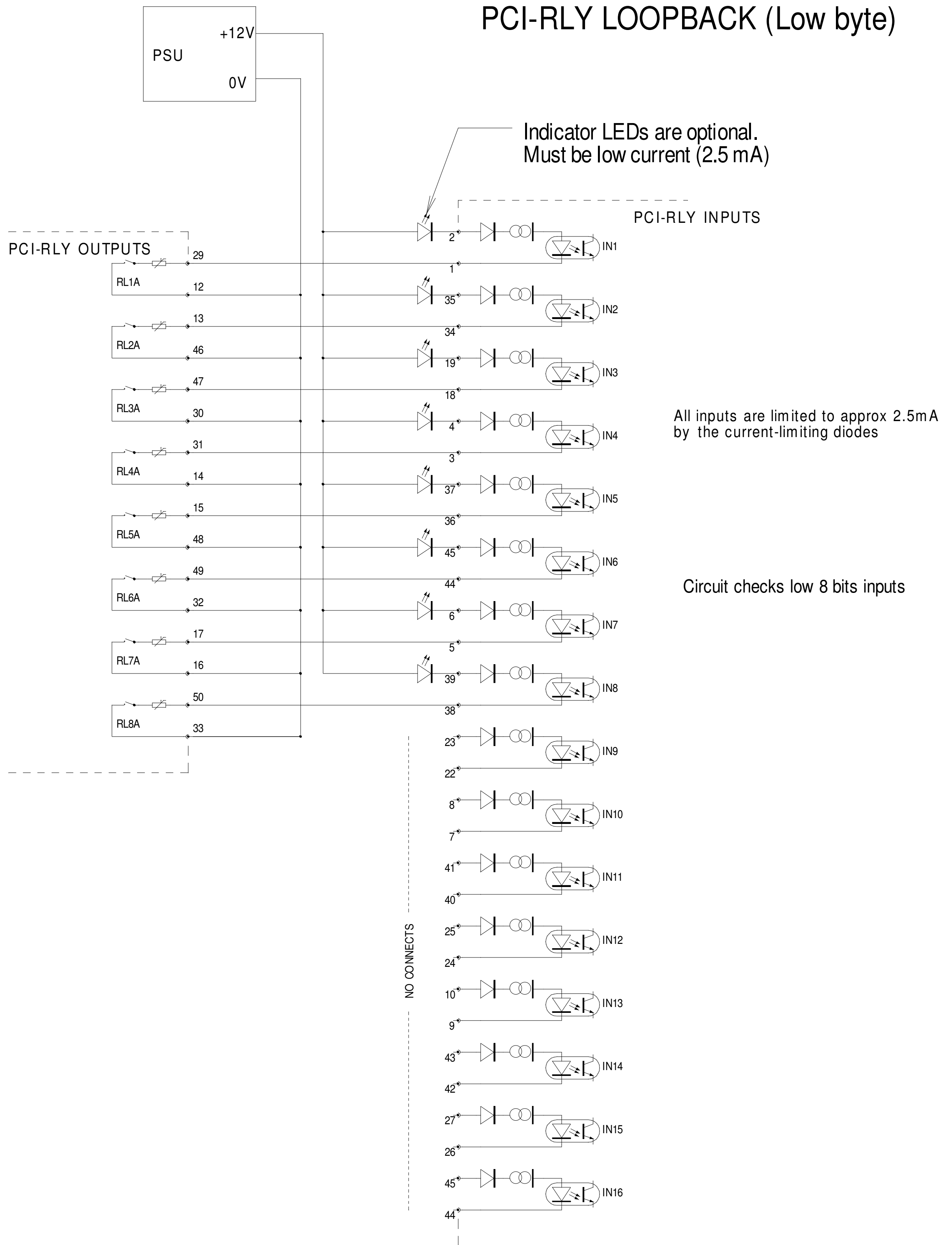
There are two test programs for the PCI-RLY card. The first (RLYTEST) requires a loop back plug to allow output values to be read back using the same card. As the relay card has an 8 bit output and 16 bit input two loopbacks are required to fully test the inputs of a card. In the first diagram shown on page 11 the outputs are linked to the lower 8 inputs (bits 0-7), and in the second diagram on page 12 the outputs are linked to the upper 8 inputs (bits 8-15).

