# AMPDIO DRIVERS

## DIGITAL/ANALOGUE INPUT/OUTPUT WINDOWS APPLICATION INTERFACE

This Instruction Manual is supplied with the AMPDIO drivers to provide the user with sufficient information to utilise the purchased product in a proper and efficient manner. The information contained has been reviewed and is believed to be accurate and reliable, however MEV Limited and **Amplicon Liveline Limited** accept no responsibility for any problems caused by errors or omissions. Specifications and instructions are subject to change without notice.



**Manual Part No 85989404 iss R3**

Prepared by Helen Elcock.

Revised by I.J. Abbott.

Approved for issue by J. Hayward, Product Manager.

**WINDOWS Analogue and Digital IO Driver Software**

**TABLE OF CONTENTS**

# 1 INTRODUCTION

## 1.1 Windows AMPDIO Drivers

The **Windows AMPDIO Drivers** are Windows software drivers that support a range of Amplicon digital and analogue data acquisition cards. They consist of kernel level drivers, a comprehensive application level Windows Dynamic link library (DLL) interface and example software. The software can handle most analogue and digital signal types.

The drivers support the following functional categories:

- Digital I/O
- Counter Timer Functions
- Analogue Input
- Analogue output

The drivers are fully compatible with Windows NT 4.0, Windows 2000, Windows XP, Windows Vista, Windows 7, Windows Server 2003, Windows Server 2008, Windows 95, Windows 98 and Windows ME. Version 5.00 onwards also includes support for the x64 editions of Windows XP, Windows Vista, Windows 7, Windows Server 2003 and Windows Server 2008.

Amplicon provide a comprehensive range of Personal Computer based data acquisition products that provide very high performance, affordable hardware with comprehensive software support.

When a large-scale system is required, multiple boards can be added without conflict. The capacity of the PC mounted hardware can be extended by external expansion panels to provide a comprehensive system with low cost per channel and maintained high performance.

## 1.2 Products supported

The drivers were initially developed to support the Amplicon 200 series of ISA digital IO/counter timer cards.

The drivers have since been expanded to support a number of ISA and PCI analogue data acquisition cards and PCI digital IO/counter timer cards.

The Amplicon 200 series cards functionality defines the underlying software architecture of the drivers. It is therefore important to have an understanding of the architecture of this family of cards in order to fully utilize the driver. Information on these cards can be found in the appropriate manuals on the Softman CD, e.g. PC215E manual.

There have been four major releases of the driver, Version 1.x, Version 2.x, Version 4.x and Version 5.x. Version 3.0 was never commercially released.

| | |
|---|---|
| Version 1.x | Supported Series 200 Digital IO / Counter Timer cards. |
| Version 2.x | Added support for additional Digital IO / Relay cards as well as rudimentary support for basic analogue cards. |
| Version 3.x | Added support for transferring large buffers of information to and from the cards under interrupt control. (Never commercially released) |
| Version 4.x | Added support for multifunction Analogue input / output cards and defines a new analogue interface standard. |
| Version 5.x | Added support for Windows Vista and x64 editions of Windows. |

### 1.2.1 PC200 Series

The 200 Series cards are a range of ISA Bus PC Digital IO / Counter Timer expansion cards. The 200 Series digital input/output products may be configured in a variety of ways to provide flexible, expansible systems.

Several digital input/output boards with timer/counter facilities are offered. These boards are complemented by four external panels for signal conditioning and user connection through individual terminals. Support and demonstration software for all variants is offered.

A full, itemized list of hardware products is shown below. To complete the family, a common software package supports all digital I/O boards and the expansion panels.

| Product Number | Product Type | Brief Description |
|---|---|---|
| PC212E[1] | ISA Counter/timer, Digital I/O board | 12 counters, clock/gate source, 24-line digital I/O |
| PC214E[1] | ISA Counter/timer, Digital I/O board | 3 counters, 48-line digital I/O |
| PC215E[1] | ISA Counter/timer, Digital I/O board | 6 counters, clock/gate source, 48-line digital I/O |
| PC218E[1] | ISA Counter/timer board | 18 counters, clock/gate source |
| PC272E[1] | ISA Digital I/O board | 72-line digital I/O |
| PC36AT[2] | ISA Digital I/O Board | 24-Line Digital I/O Board |
| PC263[2] | ISA 16-Line Relay Board | |
| PCI215[7] | PCI Counter/timer, Digital I/O board | 6 counters, clock/gate source, 48-line digital I/O |
| PCI236[7] | PCI Digital I/O Board | 24-Line Digital I/O Board |
| PCI263[8] | PCI 16-Line Relay Board | |
| PCI272[7] | PCI Digital I/O board | 72-line digital I/O |
| | | |
| EX233 | Termination/distribution panel | 78 Terminals, 3 x 37 way distribution connectors |
| EX213 | Output panel | 24 relay or high level logic source drivers |
| EX230 | Input panel | 24 isolated or non-isolated, high or low level inputs |
| EX221 | Input/output panel | 16 inputs, 8 outputs |
| | | |
| 9096 6349 | 78 way Screened Cable 1m | I/O board to EX233 Termination/distribution panel |
| 9095 6109 | 37 way Screened Cable 1m | PC36AT, PCI236 or EX233 to I/O panel |
| 9194 5753 | 37 way Screened Connector Kit | |
| 9089 1950 | 37 way screw terminal assy | |
| 9194 5953 | 78 way Screened Connector Kit | |

### 1.2.2 Analogue Input / Output Cards

The driver supports a range of analogue input and output boards, all of which have 82C53 or 82C54 compatible counter timers. External panels for signal conditioning and user connection through individual terminals are available. Support and demonstration software for all variants is offered.

#### 1.2.2.1 Analogue Output Cards

| Product Number | Product Type | Brief Description |
|---|---|---|
| PC24E[3] | ISA DAC Counter Timer card | 3 counters, 4 Channel +/- 10V 12 bit DACs |
| PC25E[3] | ISA DAC Counter Timer card | 3 counters, 4 Channel 4-20 mA 12 bit DACs |
| PCI224[6] | PCI DAC Counter Timer card | 3 counters, 16 multiplexed 12-bit DACs with FIFO |
| PCI234[6] | PCI DAC Counter Timer card | 3 counters, 4 multiplexed 16-bit DACs with FIFO |

#### 1.2.2.2 Analogue Input Cards

| Product Number | Product Type | Brief Description |
|---|---|---|
| PC26AT[4] | ISA ADC Counter Timer card | 3 counters, 16 multiplexed ADC channels |
| PC27E[3] | ISA ADC Counter Timer card | 3 counters, 16 multiplexed ADC channels |
| PCI260[5] | PCI ADC Counter timer card | 3 counters, clock/gate source, 16 multiplexed ADC channels with FIFO |
| PCI260+[9] | PCI ADC Counter timer card | 3 counters, clock/gate source, 16 multiplexed ADC channels with FIFO |

The PCI260+ is an enhanced version of the PCI260, with 16-bit ADC, start acquisition control and extra timer/counter gate controls. It is backwards compatible with the old card except that the maximum ADC sample rate has been reduced from 312500 to 250000 samples per second.

#### 1.2.2.3 Multi-function Analogue Cards

| Product Number | Product Type | Brief Description |
|---|---|---|
| PC30AT[4] | ISA AIO Counter Timer card | 3 counters, 24-line digital I/O, 16 multiplexed ADC channels, 2 DACs |
| PC230[5] | PCI AIO Counter Timer card | 3 counters, clock/gate source, 24-line digital I/O, 16 multiplexed ADC channels with FIFO, 2 DACs |
| PC230+[9, 10] | PCI AIO Counter Timer card | 3 counters, clock/gate source, 24-line digital I/O, 16 multiplexed ADC channels with FIFO, 2 DACs |

[1] Supported since Version 1.00 of the driver.
[2] Supported since Version 2.01 of the driver.
3 Supported since Version 2.01 of the driver, interface changes in Version 4.00
4 Supported in Version 4.00 of the driver
5 Supported in Version 4.10 of the driver
6 Supported in Version 4.20 of the driver
7 Supported in Version 4.30 of the driver
8 Supported in Version 4.31 of the driver
9 Supported in Version 4.42 of the driver
10 PCI230+ hardware version 2 has a DAC FIFO supported in Version 4.44 of the driver

The PCI230+ is an enhanced version of the PCI230, with 16-bit ADC, start acquisition control and extra timer/counter gate controls. It is backwards compatible with the old card except that the maximum ADC sample rate has been reduced from 312500 to 250000 samples per second.

## 1.3 Features of the Software

### 1.3.1 Overview

The software consists of low-level Windows kernel drivers, a Windows Dynamic Link Library (DLL) and a suite of example software.

The Windows Dynamic Link Library (DIO_TC.DLL) contains over 50 functions and provides a common Applications Program Interface (API) to the supported boards. The library functions allow the boards to be easily applied to many different applications, and provide an easy way of accessing the board's features. The DLL can be used by any language that supports the Windows '_stdcall' calling convention. The programming interface for this DLL is detailed in chapter 5, with any later updates detailed in the README.TXT file installed in the DIO_CODE subdirectory. AMPDIO v5.00 and later includes versions of DIO_TC.DLL compiled for IA-32 and x64 processor architectures. On 'x64' editions of Windows, both versions of the DLL are installed to support 32-bit programs and 64-bit (x64) programs. Earlier versions of AMPDIO only support the IA-32 processor architecture.

The low level kernel drivers provide a common low level interface to supported cards in Windows 95, Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003 and Windows Vista. AMPDIO v5.00 and later includes drivers for the 'x64' editions of Windows XP, Windows Vista, Windows 7, Windows Server 2003 and Windows Server 2008 (in addition to drivers for the regular IA-32 editions of Windows). A complete description of this interface is given in the additional document AMPIOCTL.RTF that is installed in the DIO_CODE directory.

Example programs written in Microsoft Visual Basic, Microsoft Visual Basic .NET, Microsoft Visual C# .NET, Borland Delphi, Agilent VEE (formerly HP VEE) and Microsoft C are also provided. Information on how to use the interface in Borland C Builder is given on MEV's web site, www.mev.co.uk.

Add-on drivers for National Instruments LabVIEW are available for some of the supported boards. See the Amplicon download area www.amplicon.co.uk/softman.cfm for a list of supported boards.

### 1.3.2 Typical Applications

The cards supported by these drivers are typically used in the following applications.

- TTL compatible digital input/output
- Relay output with isolated contacts, high level ground referenced source drivers (any combination)
- Isolated high or low level digital input, ground referenced high or low level digital input (any combination)
- Elapsed time, period, frequency measurement
- Differential, ratiometric count
- Monostable and astable generation
- Frequency division, frequency multiplication, digitally controlled oscillator
- Voltage controlled oscillator
- Temperature measurement
- 4–20 mA / analogue sensor simulation
- Low frequency Signal Generator
- Low frequency PC oscilloscope

## 1.4 Windows Installation Program

The software is installed onto the user's hard disk by a Windows installation program. See section 2 of this manual for information on getting started.

## 1.5 Technical Support

Should this product appear defective, please check the information in this manual and any 'Help' or 'READ.ME' files appropriate to the program in use to ensure that the product is being correctly applied.

If a problem persists, please request Technical Support on one of the following numbers:


| | | |
|---|---|---|
| Telephone: | UK | 0844 324 0617 |
| | Calls cost 5p per min from a BT landline. Calls from other services may vary. | |
| Fax: | UK | 01273 570 215 |
| E-mail | support@amplicon.com | |
| Internet | www.amplicon.com | |

# 2 GETTING STARTED

## 2.1 General Information

The installed software package contains a number of ready-to-run Windows 32-bit executable programs. These programs allow the user to perform I/O operations on the target card immediately after installing the board and software onto a PC.

## 2.2 Installing the Software

### 2.2.1 Software Installation from CD-ROM

To install the AMPDIO driver software from CD-ROM you will first need to decompress it on to your hard disk.

From the start/run menu, browse the Amplicon “SOFTMAN” CD. From the software directory run AMPDIO.EXE (or ADIO32.EXE on some versions) and follow the on screen instructions.

For AMPDIO 4.46 and earlier, once the software has been decompressed onto your hard drive, if the setup program does not run automatically, run the installation program C:\AMPLICON\AMPDIO\SETUP.EXE (C:\AMPLICON\ADIO32\SETUP.EXE on some versions). This is not necessary for AMPDIO 5.00 and later.

The source code and examples will have been decompressed into sub directories in the target directory. For AMPDIO 5.00 and later, installation of the source code and examples is optional.

## 2.3 Installing ADIO cards in the system

Once the AMPDIO software has been installed, it is necessary to install the drivers for your card onto the operating system. The way this is done varies between the various Windows operating systems.

Note: PCI ADIO cards are ‘plug and play’ and the operating system will try to install the drivers automatically. It may need some manual assistance to find the driver files.

For AMPDIO 5.00 and later, the AMPDIO.EXE program can pre-install the driver files as a Windows driver package for Windows 2000 onwards. Installing the Windows driver package for Windows 2000 onwards updates the drivers for any Amplicon ADIO cards already installed in the system. It also lets the system know about Amplicon ADIO cards before the first card is installed. For Windows 2000 onwards, the Windows driver package will be installed when the ‘AMPDIO Device Driver’ component is selected when running the AMPDIO.EXE installation program. On later versions of Windows it is also necessary to press the ‘Install’ button on the ‘Windows Security’ dialog that pops up during installation.

### 2.3.1 Installing a card in Windows 7

Installing a card in Windows 7 requires AMPDIO version 5.00 or later, but version 5.04 or later is recommended as it installs a hardware installation guide specific to Windows 7.

It is recommended to install the driver as a Windows driver package as described in section 2.3.

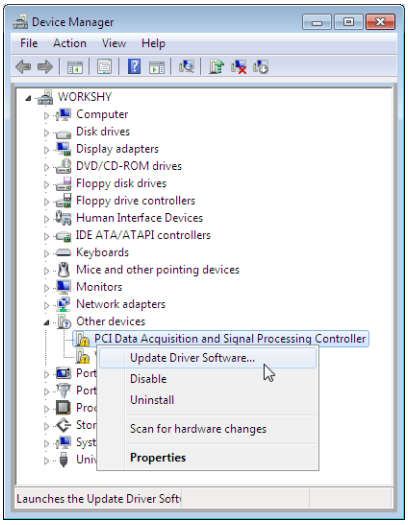
#### 2.3.1.1 PCI Card

For a PCI card, Windows will detect the new hardware automatically and attempt to install the drivers without user interraction. Unless the AMPDIO drivers have been previously installed (or pre-installed as a Windows driver package), this is likely to install the device as a non-working device of type 'PCI Data Acquisition and Signal Processing Controller' in the 'Other devices' section of Windows Device Manager.

The recommended way to install the driver is as a Windows driver package as described in section 2.3. An alternative method is to update the initially installed, non-working device in Windows Device Manager, using the unpacked driver files. These can be found on the SOFTMAN CD-ROM and may also be found in the directory on the hard disk where the AMPDIO software was installed (e.g. C:\AMPLICON\AMPDIO) if the 'AMPDIO Device Driver' component was selected in the AMPDIO Setup Wizard.

To update the driver for the initially installed, non-working PCI card, do the following:

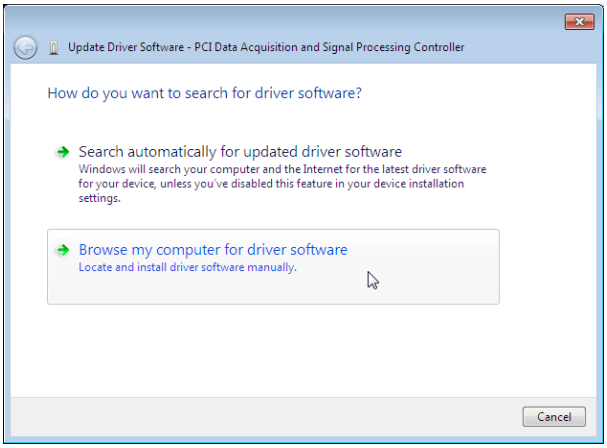
1. Go to the Control Panel by clicking START > Control Panel. Find and open the Windows Device Manager, for example by selecting 'View by: Category', clicking on 'Hardware and Sound', then on 'Device Manager'.




2. In Device Manager, find the initially installed, non-working PCI card in the 'Other devices' category. It will probably be listed as a 'PCI Data Acquisition and Signal Processing Controller', but might be listed as an unknown device.

3. Assuming the device is an Amplicon PCI ADIO card, right-click on the device and select the 'Update Driver Software...' option.

4. On the page that says 'How do you want to search for driver software?', click the option 'Browse my computer for driver software'.




5. On the page that says 'Browse for driver software on your computer', click the 'Browse...' button and browse to the location of the AMPDIO driver files on the hard disk or SOFTMAN CD (look for the file AMPDIO.INF). Then click the 'Next' button.

6. On the 'Windows Security' dialog with the question 'Would you like to install this device software?' click the 'Install' button.

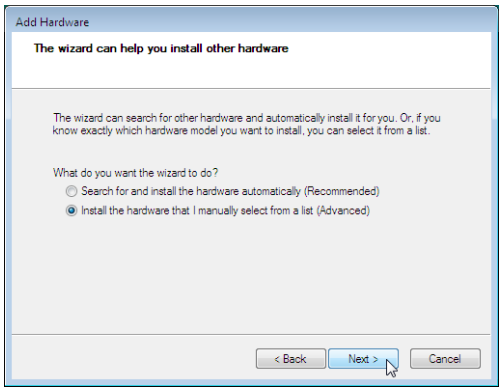
7. Windows should install the device driver software and show the message 'The software for this device has been successfully installed'. Click the 'Close' button.

#### 2.3.1.2 ISA Card

It is recommended to install the Windows driver package first, as described in section 2.3.

To install an ISA card in Windows 7, do the following:

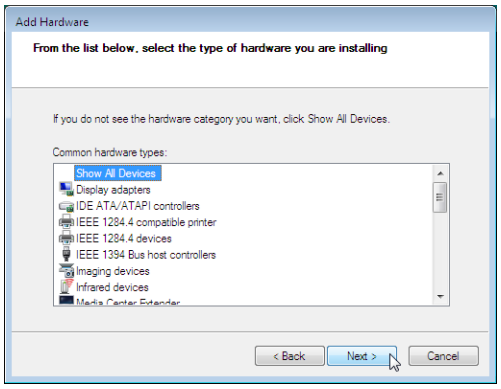
1. Run the ‘Add Hardware Wizard’ by clicking the START button, typing ‘hdwwiz.exe’ into the search box and pressing the ‘Enter’ key.




2. On the ‘Welcome to the Add Hardware Wizard’ page, click ‘Next’ to continue.

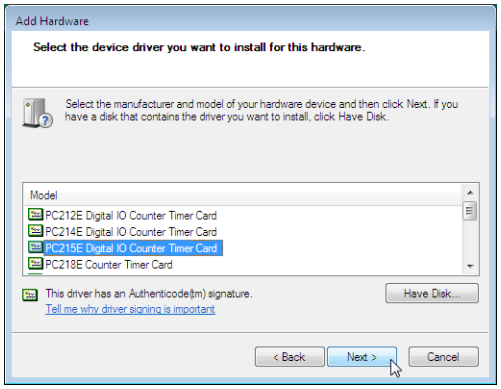
3. On the ‘Add Hardware page’, to the question ‘What do you want the wizard to do?’, select the option ‘Install the hardware that I manually select from a list (Advanced)’, then click ‘Next’.

4. Windows shows a list of hardware categories. Select the category ‘Amplicon Analogue/Digital IO Counter Timer Cards’ if it exists, otherwise select the ‘Show All Devices’ category at the top of the list, then click ‘Next’.




5. Click ‘Have Disk...’, then on the pop-up dialog click ‘Browse...’ to browse to the location on the hard disk where the AMPDIO software is installed (e.g. C:\AMPLICON\AMPDIO) (alternatively, browse to the root directory of the SOFTMAN CD-ROM), then click ‘OK’.

6. Select the card type you have installed from the list, then click ‘Next’.



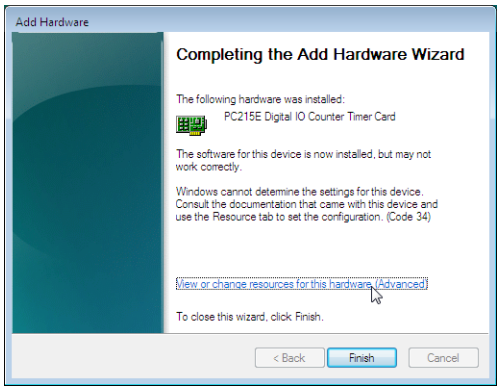

7. On the ‘Windows Security’ dialog with the question ‘Would you like to install this device software?’ click the ‘Install’ button.

8. Windows should install the driver and reach the ‘Completing the Add Hardware Wizard’ page. On this page, click the link labelled ‘View or change resources for this hardware (Advanced)’ to open the ‘Add Hardware Properties’ page.

9. On the ‘Add Hardware Properties’ page, click the button labelled ‘Set Configuration Manually’.

10. On the ‘Add Hardware Properties’ page, untick the ‘Use automatic settings’ option.

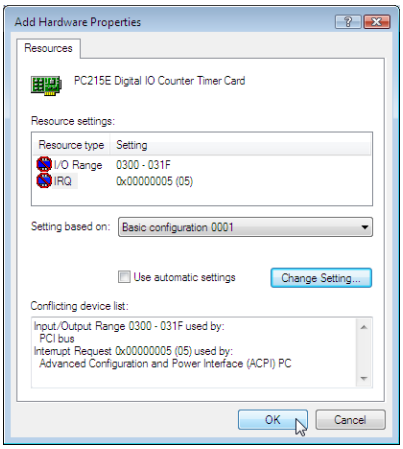
11. If the settings match the base address and IRQ set on the card’s DIP switches and jumpers, then click ‘OK’, then ‘Finish’, then reboot the computer. If the settings do not match then carry on with the following:




12. In the ‘Settings based on’ drop-down list, select ‘Basic Configuration 0001’ (or select ‘Basic Configuration 0002’ if the card’s IRQ jumper has been removed).

13. In the ‘Resource settings’ list, select the resource you wish to change (I/O range or IRQ), click the ‘Change Setting...’ button, type in the correct value (the up and down buttons will not work properly) and click ‘OK’. When entering the I/O range as a single number (e.g. 300), Windows will pop up a dialog box offering to correct the setting to a range (e.g. 0300-031F). Click ‘Yes’ to correct the I/O range if necessary. For the IRQ setting, type in the IRQ as a single number,

(e.g. 5 or 10). The IRQ will have to be reserved for use by legacy devices in the PC BIOS settings. Note that the entered resource settings will conflict with the PCI bus initially. Windows will pop-up a 'Conflict Warning' dialog box. On this dialog, click 'Yes' to keep the new resource setting. Repeat for the other resources you wish to change.




14. When you are happy with the new resource settings, click 'OK', then 'Finish'.

15. Once the device has been installed (with resource conflicts), reboot the computer.

16. Once the computer has rebooted, go to the Control Panel by clicking START > Control Panel. Find and open the Windows Device Manager, for example by selecting 'View by: Category', clicking on 'Hardware and Sound', then on 'Device Manager'. Make sure the new device is correctly installed. It should appear under 'Amplicon Analogue/Digital Counter Timer Cards' without an exclamation mark, indicating that the card is working.

### 2.3.2 Installing a card in Windows Vista

Installing a card in Windows Vista requires AMPDIO version 5.00 or later.

#### 2.3.2.1 PCI Card

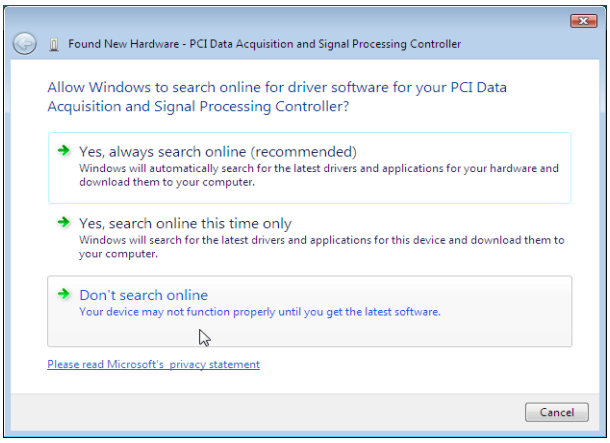
For a PCI card, Windows will detect the new hardware automatically and attempt to install the drivers. The driver files can be installed from the AMPDIO software installation directory or from the SOFTMAN CD-ROM.

To install the PCI card automatically on system start-up, do the following:

1. When Windows detects the new hardware, it opens the 'Found New Hardware' page. Click the 'Locate and install driver software (recommended)' option.

2. If the 'User Account Control'dialog pops up, click 'Continue'.

3. To the question 'Allow Windows to search online for driver software for your ... ?', click the 'Don't search online' option.

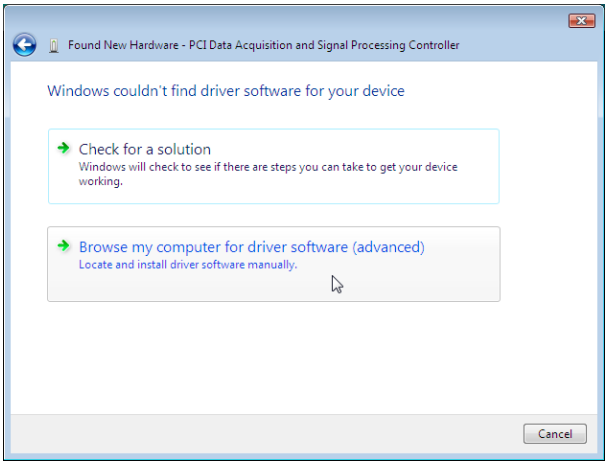



4. The Found New Hardware page shows a picture of a CD-ROM drive and instructs you to insert the disc that came with your hardware. If installing the driver from the CD-ROM, insert the SOFTMAN CD-ROM; Windows should install the driver automatically. If installing from the AMPDIO software installation directory on the hard disk, click the 'I don't have the disc. Show me other options' option.

5. On the page that says 'Windows couldn't find driver software for your device', click the 'Browse my computer for driver software (advanced)' option.

6. On the page that says 'Browse for driver software in this location', click the 'Browse...' button and browse to the AMPDIO software installation directory on the hard disk. Then click the 'Next' button.

7. On the 'Windows Security' dialog, to the question 'Would you like to install this device software?' click the 'Install' button.




8. Windows should install the device driver software and show the message 'The software for this device has been successfully installed'. Click the 'Close' button.

#### 2.3.2.2 ISA Card

To install an ISA ADIO card in Windows Vista, do the following:

1. Go to the Control Panel by clicking START > Control Panel.

2. Switch the control panel to 'Classic View' then double click the 'Add Hardware' icon.

3. If the 'User Account Control' dialog pops up, click 'Continue'.

4. On the 'Welcome to the Add Hardware Wizard' page, click 'Next' to continue.

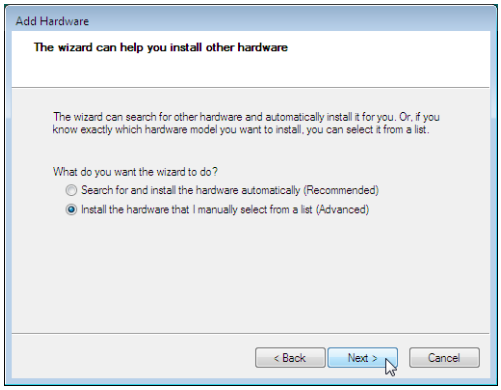
5. On the 'Add Hardware' page, to the question 'What do you want the wizard to do?' select the 'Install the hardware that I manually select from a list (Advanced)' option, then click 'Next'.

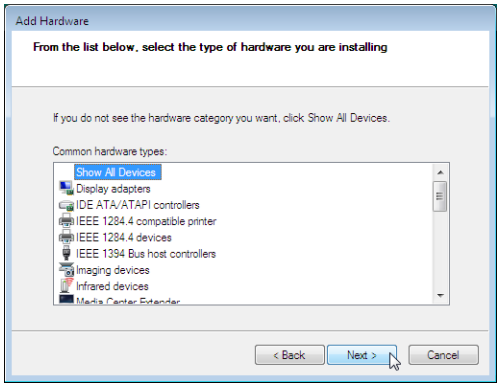
6. Windows shows a list of hardware categories. Select the 'Amplicon Analogue/Digital IO Counter Timer Cards' category if it exists, otherwise select the 'Show All Devices' category at the top of the list. Then click 'Next'.

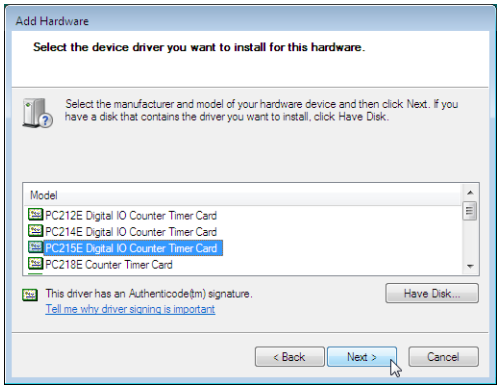
7. Click 'Have Disk...' then on the pop-up dialog click 'Browse...' to browse to the AMPDIO software installation location on the hard disk, or the root directory of the SOFTMAN CD-ROM), then click 'OK'.

8. Select the card type you have installed from the list, then click 'Next'.

9. On the page labelled 'The wizard is ready to



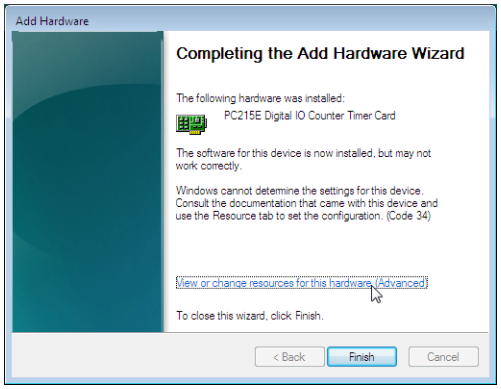



install your hardware' click 'Next'.

10. On the 'Windows Security' dialog, to the question 'Would you like to install this device software?' click the 'Install' button.

11. Windows should install the driver and reach the 'Completing the Add Hardware Wizard' page. On this page, click the link labelled 'View or change resources for this hardware (Advanced)' to open the 'Add Hardware Properties' page.



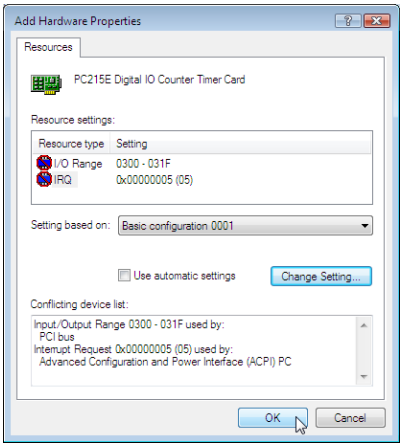

12. On the 'Add Hardware Properties' page, click the button labelled 'Set Configuration Manually'.

13. On the 'Add Hardware Properties' page, untick the 'Use automatic settings' option.

14. If the displayed resources match the base address and IRQ set on the card's DIP switches and jumpers, then click 'OK', then 'Finish', then tell Windows to reboot the computer via the usual means (e.g. via the START menu). If the displayed resources do not match the card then carry on with the following:

15. In the 'Settings based on' drop-down list, select 'Basic Configuration 0001' (or select 'Basic Configuration 0002' if the card's IRQ jumper has been removed).

16. In the 'Resource settings' list, select the resource you wish to change (I/O range or IRQ), click the 'Change Setting...' button, type in the correct value (the up and down buttons will not work properly) and click 'OK'. When entering the I/O range as a single number (e.g. 300), Windows will pop up a dialog box offering to correct the setting to a range (e.g. 0300-031F). Click 'Yes' to correct the I/O range if necessary. For the IRQ setting, type in the IRQ as a single number, (e.g. 5 or 10). The IRQ will have to be reserved for use by legacy devices in the PC BIOS settings. Note that the entered resource settings will conflict with the PCI bus initially. Windows will pop-up a 'Conflict Warning' dialog. On this dialog, click 'Yes' to keep the new resource setting. Repeat for the other resources you wish to change.




17. When you are happy with the new resource settings, click 'OK', then 'Finish'.

18. Once the device has been installed (with resource conflicts), reboot the computer via the usual means (e.g. via the START menu).

19. Once the computer has rebooted, go to the Control Panel and double click on the 'Device Manager' icon to make sure the new device is correctly installed. It should appear under 'Amplicon Analogue/Digital Counter Timer Cards' without an exclamation mark indicating that the card is working.

### 2.3.3 Installing a card in Windows XP

For versions of the AMPDIO software prior to 4.32, please follow the instructions for installing a card in Windows NT 4.0 (see section 2.3.5). For versions 4.30 and 4.31, the supplied AMPDIOV4.INF file will allow the supported PCI cards to appear under Device Manager, but these

are just dummy entries. For versions prior to 4.30 the supported PCI cards will appear as unknown devices under Device Manager.

For AMPDIO software versions 4.32 and later a 'Plug and Play' Windows 2000 / Windows XP driver is used. This section describes how to install a card to use this Plug and Play driver under Windows XP.

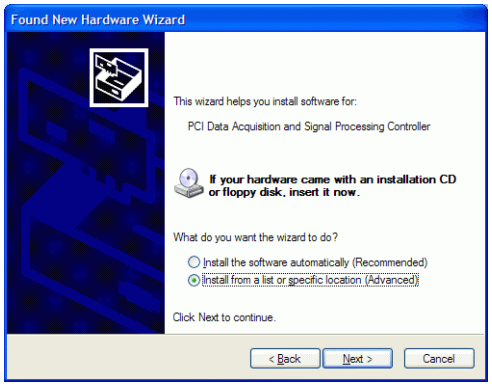
#### 2.3.3.1 PCI Card

For a PCI card, Windows will detect the new hardware automatically and attempt to install the drivers. The driver files can be installed from the AMPDIO software installation directory or from the SOFTMAN CD-ROM.

To install the PCI card automatically on system start-up, do the following:

1. If installing from CD-ROM rather than from the AMPDIO software installation directory, ensure the Amplicon SOFTMAN CD-ROM is in the CD-ROM drive.

2. When Windows detects the new hardware it opens the 'Welcome to the Found New Hardware Wizard' page. To the question 'Can Windows connect to Windows Update to search for software?' select the option 'No, not this time'. Press 'Next'.




3. If installing from the CD-ROM, select the 'Install the software automatically (Recommended)' option. If installing from the AMPDIO software installation directory on the hard disk, select the 'Install from a list or specific location (Advanced)' option. Press 'Next'.

4. If installing from the AMPDIO software installation directory, select the 'Search for the best driver in these locations' option, deselect the 'Search removable media (Floppy, CD-ROM...)' option, select the 'Include this location in the search' option, press the 'Browse' button and browse to the AMPDIO software directory. Then press 'Next'.

5. If a 'Security Alert' dialog appears, press 'Yes' to allow Windows to install the driver.

6. Windows will install the driver and reach the 'Completing the Found New Hardware Wizard' page.

7. On the 'Completing the Found New Hardware Wizard' page, press 'Finish'.

#### 2.3.3.2 ISA Card

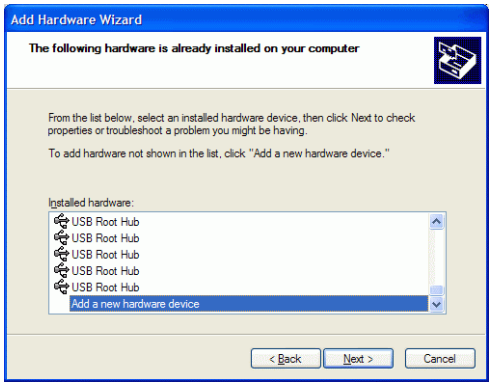
To install an ISA ADIO card in Windows XP (with the Plug and Play driver) do the following:

1. Go to the Control Panel by pressing START > Control Panel.

2. If the Control Panel is showing the Category View, switch to the Classic View. In the Classic View, double click on the 'Add Hardware' icon.

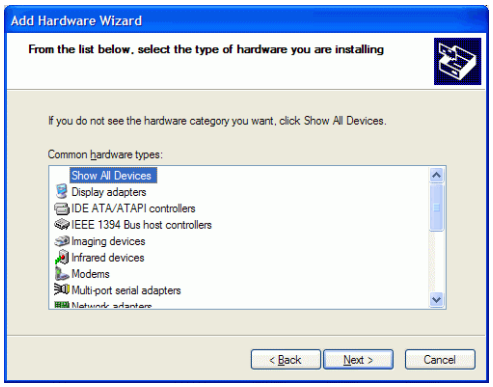
3. On the 'Add Hardware Wizard' page, press 'Next'.

4. Windows will search for Plug and Play hardware. Assuming it finds none, Windows will ask 'Is the hardware connected?'. Select the option 'Yes, I have already connected the hardware' and press 'Next'.




5. Windows will show a list of hardware already installed on your computer. Scroll to the end of the list and select the bottom entry, 'Add a new hardware device'. Press 'Next'.

6. Windows will ask 'What do you want the wizard to do?' Select the second option 'Install the hardware that I manually select from a list (Advanced)' and press 'Next'.



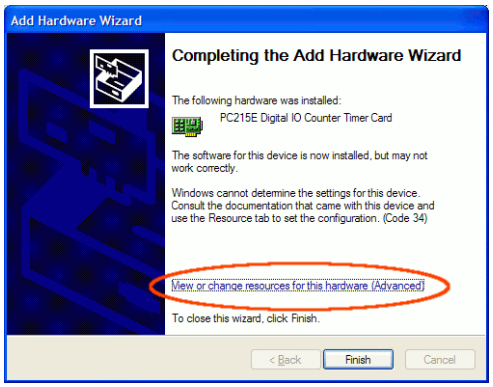

7. Windows shows a list of hardware categories. Select the 'Amplicon Analogue Digital IO Counter Timer Cards' category if it exists, otherwise select the 'Other devices' category. Then press 'Next'.

8. Press 'Have Disk...'. On the pop-up dialog, press 'Browse…' and browse to the AMPDIO software installation directory on the hard disk, or the root directory on the SOFTMAN CD-ROM. Then press 'OK'.

9. Select the card type you have installed from the list, then press 'Next'.

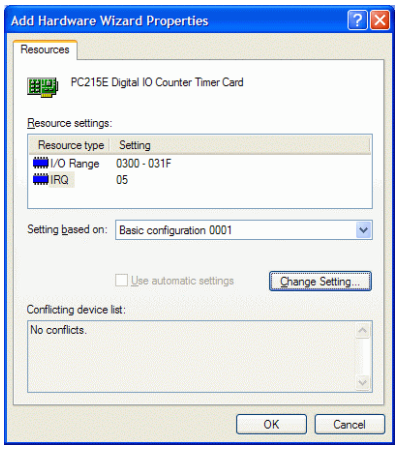
10. On the page labelled 'The Wizard is ready to install your hardware', press 'Next'.

11. Windows will install the driver and reach the 'Completing the Add Hardware Wizard' page. On this page, press the link labelled 'View or change resources for this hardware (Advanced)' to open the 'Add Hardware Wizard Properties' page.




12. On the 'Add Hardware Wizard Properties' page, press the button labelled 'Set Configuration Manually'.

13. If the settings match the base address and IRQ set on the card's DIP switches and jumpers, then press 'OK', then 'Finish', then on the pop-up dialog press 'Yes' to allow Windows to reboot. If the settings do not match then carry on with the following:




14. In the 'Settings based on' drop-down list, select 'Basic Configuration 0001' (or select 'Basic Configuration 0002' if the card's IRQ jumper has been removed).

15. In the 'Resource settings' list, select the resource you wish to change, press the 'Change Setting' button, correct the value using the up and down buttons and press 'OK'. Repeat for the other resources you wish to change.

16. When you are happy with the new resource settings, press ‘OK’, then ‘Finish’, then on the pop-up dialog press ‘Yes’ to allow Windows to reboot.

### 2.3.4 Installing a card in Windows 2000

For versions of the AMPDIO software prior to 4.32, please follow the instructions for installing a card in Windows NT 4.0 (see section 2.3.5). For versions 4.30 and 4.31, the supplied AMPDIOV4.INF file will allow the supported PCI cards to appear under Device Manager, but these are just dummy entries. For versions prior to 4.30 the supported PCI cards will appear as unknown devices under Device Manager.

For AMPDIO software versions 4.32 and later a ‘Plug and Play’ Windows 2000 driver is used. This section describes how to install a card to use this Plug and Play driver under Windows 2000.

#### 2.3.4.1 PCI Card

For a PCI card, Windows will detect the new hardware automatically and attempt to install the drivers. The driver files can be installed from the AMPDIO software installation directory or from the SOFTMAN CD-ROM.

To install the PCI card automatically on system start-up, do the following:

1. If installing from CD-ROM rather than from the AMPDIO software installation directory, ensure the Amplicon SOFTMAN CD-ROM is in the CD-ROM drive.

2. If Windows opens the ‘Welcome to the Found New Hardware Wizard’ page, press ‘Next’ and go to step 3. If Windows just asks for a disk labelled ‘Amplicon DIO Drivers Disk’ go to step 7.


3. Select the ‘Search for a suitable driver for my device (recommended)’ option and press ‘Next.

4. If installing from the CD-ROM, check the ‘CD-ROM drives’ option. If installing from AMPDIO software installation directory, check the ‘Specify a location’ option. Press ‘Next’.

5. If installing from the AMPDIO software installation directory, browse to the directory, press ‘Open’, then ‘OK’.

6. On the ‘Driver Files Search Results’ page, Windows should say ‘Windows found a driver for this device’. Press ‘Next’.

7. If Windows asks for a disk labelled ‘Amplicon DIO Drivers Disk’ when trying to copy files, click “OK” to cancel the alert box, then browse to the root directory on the CD-ROM (or the AMPDIO software installation directory on the hard disk) and press ‘Open’, then ‘OK’. Windows will copy the files and install the driver.