The RLYTEST program allows data to be written and read manually by entering values in at the console and reading back the values. The test program asks the user which loop back connector is being used for the test and automatically adjusts the 16 bit input accordingly to ascertain if the value read matches what was written. The second test in RLYTEST demonstrates pacer driven (Interrupt driven) IO.

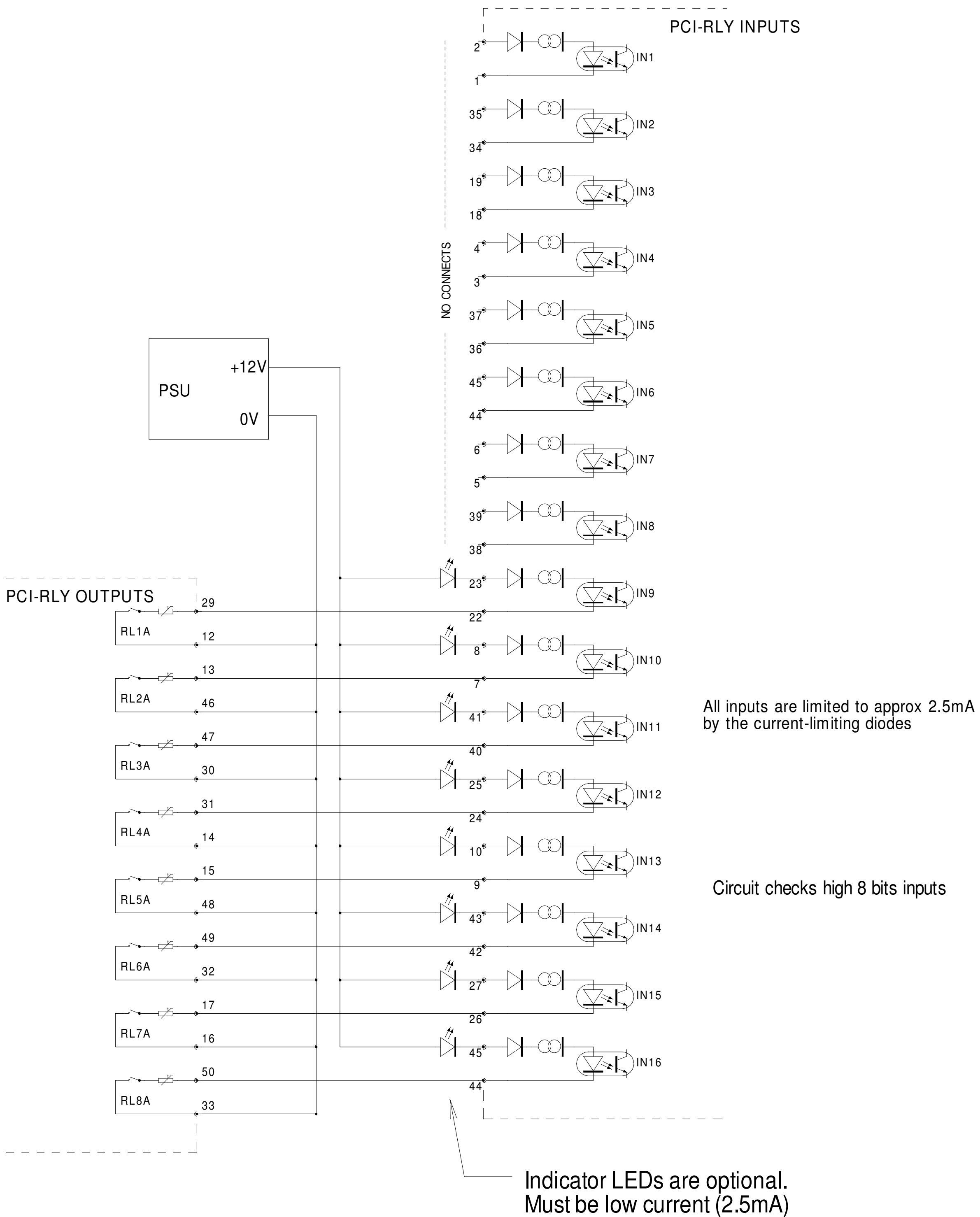
Pacer driven IO uses a preloaded buffer to output values to an output port at 30ms intervals using the control of the 8254 pacer device on the PCI-RLY board. The program then sits in a loop reading back the value that appears at the input of the card and displays it on screen. This should show values alternating between 0x55 and 0xAA.