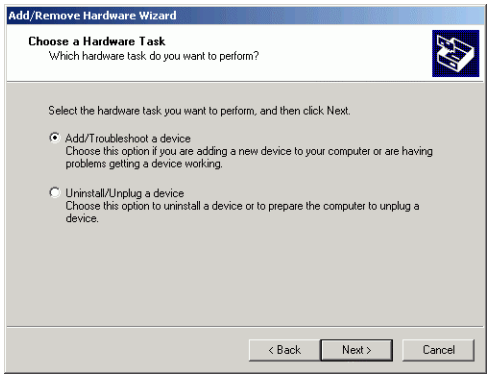
8. On the ‘Completing the Found New Hardware’ screen, press ‘Finish’

#### 2.3.4.2 ISA Card

To install an ISA ADIO card in Windows 2000 (with the Plug and Play driver) do the following:

1. Go to the Control Panel by pressing START > Settings > Control Panel.

2. Double click on the 'Add/Remove Hardware' icon. Press 'Next'.

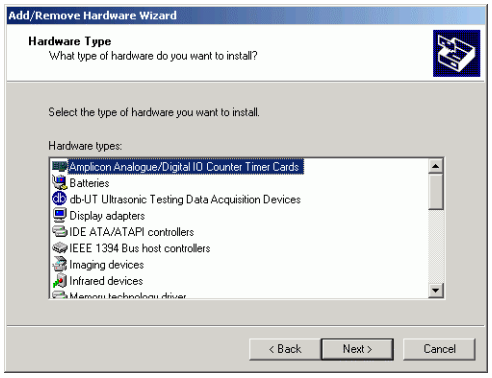



3. Select 'Add/Troubleshoot a device'. Press 'Next'.

4. Windows will search for Plug and Play hardware. Assuming it finds none, it will present a list of installed devices. Select 'Add a new device'. Press 'Next'.

5. To the question 'Do you want Windows to search for your new hardware?' select 'No' then press 'Next'.

6. Select the 'Amplicon Analogue Digital IO Counter Timer Cards' hardware type if it exists, otherwise select the 'Other devices' type. Then press 'Next'.




7. Press 'Have Disk', browse to the AMPDIO software installation directory or the root directory of the CD-ROM, select the AMPDIO.INF file (or AMPDIOV4.INF for versions prior to v5.00), press 'Open', then press 'OK'.

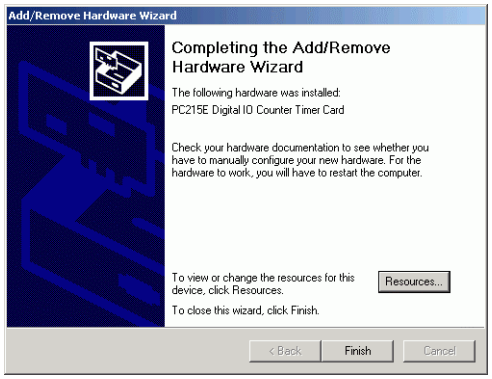
8. Select the card type you have just installed from the list, then press 'Next'.

9. On the pop-up dialog which says 'Windows could not detect the settings of this device', press 'OK'.

10. On the page listing the resources (Input/Output Range and Interrupt Request), press 'OK'.

11. On the 'Start Hardware Installation' page, press 'Next'.

12. On the 'Completing the Add/Remove Hardware Wizard' page, press the 'Resources' button.

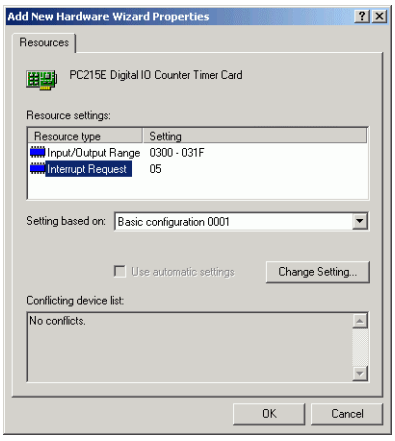



13. If the settings match the base address and IRQ set on the card's DIP switches and jumpers, then press 'OK', then 'Finish', then on the pop-up dialog press 'Yes' to allow Windows to reboot. If the settings do not match then carry on with the following:-

14. In the 'Settings based on' drop-down list, select 'Basic Configuration 0001' (or select 'Basic Configuration 0002' if the card's IRQ jumper has been removed).

15. In the 'Resource settings' list, select the resource you wish to change, press the 'Change Setting' button, correct the value using the up and down buttons and press 'OK'. Repeat for the other resources you wish to change.

16. When you are happy with the new resource settings, press 'OK', then 'Finish', then on the pop-up dialog press 'Yes' to allow Windows to reboot.



### 2.3.5 Installing a card in Windows NT 4.0

#### 2.3.5.1 PCI Card

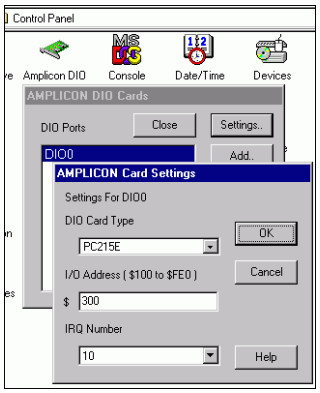
Note: the driver will automatically detect and install PCI ADIO cards to a spare DIO port name in the range DIO0 to DIO255. The 'Amplicon DIO' control panel applet may be used to display the settings for these cards and may be used to configure a card not to use interrupts.

The DIO port names of PCI cards that have been removed are remembered. The DIO port name will be restored if the PCI card is reinstalled. If a new PCI ADIO card is installed and the driver cannot find a spare DIO port name to assign it to, the card will not be available for use. In this case, the control panel applet may be used to delete one of the DIO port entries (e.g. for a PCI card which is no longer installed). When the AMPDIO driver is next restarted (e.g. by rebooting the system) the driver will assign the new PCI card to the spare DIO port name.

#### 2.3.5.2 ISA Card

To install an ISA ADIO card in Windows NT 4.0 you will need to use the control panel applet supplied.

1. Select Add.
2. Configure the required card type, base address and IRQ settings.
3. Select OK and allow the system to reboot.




### 2.3.6 Installing a card In Windows 95/98/ME

#### 2.3.6.1 PCI Card

For a PCI ADIO card, Windows 95, 98 or ME will detect the new card on boot up and attempt to install the drivers. The drivers can be installed from the root directory of the SOFTMAN CD or from the directory on the hard disk where the AMPDIO software was installed to (e.g. C:\AMPLICON\AMPDIO).

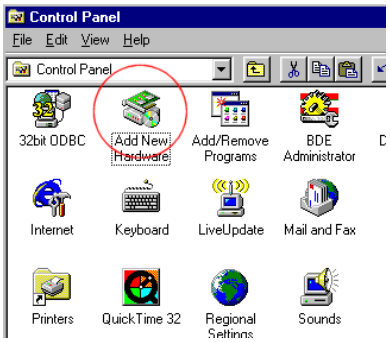
To install the PCI card automatically on system start-up, do the following:

1. If installing from CD-ROM rather than from the AMPDIO software installation directory, ensure the Amplicon SOFTMAN CD-ROM is in the CD-ROM drive.
2. If Windows fails to find a suitable INF file automatically, click on the 'Other Locations' button, browse to the root directory of the CD (or the AMPDIO software installation directory on the hard disk) and click 'OK'.
3. Windows should correctly identify the card. Click on 'Finish'.
4. If Windows asks for a disk labelled 'Amplicon DIO Drivers Disk' when trying to copy files, click 'OK' to cancel the alert box, then browse to the root directory on the CD-ROM (or the AMPDIO software installation directory on the hard disk) and click on 'OK'.
5. Windows will copy the files and install the driver.

### 2.3.6.2 ISA Card

To install an ISA ADIO card in Windows 95, 98 or ME do the following:

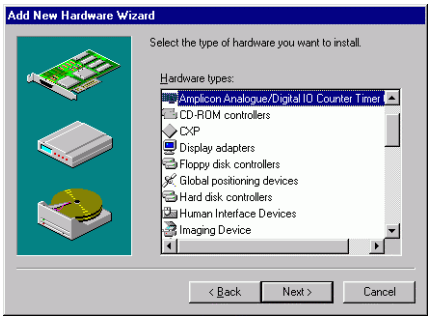
1. Go to the Control Panel by pressing START > Settings > Control Panel.




2. Double click on the 'Add New Hardware' icon. Then press 'Next'.

3. To the question 'Do you want Windows to search for your new hardware?' select 'No' then press 'Next'.

4. Select 'Amplicon Analogue Digital IO Counter Timer Cards' if it exists, otherwise select 'Other'. Press 'Next'.

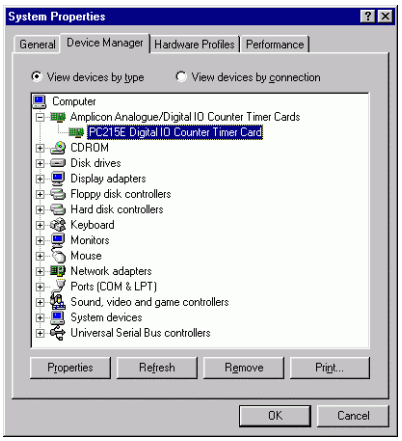



5. Press 'Have Disk' then 'OK' and browse to the root directory on the SOFTMAN CD, where the file AMPDIO.INF should be located (AMPDIOV4.INF for versions prior to v5.00). Alternatively, browse to the directory on your hard disk where the AmpDIO software was installed to (e.g. C:\AMPLICON\AMPDIO).

6. Select from the list the card type you have just installed then press 'Next'.

7. Make a note of the settings Windows has defaulted the card to (I/O Range and Interrupt) then press 'Next', then 'Finish'.

8. If the Windows default settings match the card's jumper settings then allow Windows to reboot (installation is now complete) else carry on with the following :



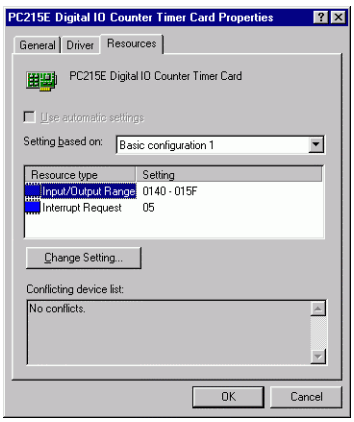

9. If the Windows default settings do not match the card's jumper settings, do not reboot yet. Go to START > Settings > Control Panel.

10. Double click the 'System' icon in the Control Panel window and then select the 'Device Manager' tab on the System Properties page.

11. Double Click on 'Amplicon Analogue Digital IO Counter Timer Cards'.

12. Click on your card.

13. Select Properties > Resources.




14. In 'Settings Based On' change from 'Basic Configuration 0' to 'Basic Configuration 1'.

15. Click on each item that needs to change, press 'Change Setting' and edit the item's value.

16. When you are happy with the settings, Click OK and allow Windows to REBOOT.

### 2.3.7 Installing Multiple Boards in a Single Host PC

More than one ADIO board may be installed in a single host PC. Furthermore, any combination of boards supported by the driver may be installed in a single host PC. Subject to the number of physical slots and resources, the maximum number of boards supported by the driver is 256.

1. To install more than one board in the host PC, the following points should be checked:
2. Sufficient space is available to mount the required number of boards.
3. Sufficient power is available for all the plug-in boards and adapters. Each PC214E requires +5V at up to 100 mA.
4. For none plug and play boards, check the base address of each board is set by switch to a different value, preferably at contiguous even addresses, and with no conflict with other installed devices. Suitable base addresses for four boards could be $300_{16}$, $320_{16}$, $340_{16}$ and $360_{16}$.
5. For none plug and play boards, check that the interrupt level (IRQ) of each board is set by jumper to a different value, and with no conflict with other installed devices.

Once the boards are installed into the PC and you must follow the steps outlined above to install the boards into your operating system.

# 3 DRIVER FUNCTIONS AND CONCEPTS

This chapter describes the functions and concepts of the AMPDIO driver and library. Details of the underlying register structures and software are given in chapters 4 and 5 respectively.

Reference should also be made to chapter 2.

The driver is shipped with a Windows dynamic link library (DIO_TC.DLL) written in C and example programs written in a number of different Windows visual programming languages. The DLL contains functions that implement typical applications for the supported devices. The source code for the DLL and the examples programs is shipped as part of the driver. As digital and analogue I/O boards can be used for a vast variety of tasks, the DLL and examples are provided as a demonstration of how to interface to the driver and are not intended as a definitive set of functions.

The DLL provides an example Windows application interface to Analogue and Digital logic on Amplicon data acquisition cards.

- It supports the industry standard 82C55 CMOS Programmable Peripheral Interface device.
- It supports the industry standard 82C54 CMOS Counter/Timer device (82C53 is supported on ISA analogue I/O cards).
- It supports analogue data acquisition.
- It supports digital to analogue conversion.
- It supports interrupts.
- It supports transferring large buffers of information under interrupt control.
- It allows boards to be configured in a variety of operating modes.

The driver originally supported the 82C55 PPI and 82C54 counter timer devices as implemented on the 200 series digital IO cards, i.e. boards in the PC215E, PC212E, PC218E and PC272E range. It has since been expanded to support a range of analogue cards. The 82C55 PPI and 82C53/4 counter timer devices are supported using the same model as that developed for the 200 series card so it is important to have an understanding of this family of cards. This register structure is outlined in chapter 4.

Note that the 200 series cards and the PCI data acquisition cards support software-programmable counter/timer clock and gate connections for use in configuring the 82C54 Timer Counter chips. For boards that do not, special care must be taken to configure the jumpers and external connections before using library functions.

Also please take into consideration the limits on the input and output frequencies when using the timer/counter functions. These limits arise because the software was written to support the whole range of Digital I/O and Timer/Counter boards, some of which have software selectable clock sources.

## 3.1 Timer Counter Functions

The library supports a number of different applications of the Timer Counter logic that do not require the use of interrupts.

### 3.1.1 Differential Counter

Two timer/counters can be used to form a Differential Counter pair from which the *ratio* of, or the *difference* between, the two count values is derived. See section 6.4.6.

The function *TCsetDiffCounters* allows you to specify the two timer/counters to be used as a differential pair. The function registers the timer/counter pair as being 'in use' and unavailable for

any other application. Provision is also made by *TCsetDiffCounters* to specify the clock and gate connections for both timer/counters.

The functions *TCgetDiffCount* and *TCgetRatio* can be called at any time after *TCsetDiffCounters*, and these two functions latch and read the current count values of the timer/counters, using the read-back command, and return the difference and ratio of the two count values respectively. Function *TCfreeDiffCounters* can be called when finished with the differential counter, and releases the timer/counter pair so they become available for use by another application.

### 3.1.2 Monostable Multivibrator

Mode 1 of the 82C54 timer/counter provides a digital one-shot output. This can be used to implement a monostable multivibrator pulse. In this mode, the output of the timer goes low at the falling edge of the first clock succeeding the gate trigger, and is kept low until the counter value becomes 0. Once the output is set high, it remains high until the clock pulse succeeding the next gate trigger.

Function *TCsetMonoShot* allows you to specify a timer/counter and a monostable pulse duration (in seconds). See section 6.4.8.1. The function calculates the initial count value required to generate the specified pulse length, and programs the timer/counter accordingly. Normally, the counter/timer's internal clock source is selected automatically by the function, but in the case of the legacy cards, the user must ensure the relevant jumper settings are selected correctly for one of the following ranges of possible pulse duration times:

| **Output pulse duration range** | | **Input clock frequency** |
|---|---|---|
| **Min** | **Max** | |
| 200 ns | 6.5 ms | 10 MHz |
| 0.2 µs | 65 ms | 1 MHz |
| 2.0 µs | 650 ms | 100 kHz |
| 20 µs | 6.5 s | 10 kHz |
| 0.2 ms | 65 s | 1 kHz |

It is the responsibility of the user to provide the external gate signal to trigger the monostable output.

### 3.1.3 Astable Multivibrator

An extension of the monostable multivibrator is to have two such timer/counters each generating an output pulse of specified duration, but each being triggered by the end of the other timer/counter's pulse. By adjusting the two pulse duration times, an astable multivibrator waveform with a given frequency and mark-to-space ratio can be attained.

This application is implemented in function *TCsetAstable* — see section 6.4.8.2. The *msratio* argument to the function specifies the mark-to-space ratio, and this is defined as follows:

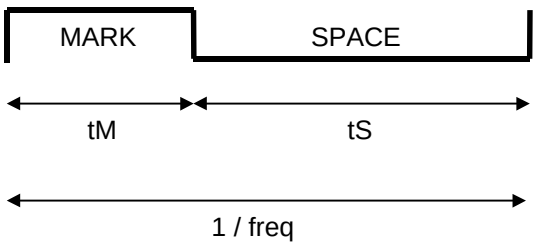
**mark-to-space ratio = mark time / overall period**

The function registers the timer/counters as being 'in use' and unavailable for any other application. Function *TCfreeAstable* can be called when finished with the astable multivibrator, and releases the timer/counters so they become available for use by another application.

The output of each timer/counter must be connected externally via the user connector, SK1, to the gate input of the other timer/counter.

The *TCsetAstable* function calculates the input clock frequencies and counter divide ratios (CDRs) for the two timers and normally makes the selections automatically. However, for some boards the

clock selections must be made by hand, and therefore a discussion of the calculations involved are necessary to obtain the correct input clock source jumper selections.




**tM = msratio / freq** where *msratio* = mark to space ratio
*freq* = output frequency (Hz)
*tM* = mark time (seconds).

The CDR for the 'mark' timer/counter, *cdrM*, is defined as

**cdrM = tM** x **fClkM** where *fClkM* = 'mark' timer's input clock frequency.

The equation for cdrM should be iterated for various values of *fClkM*, starting at 10 MHz and working down, until the result gives a value for *cdrM* that is less than $FFFF_{16}$ (the maximum value for a CDR). When this is attained, a suitable input clock frequency has now been found. A similar calculation can now take place for *tS*, with

**tS = (1 / freq) – tM**

**cdrS = tS** x **fClkS** where *tS* = space time (seconds)
fClk*S* = 'space' timer's input clock frequency
*cdrS* = CDR for 'space' timer/counter.

Note: the 82C54 timer/counters outputs are switched to the low level by the *next clock* after the gate trigger, possibly causing the mark-to-space ratio to become distorted by one or two clock pulses. This will become more apparent at higher frequencies.

### 3.1.4 Stopwatch

In mode 2, the output of the 82C54 timer/counter starts high; goes low for one clock pulse when the count value decrements to 1, and then is set to high again. The initial count value is then automatically re-loaded; counting continues and the sequence repeats. The output can be used as a clock signal for another timer/counter, and any number of timer/counters can be cascaded in this way. See section 6.4.7.

The *TCsetStopwatch* function sets up two timer/counters in this way with a clock input frequency of 1 kHz. Function *TCstartStopwatch* sets the counters counting, and function *TCgetElapsedTime* latches and reads the two count values to calculate the elapsed time, in milliseconds, since the counters were first set off by *TCstartStopwatch*. This stopwatch can count milliseconds for nearly 50 days. Function *TCfreeStopwatch* releases the timer/counters so they can become available for use by another application when the stopwatch is no longer required.

### 3.1.5 Frequency/Period Measurement

Another use for the pulse generation capabilities of the 82C54 is for one counter/timer to provide a precise GATE signal during which a second timer/counter counts an external event. In mode 0, a high level on the gate input validates counting, and a low level invalidates it (i.e. counting stops).

Also a low-to-high transition on the gate input causes the initial count value to be re-latched into the counting element.

Two functions *TCgetExtFreq* and *TCgetExtPeriod* (see sections 6.4.9.1 and 6.4.9.2) are used to program a timer/counter to provide a one-shot gate pulse of precise duration to a second timer/counter. The second timer/counter has an external signal as its clock input. When the gate pulse is over, the second timer/counter's counting stops, and its value is then read. A simple calculation is then made to determine the number of external clock cycles received during the gate period, and from this the external frequency and period can be estimated. An increasing sequence of gate periods (6.5535 ms, 65.535 ms, 655.35 ms, 6.5535 s and 65.535 s) is tried until a sufficiently accurate count (at least 1000 ticks) is read from the second timer/counter. On cards without clock and gate connection registers, such as the PC214E, a fixed gate period of 6.5535 ms is assumed.

Version 4.42 of DIO_TC.DLL provides another frequency measurement function *TCgetExtFreqRestricted* (see section 6.4.9.3). This uses two timer/counter channels in the same way as *TCgetExtFreq*, but the frequency is external frequency is measured by counting external clock pulses over a specified period. The function also indicates whether the 16-bit counter used to measure the frequency overflowed during the measurement period.

The timer/counter you specify in calls to *TCgetExtFreq*, *TCgetExtPeriod* and *TCgetExtFreqRestricted* is the second timer/counter described above. In cards that support programmable gate configuration, the counters will be configured automatically. For certain legacy cards the user must set appropriate jumpers. To use Z1 counter 2 on the PC214E, the following connections must be made:

1. Connect the external TTL signal to SK1 pin 36 with reference to GND on, say SK1 pin 56.
2. Remove jumper J4
3. Place jumper J2 in position 1 (10 MHz) for *TCgetExtFreq* or *TCgetExtPeriod*, or in position 2 (1 MHz) for *TCgetExtFreqRestricted*.
4. Link SK1 pin 54 (Z1 /OUT0 O/P) to SK1 pin 75 (Z1 GAT2 I/P)

### 3.1.6 Frequency Generation

In mode 3 the output of the timer/counter is a periodic square wave, whose frequency is the input clock frequency divided by the programmed counter divide ratio (CDR). The function *TCgenerateFreq* (see section 6.4.8.4) calculates the CDR required for a specific frequency on a given timer/counter. Normally the function selects an appropriate input clock frequency but, since the PC214E does not support software-programmable clock connections, the clock input must be set as 1 MHz on the appropriate jumper. For the PC27E, the clock input is fixed at 4 MHz.

The function *TCgeneratePulse* (see section 6.4.8.6) is provided as a variant of *TCgenerateFreq*. This uses mode 2 instead of mode 3, which results in a periodic negative-going pulse instead of a square wave. The width of the pulse is the period of the input clock.

### 3.1.7 Frequency Multiplication

An extension of the frequency measurement and frequency generation capabilities described in sections 3.1.5 and 3.1.6 above is to combine the two into a process that measures an external frequency on one timer/counter; multiplies the frequency value by a given factor and generates this new frequency on a second timer/counter. Function *TCmultiplyFreq* described in section 6.4.9.4 performs this operation. See sections 3.1.5 and 3.1.6 above for connection details.

### 3.1.8 Pulse Train Generation

By connecting the gate input of a frequency generator to the inverted output of another timer/counter channel, it is possible to generate a fixed number of negative-going pulses within a fixed period of time. These pulses will be narrow pulses if mode 2 is used for the frequency generator channel (the 'pulse' channel), or square if mode 3 is used.

The duration of each train of pulses may be set by using the inverted output of a monostable multivibrator (the 'one-shot' channel running in mode 1) as the gate input of the frequency generator. The function *TCsetOneShotPulseTrain* described in section 6.4.8.20 allows a fixed number of pulses to be output on the pulse channel during the one-shot pulse. The gate input of the one-shot channel is used as the trigger.

By triggering the one-shot periodically from the output of a third timer/counter channel (the 'train' channel running in mode 2) a periodic pulse train generator is produced. The function *TCsetPeriodicPulseTrain* described in section 6.4.8.7 allows the duration and number of pulses within each pulse train to be set, and the frequency of the pulse trains to be set.

A variant of the periodic pulse train generator uses the inverted output of the train channel directly as the gate input of the pulse channel directly with no one-shot channel between the two. The duration of the pulse train is restricted to the period of the input clock for the train channel. The function *TCsetRestrictedPulseTrain* described in section 6.4.8.14 uses this mechanism.

In all cases, only the 'pulse' (output) channel is specified and the other timer/counter channels are offset from this. For boards with timer/counter clock connection registers and timer/counter gate connection registers, everything can be set-up automatically by the functions. For the PC214E it is necessary to wire up the gate inputs and trigger inputs manually on connector SK1 and to specify to functions which of the predefined input clock sources to use, corresponding to the jumper settings.

### 3.1.9 Pulse Width Modulation

It is possible to generate a frequency with a variable mark-to-space ratio by using the *TCsetAstable* function (see section 3.1.3), but it is inconvenient to wire two one-shot monostables back to back. An alternative is to use a single hardware-retriggerable one-shot triggered periodically from the output of another timer/counter programmed in mode 2 to generate the desired frequency. This connection can be made internally using the gate connection registers on cards that have them. The *TCsetPWMTrain* function described in section 6.4.8.31 can be used to set up such a frequency generator with programmable space-to-mark ratio. Related functions are provided to change the space-to-mark ratio or the frequency after it has been set up.

The *TCsetPWPulse* function described in section 6.4.8.26 provides just the one-shot part of the above without the frequency generator. This performs a similar function to the *TCsetMonoShot* function, but the output low pulse duration is specified as an assumed overall period multiplied by a space-to-mark ratio. Related functions are provided to change the space-to-mark ratio and the overall period, but the function only has direct control over the length of the output low period.

### 3.1.10 Event Counter

Two timer/counter chips can be cascaded to provide a 32-bit count of clock pulses, which can come from an external source. A clock pulse consists of a rising edge followed by a falling edge. The *TCsetEventCounter* function described in section 6.4.7.8 provides this facility. The *TCgetEventCount* function described in section 6.4.7.10 is used to read the current event count. The count can be reset to zero by calling the *TCresetEventCount* function described in section 6.4.7.9. The function *TCfreeEventCounter* described in section 6.4.7.11 should be called to free up the timer/counter resources when the event counter is no longer required.

The specified timer/counter channel is set to mode 2 with an initial count of 65536 and divides the event clock. The output is fed to the input of another timer/counter channel offset by +1 from the specified channel. This second channel is set to mode 0 with an initial count of 65536. The initial count of the second channel is loaded by toggling the mode of the first channel. The initial count of the first channel is loaded on the first event clock pulse. To read the event count, the status of the two channels and their counts are read. The status indicates whether the first event clock has occurred yet (by checking whether the initial count of the first channel has been loaded) and indicates whether the 32-bit count has overflowed (by checking the output state of the second channel).

The 32-bit event counter functions do not work on the PC24E, PC25E, PC26AT, PC27E or PC30AT because the status of the timer/counter chips cannot be monitored on these cards.

## 3.2 Digital I/O Functions

The library supports a number of basic digital I/O functions that do not require the use of interrupts.

### 3.2.1 Basic Digital I/O

The library supports basic digital I/O for mode 0 of the 82C55 programmable peripheral interface chip. The 82C55 ports A, B and C can be configured as inputs or outputs using the *DIOsetMode* function (see section 6.4.11.2). Port C is split into two 4-bit halves, individually configurable as inputs or outputs.

The library supports the concept of digital I/O channels. The channels can be 1, 2, 4, 8, 12 or 24 bits wide. The *DIOsetChanWidth* function (see section 6.4.11.4) is used to configure channel widths. The channels are mapped onto the physical ports in the following way.

| Channel Width | Channel Mapping | Port Mapping |
|---|---|---|
| 1 | X0..X7<br>X8..X15<br>X16..X24 | Port A bits 0,1,2,..,7<br>Port B bits 0,1,2,..,7<br>Port C bits 0,1,2,..,7 |
| 2 | X0..X3<br>X4..X7<br>X8..X11 | Port A bits 0&1,2&3,..,6&7<br>Port B bits 0&1,2&3,..,6&7<br>Port C bits 0&1,2&3,..,6&7 |
| 4 | X0<br>X1<br>X2<br>X3<br>X4<br>X5 | Port A bits 0 to 3<br>Port A bits 4 to 7<br>Port B bits 0 to 3<br>Port B bits 4 to 7<br>Port C bits 0 to 3<br>Port C bits 4 to 7 |
| 8 | X0<br>X1<br>X2 | Port A bits 0 to 7<br>Port B bits 0 to 7<br>Port C bits 0 to 7 |
| 12 | X0<br>X1 | Port A and Port C bits 4..7<br>Port B and Port C bits 0..3 |
| 24 | X0 | Port A bits 0 to 7<br>Port B bits 0 to 7<br>Port C bits 0 to 7 |

Note that the 12-bit channel configuration does not follow the same pattern as the others. The 12-bit channels are arranged in a way that is compatible with mode 1 and mode 2 strobed data communication.

The *DIOsetData* and *DIOgetData* functions (see sections 6.4.11.5 and 6.4.11.6) are used to read and write the digital channels.

There are also lower level functions available. These functions were first implemented in version 2.0x of the library. They allow direct programming of the 82C55 ports without using the channel concept and allow modes 1 and 2 to be selected. The extra functions are *DIOsetModeEx*, *DIOgetModeEx*, *DIOgetDataEx* and *DIOsetDataEx* (see sections 6.4.11.7, 6.4.11.8, 6.4.11.9 and 6.4.11.10). With these functions, the supplied value is written directly to the associated 82C55 device. The function of the 82C55 is detailed in section 5.4.1.

The 82C55 chip is normally operated in mode 0. The *DIOsetModeEx* function (see section 6.4.11.7) can be used to write an arbitrary value to the 82C55's control port. This can be a mode-setting command or a single bit set/reset command (useful in modes 1 and 2).

Setting the mode using the *DIOsetMode* or *DIOsetModeEx* functions causes all configured output bits to be set to the logic level 0 (0V).

### 3.2.2 Switch Matrix

The high numbers of digital I/O channels available on the 82C55 PPI devices lend themselves to a switch matrix scanner implementation. The status of a matrix of switches can be obtained by sending test patterns into the matrix, and then reading status patterns back from the matrix.

Section 6.4.12 describes functions that allow PPIX, both PPIX and PPIY, or PPIX, PPIY and PPIZ to be used as such a device. Using only PPIX, up to 144 switches can be scanned; using both PPIX and PPIY, up to 576 switches can be scanned; using PPIX, PPIY and PPIZ, up to 1296 switches can be scanned. Group 'A' ports of the 82C55 device(s) (Port A and Port C-upper) are set for output to send test patterns to the matrix. Group 'B' ports (Port B and Port C-lower) are set for input to read the switch status information. The user must ensure that the switch matrix is wired as detailed below.

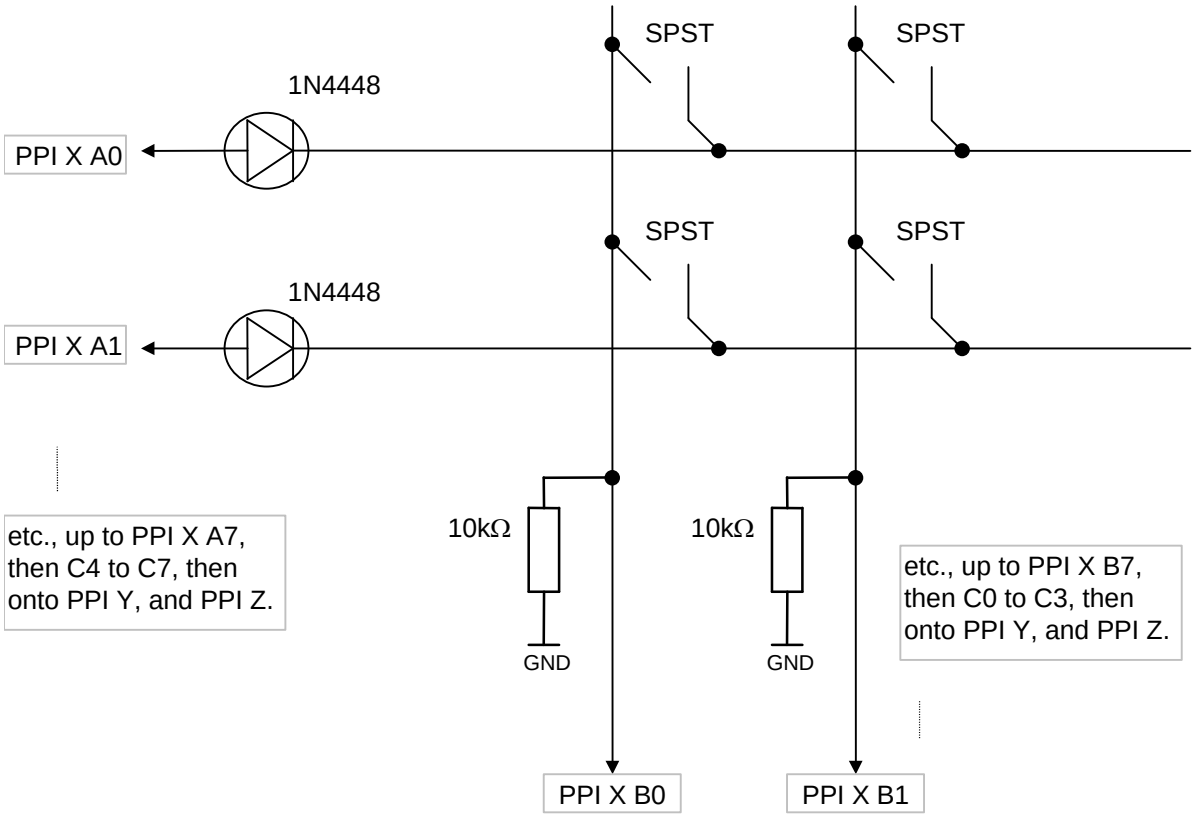



**Figure 1 – Switch Matrix Configuration**

Function *DIOsetSwitchMatrix* allows you to set up the matrix, specifying the matrix order. For the PC214E, this can be 12 X 12 or 24 X 24 switches. The function also registers the PPIs used as being 'in use' and unavailable for use by other programs. Function *DIOgetSwitchStatus* returns the status of a given switch in the matrix, and function *DIOfreeSwitchMatrix* frees the PPIs so they can be used by other programs when the switch matrix is no longer required.

## 3.3 Basic Analogue I/O Functions

The library supports a range of basic analogue I/O functions. Basic analogue I/O functions do not require the use of interrupts.

### 3.3.1 Determining Analogue Resources

The library includes functions to determine what analogue resources are on the card. *AIOcountADCchans* and *AIOcountDACchans* allow the user to determine the precise number of analogue to digital converters (ADCs) and digital to analogue converters (DACs) available on a card (see section 6.4.17).

### 3.3.2 Channel Masks

A number of cards support multiple digital to analogue or analogue to digital converters. These are either implemented as individual devices mapped into different areas of the I/O space or as single devices with a multiplexed input stage. The channel mask is used to allow these individual channels to be configured independently.

The analogue configuration functions accept a Channel Mask parameter. This parameter allows individual channels to be configured differently. In order to configure a channel the associated bit must be set to one, e.g. for channel 0 set bit 0, for channel 2 set bit 2:

ChanMask: 5 = $0000000000000101_2$ selects channels 0 and 2.

### 3.3.3 Channel Groups

If a card supported multiple sets of multiplexed analogue I/O or a mixture of multiplexed and non-multiplexed analogue I/O of the same type, then the analogue resources would be considered to be in different groups. No currently supported card has more than one ADC channel group or DAC channel group. Therefore, in functions that require an ADC or DAC channel group to be specified, the group parameter should be set to 0 for all currently supported cards.

### 3.3.4 Configuring Channels as Bipolar or Unipolar

The analogue channels can be bipolar or unipolar. In bipolar mode, the signal voltage can be negative or positive with respect to a reference. In unipolar mode, the signal voltage must be positive with respect to the reference.

The driver software treats channels differently depending on whether they are unipolar or bipolar. Channels can be marked as being configured for unipolar or bipolar operation using the *AIOsetADCchanMode* and *AIOsetDACchanMode* functions (see sections 6.4.18.3 and 6.4.18.12). These functions only affect how the driver software *cooks* ADC data read from the channel or *uncooks* DAC data written to a channel. The cooked data representation allows the interpretation of data values to be consistent across different cards. This setting is known as the *software* unipolar/bipolar setting for the card.

Cooked data to be written to a DAC channel is uncooked in a card-specific way by the driver before being output to the DAC. Raw data read from an ADC channel by the driver is cooked in a card-specific way before being passed to the user.

The cooking or uncooking of data depends on the mode the channel is in, either unipolar or bipolar. In general, the cooked data ranges are as follows:

| | |
|---|---|
| Unipolar: | 0 to 65535 (0 volts to max volts) |
| Bipolar: | 32768 to +32767 (–max volts to +max volts) |

For cards which have DAC or ADC channels which cannot be set to unipolar mode in hardware (e.g. PCI234), the above unipolar range is reinterpreted so that 0 maps to the minimum voltage (–max volts).

For cards which have DAC or ADC channels which cannot be set to bipolar mode in hardware, it is intended for the above bipolar range to be reinterpreted so that –32768 maps to 0 volts. Unfortunately, this reinterpretation of the bipolar range is not performed for the PC25E. This is because the AMPDIO software does not currently distinguish the PC25E from the PC24E and the PC24E *does* support bipolar mode in hardware (via jumper settings).

The above functions accept a ChanMask parameter that determines which channel(s) to configure. There are also the *AIOsetAllADCchanMode* and *AIOsetAllDACchanMode* functions that allow all channels in a group to be configured the same way (see sections 6.4.18.5 and 6.4.18.14). It is more likely that these functions will be used, as on most cards, as on most cards all channels have to be set as either unipolar or bipolar, and a mixture is not supported.

As well as configuring the software in bipolar/unipolar mode, it is also necessary to configure the analogue hardware using on card jumpers or the specially provided hardware configuration functions. If this is not done, output voltages or input readings may not be as expected. On the supported PCI cards, the *AIOsetAllADCchanMode* and *AIOsetAllDACchanMode* will change the hardware settings to match the software unipolar/bipolar configuration. There are also *AIOsetHWADCchanMode* and *AIOsetHWDACchanMode* functions to change the hardware unipolar/bipolar settings in a card-specific way without affecting the software unipolar/bipolar settings (see sections 6.4.18.4 and 6.4.18.13).

### 3.3.5 Basic Analogue Input

In order to read an analogue voltage, the ADC channel group's conversion source must be configured. The *AIOsetADCconvSource* function (see section 6.4.19.1) allows the conversion source to be specified. It is usual to set this to software conversion (CNV_SW) unless interrupts are being used. For ISA cards, the selected conversion source should agree with the jumper settings.

Once software conversion has been configured, the required analogue channel can be selected using the *AIOsetADCmultiplexer* function and a software-trigger conversion can be started with the *AIOstartADCconversion* function. The analogue data can then be read using *AIOgetADCdata* (see section 6.4.19). It is possible to call *AIOsetADCmultiplexer* between calls to *AIOstartADCconversion* and *AIOgetADCdata*. The data read is from the channel that was active when *AIOstartADCconversion* was called.

### 3.3.6 Basic Analogue Output

The *AIOsetDACchanData* function (see section 6.4.20.1) is used to write to digital-to-analogue convertors. The value written depends on whether the channel is unipolar or bipolar. The output voltage or current produced will depend on what card type is used and how it is configured.

This function accepts an array of values as its data. The number of values supplied depends on the number of channels enabled in the channel mask. For example, if the channel mask is set to 5 ($101_2$), then an array of two values is required where:

- Array element [0] is the data for channel 0.
- Array element [1] is the data for channel 2.

### 3.3.7 Configuring Analogue Resources on PCI Cards

As the PCI analogue acquisition cards do not have jumpers, there are a number of functions that allow the analogue hardware to be configured.

The *AIOsetHWADCchanMode* and *AIOsetHWDACchanMode* functions (described in sections 6.4.18.1 and 6.4.18.12) determine whether on card channel hardware is configured as unipolar or bipolar. These functions do not affect the way the software converts data for the channels. For data to be converted correctly, each channel's *hardware* setting should agree with its *software* setting — see section 3.3.4.

The *AIOsetHWADCchanDiff* function (described in section 6.4.18.7) determines whether input channels are single-ended or differential. Settings are interpreted in a card type-specific way. In single-ended mode, the channel reads the input voltage with respect to a fixed reference (analogue ground reference). In differential mode the channels reads difference in the input voltage compared to another input pin (single-ended channels are paired up to form differential channels). The differential input mode of the PCI230 and PCI260 cards is actually pseudo-differential — it takes a reading from each of the two inputs with respect to the analogue ground reference and subtracts them.

The *AIOsetHWADCchanGain* function (see section 6.4.18.9) allows different gains to be associated with different input channels. The setting is interpreted in a card type-specific way. On the PCI230 and PCI260 each pair of channels (0&1, 2&3, etc.) has its own gain setting.

The *AIOsetHWDACchanRange* function (see section 6.4.18.16) allows different ranges to be selected. The setting is interpreted in a card type-specific way. On the PCI224, it allows the output range to be set for the whole card, but there is no control over the output ranges of individual channels. On the PCI234, the output range is determined by a jumper setting and this function has no effect. It is not supported on any other cards.

## 3.4 Using Interrupts

There are a number of functions provided by the library that use interrupts to do specific tasks. There is also a mechanism supported by the library that allows user functions to be called on interrupt. The easiest way to use interrupts is to use one of the supplied interrupt functions.

### 3.4.1 Event Recorder

Using this function, it is possible to use a stopwatch (as described in section 3.1.4) to record the elapsed times when an external event occurs. In order to do this, the event's status output must be connected to an 82C55 digital input's port C0. A low-to-high transition on this pin causes an interrupt to occur. The driver-supplied interrupt service routine reads the elapsed time from the stopwatch timer/counters and store the time into memory.

The *TCsetEventRecorder* function (see section 6.4.7.4) allows you to specify a digital input chip (PPIX or PPIY) from which Port C bit 0 will be used as the event input, and interrupt source. Once the board's interrupt has been enabled (see function *enableInterrupts* — section 6.4.2.1) and a stopwatch timer has been started, a positive going signal on the PPI Port C bit 0 pin on SK1 will cause the elapsed time to be recorded into memory.

In order to determine whether any events have occurred, use the *getLongIntItem* function (see section 6.4.4.14). This returns the index of the current interrupt item in the supplied buffer. The Item can then be read using the *readLongBuf* function (see section 6.4.4.6).

When finished, the *TCfreeEventRecorder* function (see section 6.4.7.5) frees up the resources used so they can be used again by another service. This does not free the stopwatch or buffer.

### 3.4.2 Digitally Controlled Oscillator

The combination of the 82C55 PPI and 82C54 counter/timer devices make it possible to implement a digitally controlled oscillator, whereby the value of a binary number read into a PPI input channel is used to calculate the frequency of a square wave generated on a timer/counter output. To turn this process into a continuous background task, a second timer/counter can be deployed to generate an *update* signal by generating a periodic interrupt. The interrupt service routine then performs the DCO operation in the background. See section 6.4.10.

Function *TCsetDCO* sets up such an arrangement, allowing you to specify the digital input channel, the output timer/counter and the second timer/counter used to generate the update interrupts. The function also allows for a flexible update rate and output frequency range. The digital input channel width (i.e. the number of bits in the digital input word) can be selected to either 1, 4, 8, 12, 16 or 24 bits by calling function *DIOsetChanWidth* (see section 6.4.11.4 for more details). The PPI Port(s) used by the digital input channel must be programmed as input by calling function *DIOsetMode* for each port (see section 6.4.11.7).