The second test program for the PCI-RLY (RLYCOUNT) uses the function BCTProgramCounter to count down the number of input pulses from the first external input to counter 0. In order to see this test working an input signal (square wave) from a signal generator needs to be input to pin 2 of the connector on the PCI-RLY and a ground connected from the signal generator to pin 1 of the PCI-RLY connector. In order for this test to work correctly, sufficient voltage needs to be supplied to the input pins in order for the isolated input to register the changes in the signal level.

The on screen display will show the counter count down from 0xffff to 0 and when 0 is reached the counter will reset to 0xffff and start to count down again.


PCI-RLY LOOPBACK (Low byte)
PSU
+12V
0V
Indicator LEDs are optional.
Must be low current (2.5 mA)
PCI-RLY OUTPUTS
PCI-RLY INPUTS
RL1A
RL2A
RL3A
RL4A
RL5A
RL6A
RL7A
RL8A
IN1
IN2
IN3
IN4
IN5
IN6
IN7
IN8
IN9
IN10
IN11
IN12
IN13
IN14
IN15
IN16
NO CONNECTS
All inputs are limited to approx 2.5mA
by the current-limiting diodes
Circuit checks low 8 bits inputs

PCI-RLY LOOPBACK (High byte)



2.2.6 Miscellaneous sample programs

A number of the management functions are not specific to any given board and are used for determining board serial numbers. The sample application BOARDNUM uses both supported methods of determining the information relating to the boards in the system.

The first option takes a serial number and looks through the list of boards in the system and returns the board type and the index number of that board if a board of the serial number specified can be found in the system. If a board with the serial number specified can not be found in the system then an error is returned.

The second option that can be selected gives a display of all the boards in the system by calling BCTEnumerateBoards and displays the board type, its serial number and its index number within the driver.

The final test program that is contained on the diskette, in C form only, is TESTDLL. This is the program that was used extensively during the driver development process. This program contains calls to all of the functions contained within the driver, however it has specific requirements for loopback plugs in order for the tests to work. It is included here for reference for developers to confirm the order that functions are to be called and the parameter types that need to be passed.

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 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. This is done by BLUECHIP.H including BCTYPES.H, which will appear in the external dependencies list after the first compilation of the application.

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 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 that 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

3.2.1 Visual Basic

Accessing the library from a Visual BASIC program is straightforward. 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

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

and should always be used in you own code to improve readability.

3.2.2 VB.Net

In the case of a VB.Net program, the library functions are accessed via the header file "BLUECHIP.VB". This module may be included within a developer's program by selecting the, "Add Existing Item…" option under the 'Project' menu.

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 bar code 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). In order to determine this board number an additional function is provided, BCTFindBoardNo, that will search all the boards known to the driver for one matching the serial number passed as a parameter to the function . 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.

> In the Windows NT driver the function to determine the board number from the serial number was BCTFindSerialNo. This function required that the high and low order 16bit words were reversed prior to passing the serial number to the BCTFindSerialNo function. This function has been retained in the WDM drivers or backwards compatibility with applications written originally for the Windows NT driver but new applications should use the BCTFindBoardNo function that does not require the high and low order words to be reversed.