The *enableInterrupts* and *disableInterrupts* functions (see sections 6.4.2.1 and 6.4.2.2) must then be called to enable and disable the 'update' interrupts, and, when finished, function *TCfreeDCO* frees up the resources used so they can be used again by another program.

When using the DCO function with the PC214E, please ensure the following connections are made:

1. The digital input bit(s) must be connected to the digital input channel specified
2. The output timer/counter clock source must be 1 MHz
3. The 'update' timer/counter MUST BE Z1 Counter 1 on the PC214E. The clock source must be 1 MHz (i.e. jumper J3 in position 2), and the frequency specified must be within the range 15 Hz to 500 kHz.
4. The interrupt source must be Z1 Timer/Counter OUT1 (i.e. jumper 5 in position 5).

### 3.4.3 Interrupt Callback

The driver and DIO_TC.DLL provide a mechanism that allows an external user supplied function to be called when an interrupt occurs. This is a very powerful mechanism. Using it, you can implement sophisticated interrupt routines without having to write a completely new interface to the low-level driver. A small percentage of users need this facility in order to use their Amplicon hardware effectively.

To use this facility you must be a competent programmer with knowledge of interrupts and of using Windows callback functions. The DLL provides an alternate mechanism that allows interrupts to be used without callback functions, by waiting or polling for the occurrence of an interrupt.

There are a number of examples of how to use this feature:

- Examples shipped with the driver.
- Examples written in C on www.mev.co.uk/suppdio.htm.
- The *TCsetEventRecorder* and *TCsetDCO* functions in the DIO_TC.DLL source.

Each Amplicon card has up to 6 individual sources that can cause an interrupt to occur (some have fewer than 6 interrupt sources). The ADIO library allows a user supplied function to be called when a particular interrupt event occurs, such as a timer interrupt or a certain PPI port pin going high.

Once the user function has been set up, the *enableInterrupts* and *disableInterrupts* functions (see sections 6.4.2.1 and 6.4.2.2) can then be called to enable and disable the interrupts, and, when finished, the *TCfreeUserInterrupt* function (see section 6.4.13.5) can be called to free up the resources so they can be used again by another service.

The DLL supports a number of different functions to set up user interrupts, some for digital resources, some for analogue resources, some for transferring buffers and some that have their own threads inside the DLL for callbacks and some that do not have their own threads and do not use callbacks. The interrupt set-up functions return a handle to refer to the user interrupt or an error code. Other functions are available for handling non-callback user interrupts.

| | |
|---|---|
| TCsetUserInterrupt | Sets up a basic user interrupt callback for a digital resource. |
| TCsetUserInterruptAIO | Sets up a basic user interrupt callback for an analogue resource. |
| TCsetBufferUserInterrupt | Sets up a buffered user interrupt callback for a digital resource. |
| TCsetBufferUserInterruptAIO | Sets up a buffered user interrupt callback for an analogue resource. |
| TCsetNCBufferUserInterrupt | Sets up a non-callback buffered user interrupt for a digital resource. |
| TCsetNCBufferUserInterruptAIO | Sets up a non-callback buffered user interrupt for an analogue resource. |
| TCdriveNCBufferUserInterrupt | Used to transfer buffers of data for non-callback user interrupts. |
| TCwaitNCBufferReady | Used to poll or wait until a non-callback buffered user interrupt is ready for the next data transfer, with timeout facility. |
| TCwaitMultiNCBufferReady | Used to poll or wait until one of several non-callback buffered user interrupts is ready for the next data transfer, with timeout facility. |
| TCfreeUserInterrupt | Frees resources for a user interrupt. |
| enableInterrupts | Enable user interrupts globally at the first level. |
| disableInterrupts | Disable user interrupts globally at the first level. |
| TCenableInterruptChip | Enable an individual user interrupt trigger source at the second level. Note that all user interrupt sources are initially enabled at the second level by the *registerBoard* functions, so this is not needed unless the user interrupt trigger source has been explicitly disabled. |
| TCdisableInterruptChip | Disable an individual user interrupt trigger source at the second level. Note that all user interrupt sources are initially enabled at the second level by the *registerBoard* functions. |

| | |
|---|---|
| TCenableUserInterrupt | Like *TCenableInterruptChip* except that the user interrupt trigger source is determined from a handle returned by one of the user interrupt set-up functions. |
| TCdisableUserInterrupt | Like *TCdisableInterruptChip* except that the user interrupt trigger source is determined from a handle returned by one of the user interrupt set-up functions. |

The user can ask the driver to read from or write to some of the card's I/O locations, read timer values, read an ADC channel, or write to DAC channels when that interrupt occurs and then call the user-supplied callback function when the required amount of data has been read or more data is required to be written.

The DIO_TC.DLL supports two different flavours of user interrupt callbacks. The simpler, non-buffered *basic* version supports reading a single data value (which may be various combinations of PPI port values, one or two timer values or a single ADC channel) on each interrupt and calling the callback function each time. For example, in the meter example, the driver reads the ADC channel and then calls the user function, which converts the value and displays it as a voltage.

The more general *buffered* version of user interrupt callback supported by DIO_TC.DLL allows data to be read from ports, timers or ADC channels into a buffer, or written out to ports or DAC channels from a buffer and calls the user interrupt callback function whenever enough data has been read over a number of interrupts to fill the buffer, or more data is required to be written over a number of interrupts. This flavour of user interrupt callback can be used in single-buffer mode, or double buffer mode. In double buffer mode, the user callback function can be handling one buffer whilst the driver is emptying or filling the other buffer, and the driver attempts to maintain the flow of data whilst buffers are being switched. The double buffer mode is also known as *continuous* mode.

The mechanism for user call back when reading data from the card on interrupt is:

1. Interrupt occurs.
2. Driver reads interrupt source register.
3. Driver reads appropriate I/O registers and fills in user data.
4. If necessary, the driver signals DIO_TC.DLL, which schedules the appropriate user-supplied function to be called.
5. Driver acknowledges interrupt and re-enables other system interrupts.
6. If scheduled, the user callback function is called by DIO_TC.DLL.

When writing data to the card on interrupt, the process is similar, but the user callback function is called to get data to be written before allowing an interrupt to occur.

You can either:

- have a number of different call back routines associated with each interrupt source (recommended)

or:

- have a generic callback routine and pass a constant in the wParam that identifies which interrupt source caused the interrupt.

Note that the user call back function executes sometime after the interrupt has occurred. This time delay is typically 20 μs but may be as long as 200 ms if the system is busy writing to disk, network or printer. This is because the call back is scheduled through the operating system and other drivers will take priority.

The actual interrupt service latency is variable from machine to machine, but is typically 20 μs. The data read during the interrupt service is read with this latency and not be subject to the delay

associated with callback. This means that by asking the driver for data you can be sure that the data is sampled soon after the event that caused the interrupt.

The driver supports the following data requests:

| | |
|---|---|
| ISR_NODATA = –1 | Reads value 0 from a null source. |
| ISR_READ_16COUNT = 0 | Reads a specified 16-bit counter (*Chip1*, *Chan1*). |
| ISR_READ_16COUNTSTAT = 16 | Reads status and count from a specified 16-bit counter (*Chip1*, *Chan1*). Bits 23 to 16 contain the status. Bits 15 to 0 contain the count. (For PC24E/PC25E, PC26AT, PC27E and PC30AT, bits 31 to 16 will contain the value $FFFF_{16}$.) |
| ISR_READ_32COUNT = 1 | Reads two specified 16-bit counters (*Chip1*, *Chan1*, *Chip2*, *Chan2*). The first specified counter is read into the low 16 bits; the second specified counter is read into the next 16 bits and is assumed to be clocked from the output of the first counter. From version 5.04 of the driver onwards, both counters are read a second time. If the second counter's value changes between the two readings, the second set of readings is used, otherwise the first set of readings is used. Earlier versions of the driver (version 3.00 onwards) only read the counters twice if their values couldn't be latched simultaneously (because they were on different chips or used the older 82C53 chip). |
| ISR_READ_32COUNTSTAT= 17 | Reads status and count from two specified 16-bit counters (*Chip1*, *Chan1*, *Chip2*, *Chan2*), yielding two data values on each interrupt. The status and count read from the first specified counter is formed into the first data value. The status and count read from the second specified counter is formed into the second data value. The second counter is assumed to be clocked from the output of the first counter. From version 5.04 of the driver onwards, both counters are read a second time. If the second counter's value changes between the two readings, the second set of readings is used, otherwise the first set of readings is used. Earlier versions of the driver (version 3.00 onwards) only read the counters twice if their values couldn't be latched simultaneously (because they were on different chips or used the older 82C53 chip). For each data value, bits 23 to 16 contain the status and bits 16 to 0 contain the count. (For PC24E/PC25E, PC26AT, PC27E and PC30AT, bits 31 to 16 will contain the value $FFFF_{16}$.) |
| ISR_READ_PPIABC = 5 | Reads PPI ports A, B and C (*Chip1*, [*Chan1*=0]). Port A is read into the low 8 bits, port B into the next 8 bits and port C into the next 8 bits. |
| ISR_READ_2PPIABC = 13 | Like ISR_READ_PPIABC but reads ports A, B and C from two specified PPI chips (*Chip1*, *Chip2*), yielding two data values on each interrupt. Ports A, B and C from the first specified PPI chip are formed into the first data value. Ports A, B and C from the second specified PPI chip are formed into the second data value. |
| ISR_READ_3PPIABC = 14 | Like ISR_READ_PPIABC but reads ports A, B and C from three fixed PPI chips (PPIX, PPIY and PPIZ), yielding |

| | |
|---|---|
| | three data values on each interrupt. Ports A, B and C from the first PPI chip (PPIX) are formed into the first data value. Ports A, B and C from the second PPI chip (PPIY) are formed into the second data value. Ports A, B and C from the third PPI chip (PPIZ) are formed into the third data value. |
| ISR_READ_PPIC = 6 | Reads port C of the interrupting PPI chip. |
| ISR_PC27 = 7 | Reads the raw analogue data from the PC27's ADC chip. Not recommended. |
| ISR_READ_DATA8 = 8 | Read an 8-bit I/O port (*Chip1*, *Chan1*). |
| ISR_READ_DATA16 = 9 | Reads two 8-bit I/O ports (*Chip1*, *Chan1*, *Chip2*, *Chan2*). The first port is read into the low 8 bits and the second port is read into the next 8 bits. |
| ISR_READ_ADCS = 10 | Reads an ADC channel, cooking the data (*Group*, *ChanMask*). The channel is then switched ready for the next reading. The channels specified by the channel mask bit-vector are switched between cyclically, starting with the lowest channel specified by the mask. If the ADC channel group being read is the source of the interrupt and it has a FIFO, then the interrupt is set up to make use of the FIFO. When using the FIFO, the hardware does the channel switching by itself. As much data as possible is read from the FIFO on each interrupt. |
| ISR_READ_ADCSNOFIFO = 11 | Like ISR_READ_ADCS but does not use the FIFO. |
| ISR_READ_ADCSFIFO = 12 | Like ISR_READ_ADCS but uses the FIFO even if the ADC channel group is not the interrupt source. Requires the ADC channel group to have a FIFO. |
| ISR_READ_ADCSASAP = 15 | Like ISR_READ_ADCS but if the ADC channel group has a FIFO and it is the source of the interrupt, the FIFO interrupt trigger level will be set to fill buffers as soon as possible. Note that ISR_READ_ADCS also does this if the buffer size is small (up to 128 samples for PCI230 and PCI260) or if the interrupt is set up in *non-continuous* mode. |
| ISR_WRITE_16COUNT = 39 | Writes the initial count value of a specified 16-bit counter (*Chip1*, *Chan1*). |
| ISR_WRITE_32COUNT = 40 | Writes the initial count values of two specified 16-bit counters (*Chip1*, *Chan1*, *Chip2*, *Chan2*). The upper 16 bits of the 32-bit data value is used to set the initial count value of the first specified counter. The lower 16 bits of the 32-bit data value is used to set the initial count value of the second specified counter. |
| ISR_WRITE_DATA8 = 32 | Writes an 8-bit I/O port (*Chip1*, *Chan1*). |
| ISR_WRITE_DATA16 = 33 | Writes two 8-bit I/O ports (*Chip1*, *Chan1*, *Chip2*, *Chan2*). The low 8 bits are written to the first I/O port and the next 8 bits to the second I/O port. |

| | |
|---|---|
| ISR_WRITE_PPIABC = 34 | Writes PPI ports A, B and C (*Chip1*, [*Chan1*=0]). The low 8 bits are written to port A, the next 8 bits to port B, and the next 8 to port C. |
| ISR_WRITE_2PPIABC = 41 | Like ISR_WRITE_PPIABC but writes to ports A, B and C of two specified PPI chips (*Chip1*, *Chip2*) on each interrupt. Consumes two data values on each interrupt. The first data value is written to ports A, B and C of the first specified PPI chip. The second data value is written to ports A, B and C of the second specified PPI chip. |
| ISR_WRITE_3PPIABC = 42 | Like ISR_WRITE_PPIABC but writes to ports A, B and C of three fixed PPI chips (PPIX, PPIY and PPIZ) on each interrupt. Consumes three data values on each interrupt. The first data value is written to ports A, B and C of the first PPI chip (PPIX). The second data value is written to ports A, B and C of the second PPI chip (PPIY). The third data value is written to ports A, B and C of the third PPI chip (PPIZ). |
| ISR_WRITE_DACS = 35 | Writes to a set of DAC channels, uncooking the data (*Group*, *ChanMask*). Consumes one data value per channel specified by the channel mask on each trigger. If the DAC channel group being written is the source of the interrupt and it has a FIFO, then the interrupt is set up to make use of the FIFO. When using the FIFO, as much data as possible is written to the FIFO on each interrupt and the hardware is responsible for clocking of data out to the specified set of DAC channels on each trigger. |
| ISR_WRITE_DACSNOFIFO = 37 | Like ISR_WRITE_DACS but does not use the FIFO. |
| ISR_WRITE_DACSFIFO = 38 | Like ISR_WRITE_DACS but uses the FIFO even when the DAC group being written to is not the interrupt source. Requires the DAC group to have a FIFO. |
| ISR_WRITE_2DACS = 36 | Writes to two DAC channels uncooking the data (*Group1*, *Chan1*, *Group2*, *Chan2*). The lower 16 bits are uncooked and written to the first DAC channel. The next 16 bits are uncooked and written to the second DAC channel. |

The driver supports the following interrupt source values for each card (these values are used for the *Chip* parameter of the user interrupt set-up function). The column labelled 'Multi' indicates whether the driver allows more than one interrupt source to be in use simultaneously.

| Card Type | Interrupt Sources | Multi |
|---|---|---|
| PC212E | PPIXC0=0, PPIXC3=4, Y1=8, Y2=12, Z1=16, Z2=20 | Yes |
| PC214E | PPIXC0=0, PPIXC3=4, PPIYC0=8, PPIYC3=12, Z1=16 | No |
| PC215E, PCI215 | PPIXC0=0, PPIXC3=4, PPIYC0=8, PPIYC3=12, Z1=16, Z2=20 | Yes |
| PC218E | X1=0, X2=4, Y1=8, Y2=12, Z1=16, Z2=20 | Yes |
| PC272E, PCI272 | PPIXC0=0, PPIXC3=4, PPIYC0=8, PPIYC3=12, PPIZC0=16, PPIZC3=20 | Yes |
| PC36AT, PCI236 | PPIX=0, PPIXC3=4 (both refer to same interrupt source) | No |
| PC263, PCI263 | N/A | N/A |
| PC24E, PC25E | Y1=8 | No |
| PC26AT | ADC0=0 | No |
| PC27E | ADC0=0 | No |
| PC30AT | ADC0=0, X2=4, PPIY=8, PPIYC7=8, 12 | No |

| | | |
|---|---|---|
| PCI224, PCI234 | EXT0=0, DAC2=8, Z2=20 | Yes |
| PCI230 | PPIXC0=0, PPIXC3=4, ADC2=8, SATRIG=12[1], DAC4=16[2], Z2=20 | Yes |
| PCI260 | 0, 4, ADC2=8, Z2=20 (0 and 4 are non-functional) | Yes |

[1] SATRIG is supported on PCI230+ and PCI260+.
[2] DAC4 is supported on PCI230+ hardware version 2.

The interrupt source constants above have the following meanings:

| | |
|---|---|
| X1, X2, Y1, Y2, Z1, Z2 | Interrupt source is the output of one of the counters of the timer / counter chip at this address offset (counter 1 on PC212E, PC214E, PC215E, PCI215, PC218E, PCI224, PCI230, PCI234 and PCI260; counter 0 on PC30AT; determined by jumpers on PC24E and PC25E). |
| PPIXC0, PPIXC3, PPIYC0, PPIYC3, PPIZC0, PPIZC3 | Interrupt source is port C bit 0 or port C bit 3 of the PPI chip at this address offset. |
| PPIYC7 | Interrupt source is port C bit 7 of the PPI chip at this address offset (PC30AT only). |
| PPIX | Interrupt source is one bit of port C of PPIX (bit 3 on PC36AT and PCI236). |
| PPIY | Interrupt source is one bit of port C of PPIY (bit 7 on PC30AT). |
| ADC0 | Interrupt source is A/D conversion complete (PC26AT, PC27E and PC30AT). |
| ADC2 | Interrupt source is A/D conversion complete or ADC FIFO at or above trigger level (PCI230 and PCI260). |
| DAC2 | Interrupt source is DAC FIFO at or below trigger level (PCI224 and PCI234). |
| DAC4 | Interrupt source is DAC FIFO at or below trigger level (PCI230+ hardware version 2). |
| EXT0 | Interrupt source is external trigger (PCI224 and PCI234). |
| SATRIG | Interrupt source is “ADC start acquisition trigger occurred” (PCI230+ and PCI260+). |

The PC214E and PC30AT cards have multiple possible interrupt sources (and so do the PC24E and PC25E in choice of counter output). The choice of interrupt source is selected by hardware jumpers.

Although the PC30AT can be connected to two interrupt lines, the driver supports at most one IRQ per card, so either J20 or J22 should be disconnected from J21.

#### 3.4.3.1 Basic Interrupt Callback

The *TCsetUserInterrupt* and *TCsetUserInterruptAIO* functions (see sections 6.4.13.1 and 6.4.13.2) allow a user-specified function to be called back every time a data value is read by the driver’s interrupt routine. The functions allow the user to specify the interrupt source, the type of data to be

read and any additional parameters required to specify the source of the data to be read (see table of data request types in section 3.4.3). These functions are currently only supported by the C/C++ and Delphi bindings supplied with the driver. For Visual Basic, the non-callback functions described in section 3.4.3.3 may be used instead.

The *TCsetUserInterrupt* function is mainly for digital resources; the following types of data may be requested:

- ISR_NODATA
- ISR_READ_16COUNT
- ISR_READ_16COUNTSTAT
- ISR_READ_32COUNT
- ISR_READ_PPIABC
- ISR_READ_PPIC
- ISR_PC27
- ISR_READ_DATA8
- ISR_READ_DATA16

The *TCsetUserInterruptAIO* function is for analogue resources; the following types of data may be requested:

- ISR_READ_ADCS
- ISR_READ_ADCSNOFIFO
- ISR_READ_ADCSFIFO
- ISR_READ_ADCSASAP

If the channel mask is set up to read more than one analogue channel, then the channels will be read in a cyclic sequence starting with the lowest. For example, if the channel mask is set to 5 ($101_2$), data will be read first from channel 0, then from channel 2, then cycling back to channel 0.

The functions also specify the callback function to be called and a user parameter which is passed on to this callback function when it is called. The user must supply a callback function of the form:

```
typedef VOID (CALLBACK *TTCCALLBACK)( short h
                                     , WPARAM wParam
                                     , ULONG lParam
                                     );
```

The *wParam* parameter is the user parameter specified in the *TCsetUserInterrupt* or *TCsetUserInterruptAIO* function call. It can be any user value, a pointer to a user's data structure, a user code etc. The *lParam* contains the data the read by the driver. See section 6.4.13.4 for a detailed description of the callback function.

#### 3.4.3.2 Transferring Buffers Under Interrupt Control

The basic interrupt callback described in section 3.4.3.1 only allows data to be read from the card on interrupt and only passes one value at a time to the user interrupt callback function. In addition, each data value is passed from the kernel driver level to the user level in individual messages, which causes a large system overhead at high data rates). To deal with these problems, the buffered user interrupt functions described in this section may be used. ). These functions are currently only supported by the C/C++ and Delphi bindings supplied with the driver. For Visual Basic, the non-callback functions described in section 3.4.3.3 may be used instead.

The *TCsetBufferUserInterrupt* and *TCsetBufferUserInterruptAIO* functions (see sections 6.4.14.1 and 6.4.14.2) allow a user to be called back and to read or write large buffers of information to the driver. The callback function is only called when a bufferful of data has been read or another bufferful of data is required to be written. The functions allow the user to specify the interrupt

source, the type of data transfer required and any other parameters required to specify where the data is to be read from or written to (see table of data request types in section 3.4.3).

The *TCsetBufferUserInterrupt* function is mainly for digital resources; the following types of data transfer may be requested:

- ISR_NODATA
- ISR_READ_16COUNT
- ISR_READ_16COUNTSTAT
- ISR_READ_32COUNT
- ISR_READ_32COUNTSTAT
- ISR_READ_PPIABC
- ISR_READ_2PPIABC
- ISR_READ_3PPIABC
- ISR_READ_PPIC
- ISR_PC27
- ISR_READ_DATA8
- ISR_READ_DATA16
- ISR_WRITE_16COUNT
- ISR_WRITE_32COUNT
- ISR_WRITE_DATA8
- ISR_WRITE_DATA16
- ISR_WRITE_PPIABC
- ISR_WRITE_2PPIABC
- ISR_WRITE_3PPIABC

The *TCsetBufferUserInterruptAIO* function is used for most analogue resources; the following types of data transfer may be requested:

- ISR_READ_ADCS
- ISR_READ_ADCSNOFIFO
- ISR_READ_ADCSFIFO
- ISR_READ_ADCSASAP
- ISR_WRITE_DACS
- ISR_WRITE_DACSNOFIFO
- ISR_WRITE_DACSFIFO

In addition, there is a *TCsetBufferUserInterrupt2* function (see section 6.4.14.3) which supports the following types of data transfer:

- ISR_WRITE_2DACS

The functions also allow the user to specify the length of buffers required. The minimum length allowed is 1, but for the supplied DIO_TC.DLL code prior to AMPDIO v4.23 there is a bug which stops buffer lengths less than 2 working correctly. The buffers are created by the DLL.

The functions also allow the user to specify whether double buffering or single buffering is to be used. If double buffering is being used, the DIO_TC.DLL and the driver try to keep up a continuous flow of data between the user and the driver — the driver can be dealing with one buffer while the user callback function is dealing with the other. This mode is used when the *fContinuous* parameter is set to TRUE in the interrupt set-up function. If *fContinuous* is set to FALSE, single buffering is used and there is no attempt to maintain a continuous flow of data between buffers.

The functions also specify the callback function to be called and a user parameter that is passed on to this callback function when it is called. The user must supply a callback function of the form:

```
typedef VOID (CALLBACK *TTCBCALLBACK)( short h
                                     , WPARAM wParam
                                     , ULONG sizeofBuffer
```

```
                                    , PULONG pBuffer
                                    );
```

The *wParam* parameter is the user parameter specified in the interrupt set-up function call. It can be any user value, a pointer to a user's data structure, a user code etc. The *sizeofBuffer* parameter is the length of the buffer being passed to the function. This is generally the buffer length passed to the interrupt set-up function. It is possible for *sizeofBuffer* to be 0 if an error has occurred. The *pBuffer* parameter points to the buffer in memory. In Delphi, it may be treated as a pointer to an array. See section 6.4.14.4 for a detailed description of the callback function.

If the buffer contains data for multiple analogue channels, it may be useful to treat the buffer pointed to by the *pBuffer* parameter as a 2-dimensional array with the size of the inner dimension being the number of channels selected by the *ChanMask* parameter passed to the *TCsetBufferUserInterruptAIO* interrupt set-up function. The buffer size specified should be a multiple of the number of selected channels so that the data in each buffer starts on the same channel.

For example, if the *ChanMask* is set to 5 ($101_2$), then two channels (0 and 2) are selected. If the buffer is treated as a 1-dimensional array (starting at index 0) then:

- Array element [0] is the first data element for channel 0.
- Array element [1] is the first data element for channel 2.
- Array element [2] is the second data element for channel 0.
- Array element [3] is the second data element for channel 2.

If the above buffer is treated as a 2-dimensional array with an inner dimension of size 2 (starting at index 0, 0), then:

- Array element [0][0] is the first data element for channel 0.
- Array element [0][1] is the first data element for channel 2.
- Array element [1][0] is the second data element for channel 0.
- Array element [1][1] is the second data element for channel 2.

The above assume that the buffer size is a multiple of 2. Array syntax varies with language — The above is based on the 'C' language.

The same construct may be used for the other interrupt data transfer types that yield or consume more than one 32-bit data value on each interrupt, such as ISR_READ_2PPIABC and ISR_WRITE_2PPIABC.

#### 3.4.3.2.1 Acquiring AC Analogue Signals

By using this mechanism, the user can acquire AC analogue signals. In the "SCOPE" example, the user function instructs the driver to acquire two channels of analogue data into a buffer. This buffer is then formatted on a graph resembling an oscilloscope. It is possible to sample at a total rate of 312500 samples per second for short periods on the original PCI230 and PCI260 cards. For the new PCI230+ and PCI260+ cards, the total maximum rate is reduced to 250000 samples per second. This is the maximum total sample rate for all channels combined. The sample rate for the PCI230 and PCI260 cards is independent of machine performance, however this is not true of the supported ISA cards.

##### 3.4.3.2.1.1 Controlling Timing for Reading Multiple Analogue Channels

When reading multiple analogue channels, the channels are read sequentially, but only one channel is read per conversion trigger. If a rate generator is set up on a timer/counter channel acting as a conversion trigger, then to sample *N* channels at a frequency *f*, the rate generator can be set to generate a frequency of ($N \times f$). This will result in the channels being read interleaved

across equal periods of time, i.e. the time between sampling the first channel and the next channel is the same as the time between sampling the last channel and the first channel of the next cycle.

For some applications it is desirable to read all the desired channels in a short period, then have a long gap before starting the next reading cycle. On the PCI230 and PCI260, this may be accomplished by using the periodic pulse train generation functions introduced in version 4.32 of the library. The more general of these functions, *TCsetPeriodicPulseTrain* (see section 6.4.8.7) allows the number of pulses in each pulse train, the duration of each pulse train and the frequency of pulse trains to be set. Normally, the number of pulses in each train would be set to the number of channels being read.

If the set of channels is to be sampled at a fairly low frequency, the buffer length specified in the call to the user interrupt set-up function can also be set to the number of channels, then the user interrupt callback function will be called once for each set of readings. The user interrupt set-up function should be set to use 'continuous' mode.

There is a restricted version of the periodic pulse train generator provided by the *TCsetRestrictedPulseTrain* function (see section 6.4.8.14) which is similar but provides much less control over the duration of the pulse train, typically a duration of 1 ms or 100 µs would be used for the PCI230 or PCI260 and all the desired channels would be sampled over this period.

It is also possible to read a set of channels on a hardware trigger using the TCsetOneShotPulseTrain function (see section 6.4.8.20), rather than read them periodically. This trigger could be provided by the output of another timer/counter channel rather than an external trigger signal. This is effectively how the *TCsetPeriodicPulseTrain* function is implemented. Another possibility would be to use a timer/counter to implement a delay after an external hardware trigger using timer/counter mode 5 (hardware triggered strobe).

On previous versions of the PCI230 and PCI260 cards there is a problem using the controlled timing mechanisms. This is fixed in hardware on PCI230 cards bearing a sticker labelled PR989386.4 on chip U19, and on PCI260 cards bearing a sticker labelled PR989385.4 on chip U19. A software workaround can be used with caution on older cards in some circumstances.

The problem on older versions of the cards is due to the possibility of an ADC conversion being triggered by a change to the configured value of the ADC conversion trigger source (e.g. by the function *AIOsetADCconvSource* — see section 6.4.19.1). If the old conversion trigger source was at a low logic level and the new conversion trigger source is at a high logic level and ADC conversion is triggered. (N.B. the conversion trigger source CNV_NONE is always at a low-logic level; the conversion trigger source CNV_SW is at a low logic level except at the time of the actual software trigger.) This is a problem when setting the conversion trigger source to one of the timer channels (CNV_CT0, CNV_CT1 or CNV_CT2) as a conversion trigger may or may not occur at this time (depending on the state of the timer's output). This does not affect the correspondence between buffer positions and channels but does affect the position of the long gap within the sequence of channel readings.

The bogus ADC trigger can also occur at the start of each user interrupt data buffer if single buffering (as opposed to double buffering) is used. In single buffer mode, the driver resets the ADC FIFO and this process involves temporarily setting the conversion trigger source to CNV_NONE. A bogus trigger can occur when the conversion trigger source is reset to the configured value.

As a software workaround, there is a function *TCflushUserInterrupt* (see section 6.4.16.2) that may be called to reset the ADC FIFO (amongst other things) to work around the problem. This function should be used with caution as it uses a partial FIFO reset sequence (it does not set the conversion trigger source to CNV_NONE) that can cause the correspondence between ADC channels and buffer positions to be lost. To avoid this, the caller must use some means to ensure that an ADC conversion cannot be triggered during this function call, and it must do that without reconfiguring the conversion trigger source temporarily before the function call. This workaround is particularly tricky for single buffer mode interrupts due to the possibility of the bogus trigger

occurring at the start of every buffer. It is recommended that the hardware fix is applied to avoid the problem altogether.

#### 3.4.3.2.1.2 Controlling Start of Aquisition on PCI230+ and PCI260+

For the new PCI230+ and PCI260+ cards, it is possible to delay the start of acquisition until a programmable start acquisition trigger occurs. Alternatively, data can be acquired as normal while keeping a count of the number of samples stored until the start acquisition trigger occurs. If the card is set to delay the start of acquisition, data acquired from each channel will be placed into a temporary buffer instead of the FIFO, and then either discarded (if the trigger has not occurred yet) or transferred to the FIFO (if the trigger has occurred). Once the start acquisition trigger has occurred, subsequent channel readings will be placed into the FIFO as normal.

The function *AIOsetADCstartAcquisitionTrigger* (see section 6.4.19.5) is used to set the start acquisition trigger. This is supported since version 4.42 of the AMPDIO driver and DIO_TC.DLL. The following start acquisition trigger types are supported by PCI230+ and PCI260+ cards:

| | |
|---|---|
| TRIG_NOW | Trigger immediately. |
| TRIG_NEVER | Never trigger. |
| TRIG_EXT_LTOH | Trigger on external digital low-high transition. |
| TRIG_EXT_HTOL | Trigger on external digital high-low transition. |
| TRIG_EXT_LOW | Trigger when external digital signal is low. |
| TRIG_EXT_HIGH | Trigger when external digital signal is high. |
| TRIG_ANA_LTOH | Trigger on analogue low-high transition. |
| TRIG_ANA_HTOL | Trigger on analogue high-low transition. |
| TRIG_ANA_LOW | Trigger when analogue value is low. |
| TRIG_ANA_HIGH | Trigger when analogue value is high. |

For start acquisition triggers involving an analogue value, the channel, threshold, hysteresis and a hold-off value are specified in the function call. (The hysteresis is applied for ‘TRIG_ANA_LTOH’ and ‘TRIG_ANA_HTOL’.) The specified channel should be one of those being acquired by the user interrupt, as no other channels will be sampled. It should also be noted that analogue levels and transitions are only detected when it is the specified channel’s turn to be sampled.

If the user interrupt is set up to use ‘non-continuous’ mode, the start acquisition trigger is applied at the start of every buffer received by the driver from the DLL. In ‘continuous’ mode, the start acquisition trigger is applied only to the first buffer received by the driver from the DLL.

The following start acquisition start types are supported by PCI230+ and PCI260+ cards:

| | |
|---|---|
| START_NOW | Start acquisition as soon as possible (pre-trigger mode). |
| START_TRIG | Start acquisition when trigger occurs. |

When set to ‘START_NOW’, the trigger is not applied until the number of samples specified by the hold-off value have been aqquired; data acquired before the trigger occurs is stored in the FIFO (and read out of the FIFO) as normal. When set to ‘START_TRIG’, the hold-off value must be zero; data acquired before the trigger occurs is discarded by the hardware.

The function *AIOgetADCpretriggerCount* (see section 6.4.19.6) can be used to determine whether the start acquisition trigger has occurred. When the start acquisition type is set to 'START_NOW' it also indicates the number of samples that were acquired before the trigger occurred (a pre-trigger count).

For the original PCI230 and PCI260 and other supported ADC cards, the only supported start acquisition trigger source is 'TRIG_NOW' and the only supported start type is 'START_NOW'. For backward compatibility with older software, the driver resets the start acquisition trigger source to 'TRIG_NOW' and the start type to 'START_NOW' whenever a board handle is obtained using the *registerBoardEx* or equivalent function.

For the PCI230+ and PCI260+, the specified start acquisition trigger is only used if the ADC FIFO is used. It will not be used if the user interrupt is set-up with data type set to 'ISR_READ_ADCSNOFIFO'.

#### 3.4.3.2.2 Playing AC Analogue Signals

The same mechanism can be used to play analogue signals. In the signal generator example, the user function copies a known pattern into a buffer. The buffer is then written to the digital to analogue converters under driver control. By using this mechanism, it is possible to play a pattern out on a PC24E DAC card at a sample rate of up to 25 kHz using a typical P166 machine. Note that the rate at which the driver can produce a signal deterministically is related to machine performance and may be impinged by other driver software loaded on to the machine.

### 3.4.3.3 Using Interrupts Without Callbacks

The basic user interrupt callback mechanism described in section 3.4.3.1 and the buffered user interrupt callback mechanism described in section 3.4.3.2 both result in the creation of a thread within the DLL which is responsible for transferring data between the driver and the callback function. In some programming environments, this is impossible to handle. They are currently only supported by C/C++ and Delphi bindings supplied with the driver.

Since AMPDIO v4.00, the DLL supports another mechanism for setting up buffered user interrupts which does not involve callbacks or creation of extra threads. It allows the main thread to poll the interface to see if data is ready (similar to the way the Event Recorder mechanism is used) or to wait until data is ready with an optional timeout (a poll is just a wait with a timeout of 0). These functions were originally written to support HP Vee applications, but can be used in other programming environments.

The functions to set-up the buffered user interrupts without callbacks are *TCsetNCBufferUserInterrupt*, *TCsetNCBufferUserInterruptAIO* and *TCsetNCBufferUserInterrupt2* (see sections 6.4.15.1, 6.4.15.2 and 6.4.15.3). These are called similarly to the *TCsetBufferUserInterrupt*, *TCsetBufferUserInterruptAIO* and *TCsetBufferUserInterrupt2* functions to set up the interrupt source, the type of data transfer, the data source or destination on the card, the length of the buffers and whether single or double buffering is to be used (see section 3.4.3.2), but do not have parameters for passing a callback function or a user parameter.

Once interrupts have been enabled by calling *enableInterrupts*, the user program can use the *TCdriveNCBufferUserInterrupt* function (see section 6.4.15.4) to copy data from one of the interrupt data buffers into a user array (when reading data from the card) or from a user array into an interrupt data buffer (when writing data to the card). A whole buffer length of data is copied in each case.

The *TCdriveNCBufferUserInterrupt* function will perform a blocking wait if necessary until it is possible to transfer the data. It is useful to be able to poll to see whether or not the *TCdriveNCBufferUserInterrupt* function will wait, so that the program can go and do something else for a while instead. Since AMPDIO v4.02, the *TCwaitNCBufferReady* function (see section

6.4.15.5) has been available, which allows the caller to tell whether or not *TCdriveNCBufferUserInterrupt* would block. The *TCwaitNCBufferReady* function has a timeout parameter that allows it to wait until either a specified maximum number of milliseconds have elapsed (rounded up to a number of system clock ticks), or it is known that *TCdriveNCBufferUserInterrupt* may be called without blocking. The return value indicates which of these conditions is the case. Specifying a timeout of 0 milliseconds results in a simple poll. The timeout can be set to INFINITE to stop it expiring, but you might as well not bother calling the *TCwaitNCBufferReady* function in that case.

If multiple non-callback user interrupts are to be handled, The *TCwaitNCBufferReady* function may be used with a timeout of zero with each user interrupt in turn to see which ones require attention. In AMPDIO v4.20, the *TCwaitMultiNCBufferReady* function was introduced (see section 6.4.15.6). This function allows you to wait on or poll multiple non-callback user interrupts simultaneously. It is quite tricky to set up, requiring two input arrays which between them hold a board handle and user interrupt handle pairing at each index of the array. Another parameter indicates the number of user interrupts being handled. There is also a timeout parameter to indicate the maximum number of milliseconds to wait, which may be 0, some number or INFINITE. The function returns indicating whether the timeout expired or *TCdriveNCBufferUserInterrupt* can be called without blocking for one of the specified user interrupts. Two other parameters are used by reference to return the board handle and user interrupt handle of such a user interrupt.

When using the non-callback user interrupt handling to read data values from the driver (ISR_READ_... data requests), the following should be noted. Once the user interrupt has been set up and enabled, then if the *fContinuous* parameter was set to FALSE when the user interrupt was set up, no buffer is sent to the driver to be filled in until the first call to *TCdriveNCBufferUserInterrupt*, *TCwaitNCBufferReady* or *TCwaitMultiNCBufferReady* for this user interrupt. If the *fContinuous* parameter was set to TRUE for continuous double-buffered operation, then only one buffer is sent to the driver to be filled in when the user interrupt is enabled and the other buffer is sent on the first call to *TCdriveNCBufferUserInterrupt*, *TCwaitNCBufferReady* or *TCwaitMultiNCBufferReady*.

The driver will not enable the interrupt until it receives the first buffer to be filled in. For some applications, it is necessary to ensure that the driver has enabled the interrupt and has a buffer to fill in before some other initialization is performed that allows interrupts to be generated. This can be done with a call to *TCwaitNCBufferReady* or *TCwaitMultiNCBufferReady* with a timeout of 0 before this other initialization is performed (but after the call to *enableInterrupts*).

# 4 SOFTWARE INSTALLED WITH THE DRIVER

## 4.1 Installed Software

The self-extracting executable installs the following software into the target directory.

| Sub-Directory | |
|---|---|
| \DIO_CODE | Source code and documentation for DIO_TC.DLL API library. |
| \EX_VB | Visual Basic examples (VB 5.0 onwards) |
| \EX_DELPH | Delphi examples (Delphi 3.0 onwards) |
| \EX_VEE | Agilent Vee Pro / Hewlett Packard HP VEE examples |
| \EX_C | Win32 console examples in C |
| \EX_VBNET | Visual Basic .NET examples |
| \EX_C# | Visual C# .NET examples |
| \SYS_DLLs | System DLLs (not present for AMPDIO v5.00 onwards) |

The software includes examples as both runtime and source code. The runtime examples can be used to exercise and become familiar with the hardware. The source code serves two purposes. Firstly, it can be used as a source of reference to see how the DLL functions are used. In addition, it provides a starting point for anyone who wishes to write software with similar functionality.

## 4.2 Visual Basic Examples

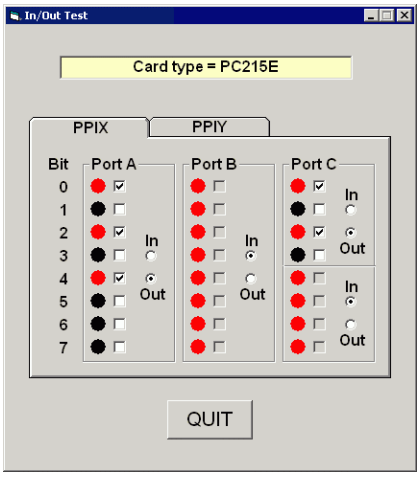
The sub directory EX_VB contains the executables and source code for a number of example applications written in Microsoft Visual Basic 5.0, service pack 2. AMPDIO v5.00 switched to using Visual Basic 6.0 to build the executables, but the project files can still be loaded in VB 5.0 if the warnings about invalid key ‘Retained’ are ignored. AMPDIO v5.00 installs the VB 6.0 run-time support files automatically. Earlier versions may require the VB 5.0 run-time support files to be installed manually — see section 4.9 or follow the instructions in the ‘README_DLL.TXT’ file found in the ‘SYS_DLLS’ directory (not present in AMPDIO 5.00 and later).

### 4.2.1 Digital IO — INOUT.EXE

The “IN OUT” example demonstrates basic Digital IO using the 82C55 peripheral port interface. It runs on all supported Amplicon cards that have PPI resources, PC212E, PC214E, PC215E, PC263, PC272E, PC30AT, PC36AT, PC36LP, PCI215, PCI230, PCI236, PCI263 and PCI272.

It allows the user to select PPI ports as inputs or outputs, set outputs high or low and monitor their actual state.

It does not use interrupts.




### 4.2.2 Timer — BASICTMR.EXE

The “Basic Timer” example demonstrates setting up and using the 82C54 compatible timers on Series 200 Digital IO Cards. It runs on the PC212E, PC214E, PC215E, PC218E, PC24E/PC25E, PC26AT, PC27E, PC30AT, PCI215, PCI224, PCI230, PCI234, and PCI260.

It allows the available timer resources to be programmed in a variety of modes (rate, single shot etc.) and frequencies.

It does not use interrupts.

### 4.2.3 Frequency Multiplier — FREQMULT.EXE

The “Frequency Multiplier” example demonstrates using the *TCmultiplyFreq* function on Series 200 Digital IO Cards. It runs on the PC212E, PC214E, PC215E, PC218E and PCI215 digital counter timer cards.

It measures a frequency using one set of timer channels and produces a multiplied version of that frequency on another.

It does not use interrupts

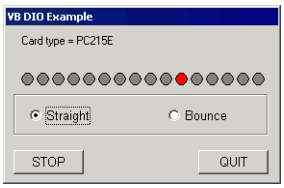
### 4.2.4 Event Recorder — EVENTREC.EXE

The “Event Recorder” example demonstrates the use of the “TCsetEventRecorder” function on Series 200 Digital IO Cards. It runs on the PC212E, PC214E, PC215E, PCI215 and PCI230.

It starts a millisecond timer and records when PPI X Bit C3 interrupt occurs.

It requires that the cards be installed with interrupts.