3.4 APPLICATION DEVELOPMENT

No matter which development language is being used the steps for developing an application are 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 a handle to each port being used
5. Run required functions
6. Close all handles and release the board ID structure prior to terminating

If we take as an example writing an application that uses 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;
```

In Visual Basic (and VB.Net) we define the same variables as:

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

' Define two handles one for input and one for  output
Dim nOutputHandle As BCT_HANDLE
Dim nInputHandle As BCT_HANDLE
```

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 the structures initialised. 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 dwStatus;
char *szErrTxt;

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

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

In Visual Basic (and VB.Net) we define the same variables and function calls as:

```
Dim dwStatus as Long
Dim szErrTxt as String
szErrTxt = Space(120)

' Get the board id structure completed by the driver
dwStatus = BCTGetBoardId(nPCIPIOBoardID, PCI_PIO, 0)
If dwStatus <> BCT_OK Then
       Call BCTErr2Txt(dwStatus, szErrTxt)
End If

' Get the handles initialised by the driver
dwStatus = BCTInitHandle(nOutputHandle)     ' Check dwStatus returned
dwStatus = BCTInitHandle(InputHandle)       ' Check dwStatus 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.

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

In Visual Basic (and VB.Net) we define the same function calls as:

```
‘ 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.

dwStatus = BCTInit8255Modes(nPCIPIOBoardID, 0, MODE_0, INPUT8, OUTPUT8, NOCARE)
‘ Check dwStatus returned
```

3.4.4 Assign each allocated handle to a port

Now that an 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
dwStatus = BCTOpen(&nInputHandle, &nPCIPIOBoardID, BCT_8255, 0, BCT_PORT_A);
// Check dwStatus returned

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

In Visual Basic (and VB.Net) we define the same function calls as:

```
‘ nInputHandle is for port A
dwStatus = BCTOpen(nInputHandle, nPCIPIOBoardID, BCT_8255, 0, BCT_PORT_A)
‘ Check dwStatus returned

‘ nOutputHandle is for port B
dwStatus = BCTOpen(nOutputHandle, nPCIPIOBoardID, BCT_8255, 0, BCT_PORT_B)
‘ Check dwStatus returned
```

3.4.5 Using the ports

If all the initialisation detailed above has completed successfully 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 *szErrTxt;

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

// To write a port…
```

```
dwStatus = BCTWritePort(&nOutputHandle, 0x55);
if (dwStatus == BCT_IO_PENDING)
{
       dwStatus = BCTWait(&nOutputHandle, INFINITE);
}
if ((dwStatus != BCT_OK) && (nStatus != BCT_IO_PENDING))
{
       BCTErr2Txt(dwStatus, szErrTxt);
       printf("Status: %s\n", szErrTxt);
       return(dwStatus);
}
```

In Visual Basic (and VB.Net) we define the same variables and function calls as:

```
' Variable to store result
Dim nVal As Byte
Dim szErrTxt As String
szErrTxt = Space(120)

' To read a port…
dwStatus = BCTReadPort(nInputHandle, nVal)
If dwStatus = BCT_IO_PENDING Then
       dwStatus = BCTWait(nInputHandle, INFINITE)
End If
If (dwStatus <> BCT_OK) And (dwStatus <> BCT_IO_PENDING) Then
       Call BCTErr2Txt(dwStatus, szErrTxt)
End If

// To write a port…
dwStatus = BCTWritePort(nOutputHandle, 0x55)
If dwStatus = BCT_IO_PENDING Then
       dwStatus = BCTWait(nOutputHandle, INFINITE)
End If
If (dwStatus <> BCT_OK) And (nStatus <> BCT_IO_PENDING) Then
       Call BCTErr2Txt(dwStatus, szErrTxt)
End If
```

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
dwStatus = BCTClose(&nInputHandle);                  // Check nStatus returned
dwStatus = BCTClose(&nOutputHandle);                 // Check nStatus returned
dwStatus = BCTReleaseBoardId(&nPCIPIOBoardId);       // Check nStatus returned
```

In Visual Basic (and VB.Net) we define the same function calls as:

```
' Close the handles
dwStatus = BCTClose(nInputHandle)                    ' Check nStatus returned
dwStatus = BCTClose(nOutputHandle)                   ' Check nStatus returned
dwStatus = 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 that 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-RLY, 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 counter 2 on the PCI-ADC board being used for analogue input (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 0xffff, counts down to 0 then wraps back to 0xffff, 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 0xffff 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-RLY, 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, and the maximum that is 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 in order to use the pacer to repeat the operations required.

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 from or writing to 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).

Note:
In VB.NET implementations, a dimensioned array, as used in BCT_BUFFER, cannot be declared from within the structure. Instead, the 'MarshalAs Attribute' class is used to handle each array data.
Furthermore, prior to using the buffer, vb.net requires the array to be initialised. One method is to use the Redim statement e.g. ReDim pBuffer.Buffer(16383) - as shown in the piotest example.

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 6th 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 with all bits 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.

3.8.3 Isolated Digital Output

In the Windows NT device drivers for the PCI data acquisition cards, the isolated digital IO on the PCI-DIO was configured using the function BCTInitIsoDigModes. In the WDM drivers this has been replaced by a new function called BCTInitPCIDio. This requires only a handle and a board index number, with this single handle used for 16bit input and output using BCTReadPort16 and BCTWritePort16.

To initialize the PCI_RLY a new function called BCTInitPCIRelay that populates a BoardID structure based on a board index number has been created. The boardID structure returned

from this function can be used to create a handle to either the input port or output port of a PCI-RLY.

Due to the nature of the design of the PCI-DIO and PCI-RLY card there is a 2.4ms latency between writing to the output buffer and the value appearing at the output pins. This is due to the data written to the PCI-DIO and PCI-RLY output buffer having to be serially driven to each output pin. Due to this limitation, if the outputs and inputs of the PCI-DIO or PCI-RLY are looped together in order to monitor the status of the outputs then the application should delay for at least 2.4ms between writing to the outputs and reading back the input value.

Further to this, if the isolated digital outputs on the PCI-DIO or PCI-RLY are set to be written using a buffer of data under pacer control then the rate at which the pacer is set to output data in this manner should be set to a value larger than 2.4ms. If the pacer is set to output new values faster than 2.4ms then the data appearing at the output pins may be invalid. It is the responsibility of the application programmer to ensure that the data that is being observed at the output pins of the PCI-DIO and PCI-RLY is correct at all times.

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 the complexity of the Windows device driver interface away, the drivers are supplied with an accompanying 32bit DLL which provides a number of library functions for interfacing with 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
- BCTEnumerateBoards
- BCTErr2Txt
- BCTFindBoardNo
- BCTFindSerialNo
- BCTGetBoardId
- BCTInitHandle
- BCTInitPacer
- BCTOpen
- BCTRelease
- BCTReleaseAPacer
- BCTReleaseBoardId
- BCTWait

Digital Functions:
- BCTInit8255Modes
- BCTInitIsoDigModes
- BCTInitPCIRelay
- BCTInitPciDio
- 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 however the use from Visual Basic is identical and all functions and their parameter types are defined for Visual Basic in the BLUECHIP.BAS module or BLUECHIP.VB supplied on the driver CD.

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

BCTEnumerateBoards

```
nError BCTEnumerateBoards(BCT_WORD nBoard,
                          BCT_BOARDTYPE *pBoardType,
                          BCT_WORD *pBoard,
                          BCT_DWORD *pSerial);
```

`nBoard` Index number of the board to query in the system (0 based)

`pBoardType` Address of a variable for the return of the unique identifier for the board type: PCI_PIO, PCI-RLY, 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.

`pSerial` Address of a variable for the return of the serial number of the board in question.