### 4.2.5 Digital IO With Interrupts — DIO_EX.EXE

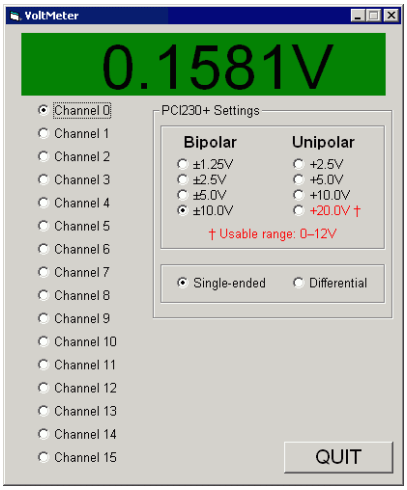



The “Extended DIO” example demonstrates sending buffers of information to the 82C55 PPI interface under interrupt control. A strobe pattern is played out the PPI port. It runs on the PC212E, PC214E, PC215E, PC30AT, PCI215 and PCI230 cards.

It requires that the cards be installed with interrupts.

Versions of this example supplied with AMPDIO v4.31 or earlier cannot be compiled and run under Visual Basic 6.0.

### 4.2.6 Voltmeter — METER.EXE



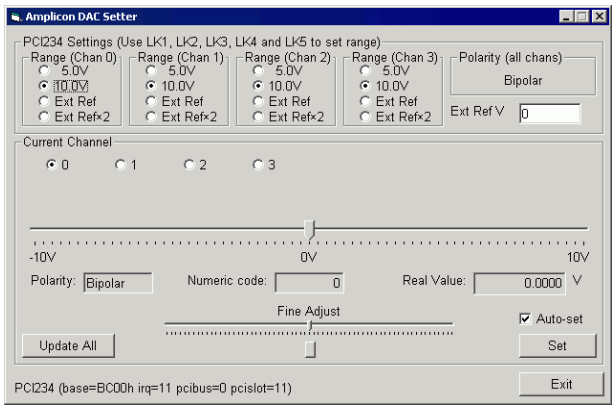

The “Meter” example is a multi-channel voltage meter example. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

It does not use interrupts.

Versions of this example supplied with AMPDIO v4.31 and earlier required the card to be installed with interrupts. Because of the style of user interrupt set-up functions used, these versions could not be compiled and run under Visual Basic 6.0.

### 4.2.7 D-to-A Converter — DACSET.EXE

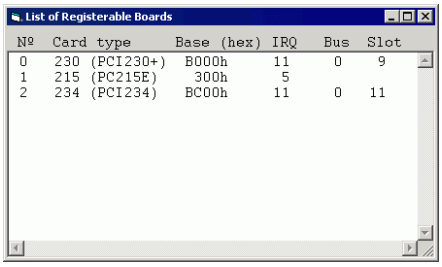
The “DACSet” example uses a slider to write values to a DAC channel. It allows the jumper settings in use to be set and the channel to be chosen and displays the numeric values written to the driver and the real value of the output in volts or milliamps. It may be of use for calibration of the DACs. It runs on the PC24E/PC25E, PC30AT, PCI224, PCI230 and PCI234. It requires Visual Basic 5.0 SP2 or higher.




It does not use interrupts.

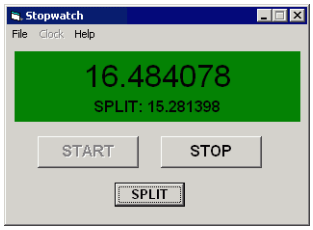
### 4.2.8 Registerable Board Lister — REGBOARD.EXE

The “RegBoard” example attempts to register each board in turn using the *registerBoardEx* function (see section 6.4.1.2) and lists details of the boards it managed to register using information provided by the *GetBoardModel*, *GetBoardBase*, *GetBoardIRQ* and *GetBoardPciPosition* functions (see sections 6.4.1.4, 6.4.1.5, 6.4.1.6 and 6.4.1.7).




### 4.2.9 Stopwatch — STOPWATCH.EXE

The “Stopwatch” example source code illustrates the setting up and reading of a cascaded pair of 82C54-compatible timer channels using the *TCsetClock*, *TCsetGate*, *TCsetMode*, *TCsetCount*, *TCgetStatus* and *TCgetCounts* functions (see sections 6.4.5.3, 6.4.5.6, 6.4.5.9, 6.4.5.12, 6.4.5.10 and 6.4.5.16).

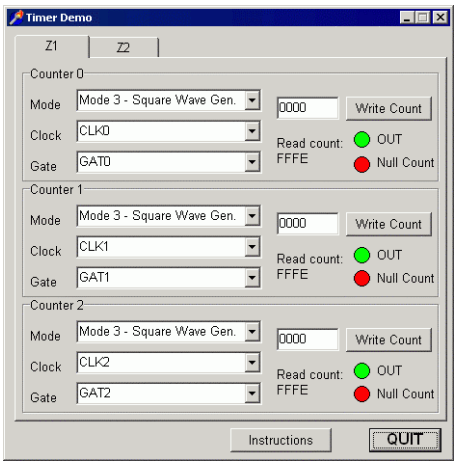



## 4.3 Delphi Examples

The sub directory EX_DELPH contains the executables and source code for a number of example applications written in Borland Delphi 3.0. The source code can be recompiled for newer versions of Delphi.

### 4.3.1 Timer — TIMER.EXE

The “Timer” example demonstrates setting up and using the 82C54 compatible timers on Series 200 Digital IO Cards. It runs on the PC212E, PC214E, PC215E, PC218E, PC24E/PC25E, PC26AT, PC27E, PC30AT, PCI215, PCI224, PCI230, PCI234 and PCI260.




It allows the available timer resources to be programmed in a variety of modes (rate, single shot etc.) and frequencies.

It is similar in operation to the Visual Basic “BASICTMR” example.

It does not use interrupts.

### 4.3.2 Digital IO — INOUT.EXE

The “IN OUT” example demonstrates basic Digital IO using the 82C55 peripheral port interface. It runs on all supported Amplicon cards that have PPI resources, PC212E, PC214E, PC215E, PC263, PC272E, PC30AT, PC36AT, PC36LP, PCI215, PCI230, PCI236, PCI263 and PCI272.

It allows the user to select PPI ports as inputs or outputs, set outputs high or low and monitor their actual state.

It is equivalent to the Visual Basic “INOUT” example.

It does not use interrupts.

### 4.3.3 Digital IO With Interrupts — PDIO_EX.EXE

The “Extended DIO” example demonstrates sending buffers of information to the 82C55 PPI interface under interrupt control. A strobe pattern is played out the PPI port. It runs on the PC212E, PC214E, PC215E, PC30AT, PCI215 and PCI230 cards. It is equivalent to the Visual Basic “DIO_EX” example.

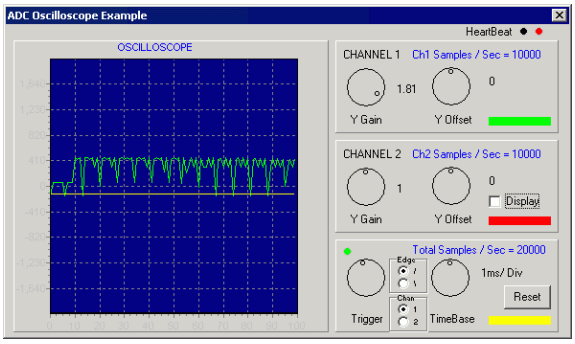
It requires that the cards be installed with interrupts.

### 4.3.4 Voltmeter — METER.EXE

The “Meter” example is a multi-channel voltage meter example. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

Unlike the Visual Basic “METER” example, this one uses interrupts.

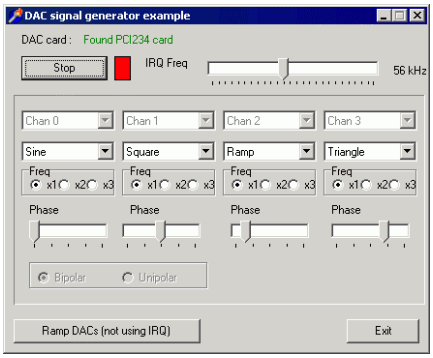
### 4.3.5 Oscilloscope — OSSCOPE.EXE




The “Scope” example demonstrates a simple low frequency two-channel oscilloscope function with adjustable trigger level. Depending on computer performance, it can be configured with a sample frequency of up to 25 kHz on the PC30AT card. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

It requires that the cards be installed with interrupts.

### 4.3.6 Signal Generator — SIGGEN.EXE




The “Siggen” example is a simple signal generator that demonstrates sending data buffers to DAC converters under interrupt control. It can generate sine, square, triangle and pulse waveforms on up to 4 DAC channels. It runs on the PC24E, PC25E, PC30AT, PCI224 PCI230 and PCI234 cards. Care must be taken to set up any card

jumpers correctly.

It requires that the cards be installed with interrupts.

## 4.4 Agilent VEE Pro / Hewlett Packard HP VEE Examples

The sub directory EX_VEE contains the VEE examples written in VEE 4.0 and VEE 5.0. It also contains “dio_vee.h” the header file that allows DIO_TC.DLL to be imported into VEE and “dio_tc_lib.vee” which contains a number of VEE helper functions.

### 4.4.1 ADC Test — ADCTEST.VEE

The ADC test example implements a simple oscilloscope in VEE. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

It requires that the cards be installed with interrupts.

### 4.4.2 DAC Test — DACTEST.VEE

The DAC test example implements a simple signal generator function in VEE. It runs on the PC24E, PC25E, PC30AT, PCI224 PCI230 and PCI234 cards. Care must be taken to set up any card jumpers correctly.

It requires that the cards be installed with interrupts.

### 4.4.3 Digital Input — DIGINPUT.VEE

The Digital Input example demonstrates simple digital IO in VEE. It runs on the series 200 digital IO / counter timer cards, PC212E, PC214E, PC215E, PC272E, PC36AT, PCI215 and PCI236. It does not use interrupts.

### 4.4.4 Timer Demo — TIMERDEM.VEE

The Timer Demo example demonstrates using timers in VEE. It runs on the series 200 digital IO / counter timer cards, PC212E, PC214E, PC215E, PC218E and PCI215.

It does not use interrupts.

## 4.5 Win32 Console Examples in C

The sub-directory EX_C contains source code and executables for Win32 console (character mode) examples written in C.

The examples have been developed with Microsoft Visual C++ 4.2, using the C language (rather than C++). They run in the Win32 console (DOS box) and make use of standard C library functions and Win32 library functions, in addition to the Amplicon DIO_TC.DLL library functions.

The .MDP files are project files for Microsoft Visual C++ 4.0 onwards. The .DSW files are project files for Microsoft Visual C++ 6.0 onwards. The .vcproj files are project files for Microsoft Visual Studio 2005 onwards. As supplied, they define '..\DIO_CODE' as an additional include directory (under Build -> Settings -> C/C++ -> Category: Preprocessor) and add ..\DIO_CODE\dio_tc.lib to the list of Object/library modules (under Build -> Settings -> Link). For AMPDIO v5.00 and later, dio_tc.lib has been moved to ..\DIO_CODE\Win32 for the 32-bit “Win32” build and to

..\DIO_CODE\x64 for the “x64” build. If things are moved around, then the project settings need to be adjusted accordingly (in which case it is probably easiest to copy the adioctl.h, dio_tc.h and dio_tc.lib into the same directory as the rest of the application files).

For later versions of Microsoft Visual C++, the project files can be automatically converted to use the new version when they are opened. It is possible to build the examples using the free “Express” edition of Visual C++, but it will also be necessary to obtain and install “Microsoft Platform SDK” and configure the directories in the “Projects and Solutions” section in the “Options” dialog in Visual C++ Express. In that section the following paths should be added to the appropriate subsections:

- Executable files: C:\Program Files\Microsoft SDK\Bin
- Include files: C:\Program Files\Microsoft SDK\include
- Library files: C:\Program Files\Microsoft SDK\Lib

(The above paths need to be changed if the Platform SDK has been installed somewhere else.) These options are global, not part of a specific project, so they only need to be configured once.

The following header files from the ‘..\DIO_CODE’ directory are common to all the applications:

- adioctl.h
- dio_tc.h

In addition, the standard C header files and the Win32 header files are required, but should already be in the standard include path.

For Microsoft Visual C++, the applications are linked to ‘..\DIO_CODE\dio_tc.lib’, which is a stub library for DIO_TC.DLL. Note that ‘..\DIO_CODE\dio_tc.lib’ is not compatible with other non-Microsoft compilers such as Borland C++ builder, but it is relatively straightforward to create a ‘dio_tc.lib’ file compatible with other compilers from the supplied ‘DIO_TC.DLL’ or ‘..\DIO_CODE\dio_tc.def’ files, e.g. using the IMPLIB utility for Borland C.

In addition to ‘dio_tc.lib’, the standard C libraries and Win32 libraries are linked to.

See the README.TXT file in the EX_C sub-directory for more information about the examples.

### 4.5.1 Capture Analogue Input to Comma-Separated Variables (CSV) or Binary File

This example demonstrates timed capture of analogue input data to a comma-separated variable (CSV) file, which may then be opened in a spreadsheet program such as Microsoft Excel. It also supports capturing to a raw binary file. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards.

It allows the user to choose a supported card from a list, then asks a series of questions about the capture progress. Default answers in square brackets can be selected by pressing the carriage return key. All user input to the program is from standard input so answers to questions could be provided by redirecting standard input from a file.

Once the user has set up the capture process, it is started and captured input data is converted to ASCII CSV format and written to the output file, either as raw numbers from the driver or as voltage values.

The example uses interrupts. See the README.TXT file in the EX_C sub-directory for more details.

## 4.6 Visual Basic .NET Examples

The subdirectory EX_VBNET contains executables and source code for example programs written in Microsoft Visual Basic .NET. In order to run the examples the Microsoft .NET Framework version 1.1 or later is required. If not already installed, a suitable version may be downloaded using the Windows Update service or the Microsoft Download Center.

The project files (*.vbproj) have been created with Microsoft Visual Studio .NET 2003. If a later version of Visual Studio is used, including the free “Express” edition of Visual Basic, the project files can be automatically converted to use the later version when they are opened.

For developers, the Visual Basic .NET bindings for DIO_TC.DLL are in the DIO_TC.VB file.

### 4.6.1 Digital IO — InOut_VBNET.exe

The “IN OUT” example demonstrates basic Digital IO using the 82C55 peripheral port interface. It runs on all supported Amplicon cards that have PPI resources, PC212E, PC214E, PC215E, PC263, PC272E, PC30AT, PC36AT, PC36LP, PCI215, PCI230, PCI236, PCI263 and PCI272.

It allows the user to select PPI ports as inputs or outputs, set outputs high or low and monitor their actual state.

It is equivalent to the Visual Basic 5.0 “INOUT” example.

It does not use interrupts.

### 4.6.2 Digital IO With Interrupts — DIO_EX_VBNET.exe and DIO_EX2_VBNET.exe

The “Extended DIO” example demonstrates sending buffers of information to the 82C55 PPI interface under interrupt control. A strobe pattern is played out the PPI port. It runs on the PC212E, PC214E, PC215E, PC30AT, PCI215 and PCI230 cards. It is equivalent to the Visual Basic 5.0 “DIO_EX” example.

It requires that the cards be installed with interrupts.

DIO_EX_VBNET.exe uses non-callback mode user interrupt functions like the Visual Basic 5.0 “DIO_EX” example. DIO_EX2_VBNET.exe uses “delegates” for user interrupt callback functions.

### 4.6.3 Voltmeter — Meter_VBNET.exe

The “Meter” example is a multi-channel voltage meter example. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

Unlike the Visual Basic “METER” example, this one uses interrupts. It uses a “delegate” to set up the user interrupt callback function.

## 4.7 Visual C# .NET Examples

The subdirectory EX_C# contains executables and source code for example programs written in Microsoft Visual C# .NET. In order to run the examples the Microsoft .NET Framework version 1.1 or later is required. If not already installed, a suitable version may be downloaded using the Windows Update service or the Microsoft Download Center.

The project files (*.csproj) have been created with Microsoft Visual Studio .NET 2003. If a later version of Visual Studio is used, including the free “Express” edition of Visual C#, the project files can be automatically converted to use the later version when they are opened.

For developers, the Visual C# .NET bindings for DIO_TC.DLL are in the Dioc_tc_h.CS file.

### 4.7.1 Digital IO — InOut_CSHARP.exe

The “IN OUT” example demonstrates basic Digital IO using the 82C55 peripheral port interface. It runs on all supported Amplicon cards that have PPI resources, PC212E, PC214E, PC215E, PC263, PC272E, PC30AT, PC36AT, PC36LP, PCI215, PCI230, PCI236, PCI263 and PCI272.

It allows the user to select PPI ports as inputs or outputs, set outputs high or low and monitor their actual state.

It is equivalent to the Visual Basic 5.0 “INOUT” example.

It does not use interrupts.

### 4.7.2 Digital IO With Interrupts — DIO_EX_CSHARP.exe and DIO_EX2_CSHARP.exe

The “Extended DIO” example demonstrates sending buffers of information to the 82C55 PPI interface under interrupt control. A strobe pattern is played out the PPI port. It runs on the PC212E, PC214E, PC215E, PC30AT, PCI215 and PCI230 cards. It is equivalent to the Visual Basic 5.0 “DIO_EX” example.

It requires that the cards be installed with interrupts.

DIO_EX_CSHARP.exe uses non-callback mode user interrupt functions like the Visual Basic 5.0 “DIO_EX” example. DIO_EX2_CSHARP.exe uses “delegates” for user interrupt callback functions.

### 4.7.3 Voltmeter — Meter_CSHARP.exe

The “Meter” example is a multi-channel voltage meter example. It runs on the PC26AT, PC27E, PC30AT, PCI230 and PCI260 cards. Care must be taken to set up any card jumpers correctly.

Unlike the Visual Basic “METER” example, this one uses interrupts. It uses a “delegate” to set up the user interrupt callback function.

## 4.8 DIO_TC.DLL Source Code

The DIO_CODE sub directory contains the full C source code and documentation for the application interface library (DIO_TC.DLL).

The Windows Dynamic Link Library (DLL) contains over 50 functions and provides a common Applications Program Interface (API) to the supported boards. The library functions allow the boards to be easily applied to many different applications, and provide an easy way of accessing the board's features. The DLL can be called by any language that uses Windows calling conventions.

The library can be built in Microsoft Visual C++ version 4.0 or later using DIO_TC.MDP, with Microsoft Visual C++ version 6.0 or later using DIO_TC.DSW, with Microsoft Visual Studio 2005 or later using DIO_TC.vcproj, or with Borland C version 4.2 using DIO_TC.IDE.

The DIO_TC.vcproj file can also be used to build DIO_TC.DLL using the free Microsoft Visual C++ 2005 “Express” edition, but it will also be necessary to obtain and install “Microsoft Platform SDK” and configure the directories in the “Projects and Solutions” section in the “Options” dialog in Visual C++ Express. In that section the following paths should be added to the appropriate subsections:

- Executable files: C:\Program Files\Microsoft SDK\Bin
- Include files: C:\Program Files\Microsoft SDK\include
- Library files: C:\Program Files\Microsoft SDK\Lib

(The above paths need to be changed if the Platform SDK has been installed somewhere else.) These options are global, not part of a specific project, so they only need to be configured once.

For AMPDIO versions up to 4.46, the DIO_TC.DLL file is built in the “Release” or “Debug” subdirectory (according to the selected build configuration), along with the DLL export library, DIO_TC.LIB. For AMPDIO version 5.00, these have been changed to “Win32\Release” and “Win32\Debug” for the 32-bit version of DIO_TC.DLL, with the x64 version being built in “x64\Release” or “x64\Debug”. The examples in the “EX_C\” directory expect to find DIO_TC.LIB in the “DIO_CODE\” directory (“DIO_CODE\Win32” or “DIO_CODE\x64” for AMPDIO v5.00 and later), so it will be necessary to copy the file there from the “Release” or “Debug” subdirectory if significant changes have been made. The newly built DIO_TC.DLL should be copied to the correct Windows system directory. For Windows 95, 98 or ME, this is the “system” directory. For Windows NT, Windows 2000 and 32-bit versions of Windows XP, Windows Vista, Windows 7 and Windows Server 2003, this is the “system32” directory. For “x64” editions of Windows XP, Windows Vista, Windows 7, Windows Server 2003 and Windows Server 2008, the “x64” build of DIO_TC.DLL goes in the “system32” directory and the 32-bit “Win32” build of DIO_TC.DLL goes in the “SysWoW64” directory (this may seem the opposite of what one might expect!).

Up to AMPDIO v4.46, the shipped version of DIO_TC.DLL and DIO_TC.LIB were built with Microsoft Visual C++ 4.2. From AMPDIO v5.00 onwards, the shipped “Win32” version of DIO_TC.DLL was built with Microsoft Visual C++ 6.0, but the corresponding DIO_TC.LIB was still built with Visual C++ 4.2 for backwards compatibility. The shipped “x64” versions of DIO_TC.DLL and DIO_TC.LIB were built with Microsoft Visual Studio 2005.

The “Readme.TXT” file in the “DIO_CODE\” directory describes changes made in each version of DIO_TC.DLL and describes each function in the DLL. The function descriptions are also in section 6.4 of this document.

The ADIOCTL.RTF file documents the low-level IOCTL interface to the driver. Under most circumstances, it is recommended that the more user-friendly DLL interface be used.

## 4.9 SYS_DLLS

This directory contains Microsoft Visual Basic and Visual C system DLLs that may be required to run the examples. Under normal circumstances, these DLLs will have already been installed onto your system. If you experience difficulties running the examples, follow the instructions in the ‘README_DLL.TXT’ file found in the ‘SYS_DLLS’ directory.