This function returns the board type, its index number and the serial number for a board in the system. In order to enumerate all the boards call the function repeatedly incrementing the value of nBoard each time until the call to BCTEnumerateBoards returns BCTERR_PCI_BOARD_NOT_FOUND.

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.

BCTFindBoardNo

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

`nSerial`	The unique serial number of the board to be identified.
`nBoardType`	Address of a variable for the return of the unique identifier for the board type: PCI_PIO, PCI_RLY, PCI_DIO, PCI_ADC, and 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.

BCTFindSerialNo

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

`nSerial`	The unique serial number of the board to be identified.
`nBoardType`	Address of a variable for the return of the unique identifier for the board type: PCI_PIO, PCI-RLY, PCI_DIO, PCI_ADC, and 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.

> In order for this function to correctly identify the board the high and low order 16bit words need to be reversed prior to calling the function i.e. for a board with a serial number of 22155 the nSerial parameter would need to be passed as 21550002.

> This function is only included in the WDM drivers for backward compatibility with the Windows NT drivers. All new application development should use the BCTFindBoardNo function that does not require the high and low order 16bit words to be swapped.

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_RLY, PCI_DIO, PCI_ADC, PCI_WDT

`nBoard` Which board of a particular type starting from 0

This function will validate the nBoardType and nBoard 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_RLY	Isolated Digital I/O on a relay card
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_PORT_ISO	16bits of I/O on a PCI-DIO
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	All 16 bits of I/O on a PCI-DIO
BCT_RLY_RLY_OUT	8 bit output port of PCI-RLY
BCT_RLY_OPTO_IN	16 bit input port of PCI-RLY

`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

ISO_DIG_LOW8, ISO_DIG_HIGH8 and ISO_DIG_ALL16 are retained only for backward compatibility with applications written for the Windows NT drivers that were initialised with the function BCTInitIsoDigModes. New applications should use the function BCTInitPciDio and call BCTOpen using BCT_PORT_ISO.

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 are 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 0xFFFFFFFE will wait for 4294967294ms (this equates to approx 50 days). INFINITE (0xFFFFFFFF) will not return until the I/O is complete.

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 need 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.

> This function is only included in the WDM drivers for backward compatibility with the Windows NT drivers. All new application development should use the BCTInitPciDio function and use a single handle for a single 16bit bi-directional port.

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 are 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

BCTInitPCIDio

```
nError BCTInitPCIDio(BCT_BOARD_ID *pBoardId,
                     BCT_WORD nDev);
```

`pBoardId` Identifier returned from BCTGetBoardId

`nDev` Specifies which of the PCI-DIO boards is to be used. If only 1 board is fitted this value will be zero.

This routine is used to initialise the PCI-DIO board prior to the board being used. This is equivalent to calling BCTInitIsoDigModes to set port C as bi-directional. As a result of calling this function one handle is required that can be used for both 16bit input and output operations and single bit writing.

BCTInitPCIRelay

```
nError BCTInitPCIRelay(BCT_BOARD_ID *pBoardId,
                     BCT_WORD nDev);
```

`pBoardId` Identifier returned from BCTGetBoardId

`nDev` Specifies which of the PCI-RLY boards is to be used. If only 1 board is fitted this value will be zero.

This routine is used to initialise the PCI-RLY board structure prior to the board being used. The board structure create by this function can be used to create handles for the input and output port of the card.

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 or PCI-RLY.

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_MULTI_CHANNEL 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_MULTI_CHANNEL

`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 input 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 0xffff 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

BCT_RLY All the functions are supported on the PCI-RLY

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` O 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
BCT_WD_RESET_BOARD
```

`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.

Message	Probable cause
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.

Error	Text equivalent	Description
0	BCT_OK	Function call was successful.
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.
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.
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.
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.
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.
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.

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_AIN,
      BCT_AOUT,
      BCT_WDT,
      BCT_RLY
} 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 i.e. 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_WORD            nPort;
      BCT_DWORD           LastError;
      clock_t             NextTime;
} 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];
      char                Clock2Name[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;
}
```