For AMPDIO v5.00 and later, the required system DLLs and other run-time files needed to run the example programs are automatically installed, so the SYS_DLLS directory is no longer installed.

# 5 STRUCTURE AND ASSIGNMENTS OF THE REGISTERS

In order to gain the maximum out of the ADIO driver it is useful to have an appreciation of the underlying register locations. The driver was originally developed for the series 200 DIO cards and the register structure used on those cards forms the fundamental basis of the driver architecture.

## 5.1 Register Assignments on Series 200 DIO Cards

The series 200 registers occupy 32 consecutive address locations in the I/O space. A table summarising the register assignments is shown in section 5.3. Please note that the actual register address is the base address configured on the board plus the register offset given in the table.

## 5.2 Register Grouping

All the DIO boards in the 200 series, PC212E, PC214E, PC215E, PC218E, PC272E, PCI215 and PCI272 series have the same register map, which is split up into five groups. Other supported cards are fit into the same grouping scheme as far as possible.

### 5.2.1 Cluster X, Y and Z Groups

Each of the Cluster X, Y and Z groups is populated with either an 82C55 Programmable Peripheral Interface (PPI) device to provide digital input/output, or two 82C54 Counter/Timer devices. Each of the boards in the range deploys various combinations of these devices. The analogue I/O cards still support this idea, but other resources can be found in the X,Y, Z groups.

### 5.2.2 Counter Connection Register Group

The Counter Connection Register (CT) group is supported by the PC212E, PC215E, PC218E and PCI215 series 200 boards and the PCI230, PCI260, PCI224, PCI234 PCI analogue I/O cards. These registers provide software-programmable clock and gate connections for the on-board counter/timer groups. Other supported cards allow selection of the timer/counter clock source by means of jumpers, and do not provide gate source selection.

### 5.2.3 Interrupts Group

Most of the supported cards have an interrupt enable (IE) register. This register provides programmable interrupt source selection and interrupt status information. On cards which do not have an interrupt enable/ interrupt status register, only one interrupt source should be used at a time. Sometimes the interrupt source is selected by means of a jumper on these cards.

## 5.3 The Drivers View of The Register Layout

The driver divides the I/O space into 8 I/O blocks; the first 6 blocks can be 82C54 counter timers (CT) or 82C55 programmable peripheral interfaces (PPI) or analogue I/O or empty, depending on which card is installed. The next block is the counter timer clock and gate connection block and the last is the interrupt enable block.

Most cards fit into this generalised I/O structure, but some cards do not support the counter timer clock and gate connection block, or the interrupt enable block.

Each CT or PPI I/O block contains 4 ports, 3 data ports and a control port. When writing to data registers, the port number can be set between 0 and 3. A PPI block is usually followed by an empty block.

Up to six individually programmable interrupt sources can be assigned to the cards interrupt using the Interrupt Enable (IE) registers. The precise interrupts sources available are card specific and are detailed in the card's manual.

| ADDRESS OFFSET | IO Block | PC218E | PC212E | PC215E | PC214E | PC272E |
|---|---|---|---|---|---|---|
| $00_{16} – 03_{16}$ | 0 (X1) | CT | PPI | PPI | PPI | PPI |
| $04_{16} – 07_{16}$ | 1 (X2) | CT | | | | |
| $08_{16} – 0B_{16}$ | 2 (Y1) | CT | CT | PPI | PPI | PPI |
| $0C_{16} – 0F_{16}$ | 3 (Y2) | CT | CT | | | |
| $10_{16} – 13_{16}$ | 4 (Z1) | CT | CT | CT | CT | PPI |
| $14_{16} – 17_{16}$ | 5 (Z2) | CT | CT | CT | | |
| $18_{16} – 1D_{16}$ | 6 | CT Connect | CT Connect | CT Connect | Unused | Unused |
| $1E_{16}$ | 7 | IE | IE | IE | Unused | IE |
| $1F_{16}$ | | Unused | Unused | Unused | Unused | Unused |

The version 2.00 (and above) driver also supports the following card types:

| ADDRESS OFFSET | IO Block | PC36AT | PC263 |
|---|---|---|---|
| $00_{16} – 03_{16}$ | 0 (X1) | PPI | 16 relay |
| $04_{16} – 07_{16}$ | 1 (X2) | | |
| $08_{16} – 0B_{16}$ | 2 (Y1) | | |
| $0C_{16} – 0F_{16}$ | 3 (Y2) | | |
| $10_{16} – 13_{16}$ | 4 (Z1) | | |
| $14_{16} – 17_{16}$ | 5 (Z2) | | |
| $18_{16} – 1D_{16}$ | 6 | | |
| $1E_{16}$ | 7 | | |
| $1F_{16}$ | | | |

The version 4.00 (and above) driver also supports the following card types:

| ADDRESS OFFSET | IO Block | PC24/ PC25 | PC27 | PC30AT | PC26AT |
|---|---|---|---|---|---|
| $00_{16} – 03_{16}$ | 0 (X1) | 2 DACs | ADC | ADC | ADC |
| $04_{16} – 07_{16}$ | 1 (X2) | 2 DACs | CT | CT | CT |
| $08_{16} – 0B_{16}$ | 2 (Y1) | CT | | PPI | |
| $0C_{16} – 0F_{16}$ | 3 (Y2) | | | DAC | |
| $10_{16} – 13_{16}$ | 4 (Z1) | | | DAC | |
| $14_{16} – 17_{16}$ | 5 (Z2) | | | | |
| $18_{16} – 1D_{16}$ | 6 | | | | |
| $1E_{16}$ | 7 | | | | |
| $1F_{16}$ | | | | | |

The blocks marked ‘2 DACs’ consist of a pair of D-to-A converters. The blocks marked ‘DAC’ consist of a single D-to-A converter. The blocks marked ‘ADC’ consist of a multiplexed A-to-D converter.

The version 4.10 (and above) driver also supports the following PCI card types:

| ADDRESS OFFSET | IO Block | PCI230 | PCI260 |
|---|---|---|---|
| $00_{16}$ – $03_{16}$ | 0 (X1) | PPI | |
| $04_{16}$ – $07_{16}$ | 1 (X2) | | |
| $08_{16}$ – $0B_{16}$ | 2 (Y1) | ADC | ADC |
| $0C_{16}$ – $0F_{16}$ | 3 (Y2) | | |
| $10_{16}$ – $13_{16}$ | 4 (Z1) | | |
| $14_{16}$ – $17_{16}$ | 5 (Z2) | CT | CT |
| $18_{16}$ – $1D_{16}$ | 6 | CT Connect | CT Connect |
| $1E_{16}$ | 7 | IE | IE |
| $1F_{16}$ | | Unused | Unused |

The blocks marked ‘ADC’ on these PCI cards are just placeholders for the interrupt source. The actual registers are not mapped into this area. The PCI230 also has two DAC channels, but the registers are not mapped into this area. See the PCI230/PCI260 manual for full details of the registers.

The version 4.20 (and above) driver also supports the following PCI card types:

| ADDRESS OFFSET | IO Block | PCI224 | PCI234 |
|---|---|---|---|
| $00_{16}$ – $03_{16}$ | 0 (X1) | EXT | EXT |
| $04_{16}$ – $07_{16}$ | 1 (X2) | | |
| $08_{16}$ – $0B_{16}$ | 2 (Y1) | DACS | DACS |
| $0C_{16}$ – $0F_{16}$ | 3 (Y2) | | |
| $10_{16}$ – $13_{16}$ | 4 (Z1) | | |
| $14_{16}$ – $17_{16}$ | 5 (Z2) | CT | CT |
| $18_{16}$ – $1D_{16}$ | 6 | CT Control | CT Control |
| $1E_{16}$ | 7 | IE | IE |
| $1F_{16}$ | | Unused | Unused |

The blocks marked ‘DACS’ on these PCI cards are just placeholders for the interrupt source. The actual registers are not mapped into this area. The PCI224 has 16 multiplexed 12-bit DAC channels. The PCI234 has 4 multiplexed 16-bit DAC channels. See the PCI224/PCI234 manual for full details of the registers.

Similarly, the blocks marked ‘EXT’ on these PCI cards are just placeholders for an externally triggered interrupt source.

The version 4.31 (and above) driver also supports the following PCI card types:

| ADDRESS OFFSET | IO Block | PCI215 | PCI236 | PCI272 | PCI263 |
|---|---|---|---|---|---|
| $00_{16}$ – $03_{16}$ | 0 (X1) | PPI | PPI | PPI | 16 relay |
| $04_{16}$ – $07_{16}$ | 1 (X2) | | | | |
| $08_{16}$ – $0B_{16}$ | 2 (Y1) | PPI | | PPI | |
| $0C_{16}$ – $0F_{16}$ | 3 (Y2) | | | | |
| $10_{16}$ – $13_{16}$ | 4 (Z1) | CT | | PPI | |
| $14_{16}$ – $17_{16}$ | 5 (Z2) | CT | | | |
| $18_{16}$ – $1D_{16}$ | 6 | CT Connect | | Unused | |
| $1E_{16}$ | 7 | IE | | IE | |
| $1F_{16}$ | | Unused | | Unused | |

The PCI215, PCI272 and PCI263 are the PCI equivalents of the PC215E and PC272E and PC263 respectively. The PCI236 is the PCI equivalent of the PC36AT.

## 5.4 The Register Details

### 5.4.1 82C55 Programmable Peripheral Interface Registers

The following paragraphs describe the operations of the 82C55 Programmable Peripheral Interface, which is a common element on many of the supported cards. Full details may be found in the ‘App82c55.pdf’ file in the ‘manual’ sub-directory of the SOFTMAN CD.

Each programmable peripheral interface has the following register configuration. Offsets are from the start of the PPI block:

| Port Offset | Description | Access | Bits |
|---|---|---|---|
| $00_{16}$ | PPI Port A | R/W | 8 |
| $01_{16}$ | PPI Port B | R/W | 8 |
| $02_{16}$ | PPI Port C | R/W | 8 |
| $03_{16}$ | PPI Control | W | 8 |

#### 5.4.1.1 82C55 Programmable Peripheral Interface PPI Data Register Port A

This eight-bit register writes to and reads from port A of the 82C55 Programmable Peripheral Interface PPI.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$00_{16}$** | Write and Read | 8 bits | 82C55 Programmable Peripheral Interface Port A Data Register | **PPI A** |

**FUNCTION**

The PPI Port A Data Register is used to write or read 8 bit data to port A of the 82C55 Programmable Peripheral Interface device PPI.

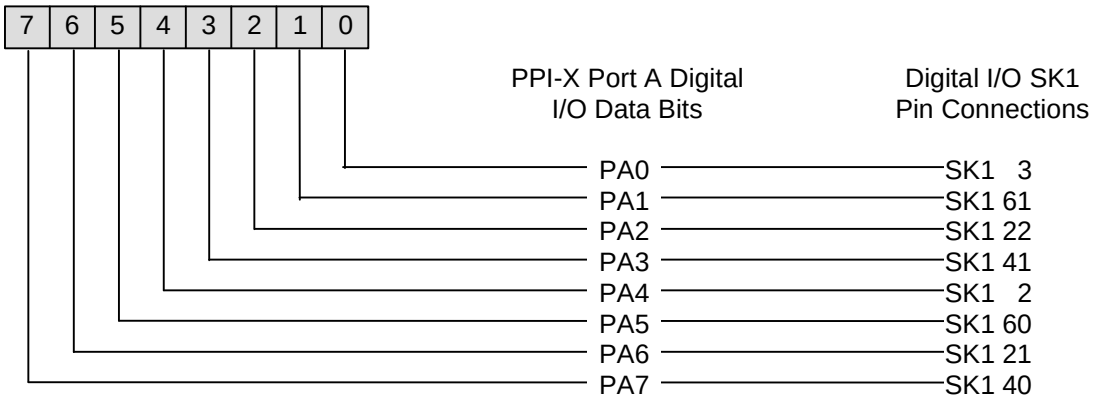
The PPI can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C55 data sheets in the appendices on the SOFTMAN CD.

The eight data bits of port A are data input, data output or bi-directional data I/O according to the PPI mode:

| | |
|---|---|
| Mode 0 | Input or Output |
| Mode 1 | Input or Output |
| Mode 2 | Bi-directional Input/output |

**BIT ASSIGNMENTS**

The bit layout of the PPI-X Port A data register is shown below.




### 5.4.1.2 82C55 Programmable Peripheral Interface PPI Data Register Port B

This eight-bit register writes to and reads from port B of the 82C55 Programmable Peripheral Interface PPI.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$01_{16}$** | Write and Read | 8 bits | 82C55 Programmable Peripheral Interface Port B Data Register | **PPI B** |

**FUNCTION**

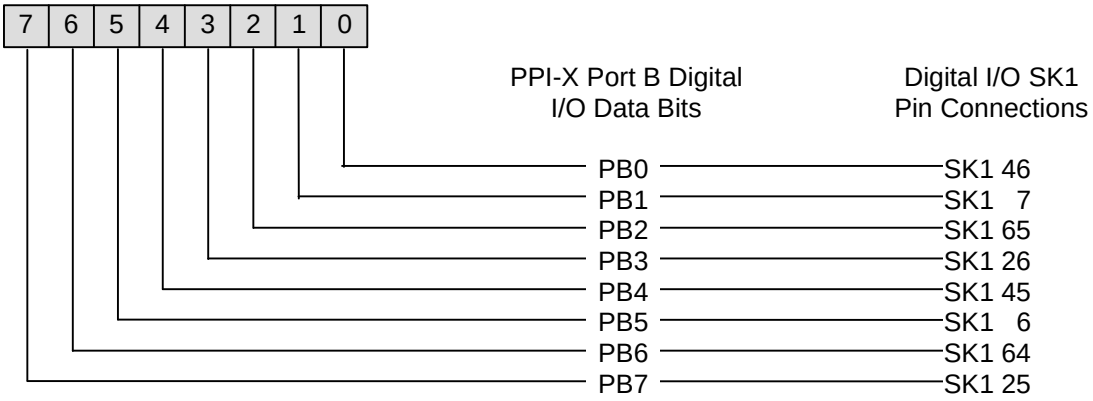
The PPI Port B Data Register is used to write or read 8 bit data to a port of the 82C55 Programmable Peripheral Interface device.

The PPI can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C55 data sheets in the appendices on the SOFTMAN CD.

The eight data bits of port B are data input or data output in all modes

**BIT ASSIGNMENTS**

The bit layout of the PPI-X port B data register is shown below.




### 5.4.1.3 82C55 Programmable Peripheral Interface PPI Data Register Port C

This eight-bit register writes to and reads from port C of the 82C55 Programmable Peripheral Interface PPI.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$02_{16}$** | Write and Read | 8 bits | 82C55 Programmable Peripheral Interface Port C Data Register | **PPI C** |

**FUNCTION**

The PPI Port C Data Register is used to write or read 8 bit data to a port of the 82C55 Programmable Peripheral Interface device

The PPI can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C55 data sheets in the appendices on the SOFTMAN CD.

The eight data bits of port C are split into two groups, the upper port C bits 4 to 7 and the lower port C bits 0 to 3. These bits can be data input, data output or control/handshake lines according to the PPI mode:

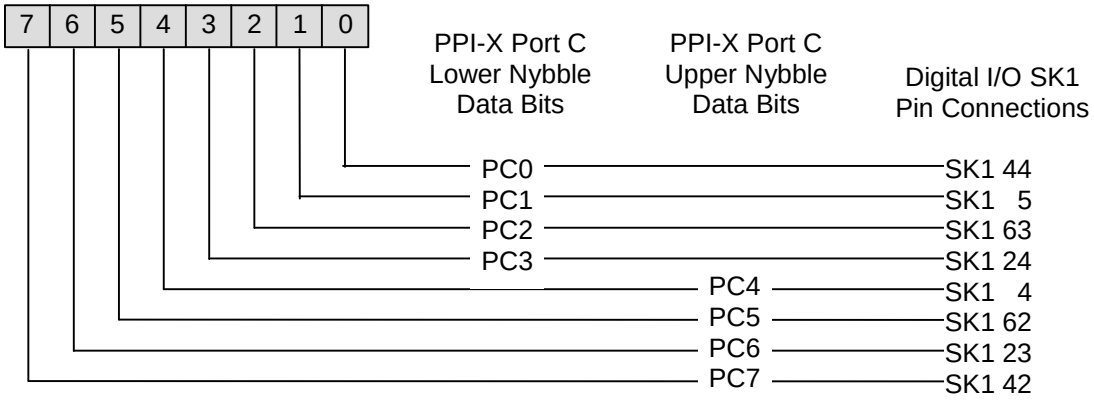
| Mode | Port C Upper | Port C Lower |
|---|---|---|
| Mode 0 | Input or Output | Input or Output |
| Mode 1 | Control/Data | Control/Data |
| Mode 2 | 5 bit Control (PC3 to PC7) | 3 bit Control/Data (PC0 to PC2) |

With bit 7 'Command Select' set to '0', any of the eight bits of port C can be set or reset using a single output instruction. When port C is being used as status/control for port A or port B, these bits can be set or reset using the Bit Set/Reset operation just as if they were data output ports.

A full description of the operating modes and all other features of the 82C55 are available in the manufacturer’s data sheet for the 82C55, available on the SOFTMAN CD.

**BIT ASSIGNMENTS**

The bit layout of the PPI-X port C data register is shown below.




#### 5.4.1.4 82C55 Programmable Peripheral Interface PPI Command Register

This is the command register for the PPI and can be used to set the operational mode of the three digital I/O ports or to manipulate the bits of port C.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$03_{16}$** | Write | 8 bits | 82C55 Programmable Peripheral Interface PPI Command Register | **PPI CMD** |

**FUNCTION**

Provides a command word to define the operation of the PPI ports A, B and C. Any port programmed as output is initialized to all zeroes when a command word is written. A separate feature allows any bit of port C to be set or reset using a single instruction.

The programming procedure for the 82C55 is flexible, but the command word must be written before data bytes are loaded. As the command register and each port have separate addresses (offsets 0 to 3) and each command word specifies the mode of each port, no other special instruction sequence is required.

**The Three Modes**

The register function depends on the setting of bit 7 'Command Select' and the three mode selections assume that bit 7 is set to '1', which allows mode configuration.

Mode 0 provides basic input and output operations through each of the ports A, B and C. Output data bits are latched and input data follows the signals applied to the I/O lines. No handshaking is needed.

- 16 different configurations in mode 0
- Two 8 bit ports and two 4 bit ports
- Inputs are not latched
- Outputs are latched

Mode 1 provides strobed input and output operations with data transferred through port A or B and handshaking through port C.

- Two I/O groups (Group A — also known as Group 0 or Group I)
  (Group B — also known as Group 1 or Group II)
- Both groups contain an 8 bit port and a 4 bit control/data port
- Both 8 bit data ports can be latched input or latched output

Mode 2 provides strobed bi-directional operation using port A as the bi-directional data bus. Port C3 to C7 bits are used for interrupts and handshaking bus flow control similar to mode 1. NOTE: Port B and port C0 to C2 bits may be defined as mode 0 or 1, input or output in conjunction with port A in mode 2.

- An 8 bit latched bi-directional bus port and 5 bit control port
- Both input and outputs are latched
- An additional 8 bit input or output port with a 3 bit control port

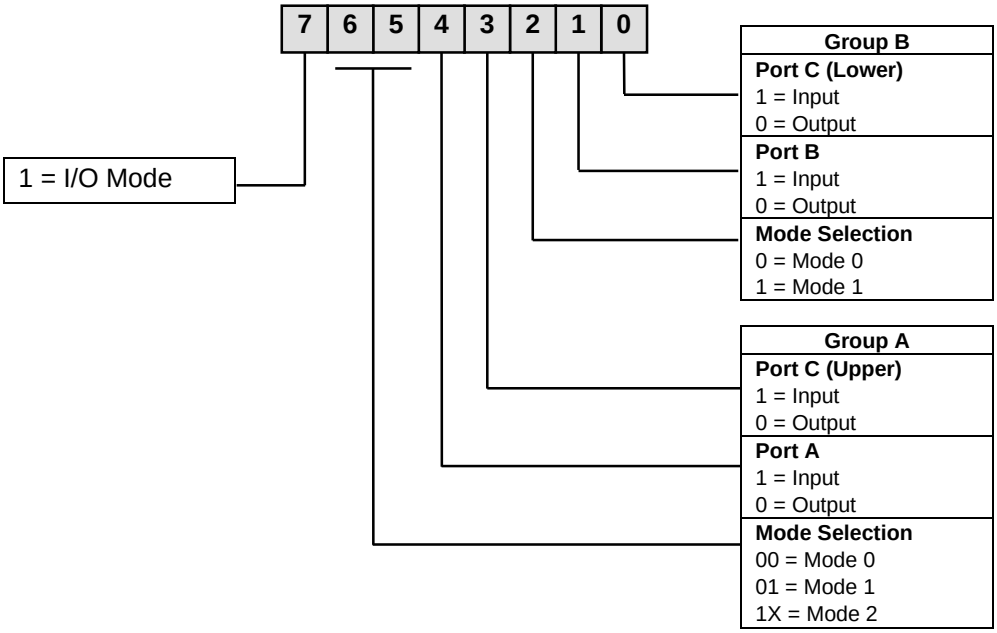
**Single Bit Set/Reset Feature**

With bit 7 'Command Select' set to '0', any of the eight bits of port C can be set or reset using a single output instruction. This feature reduces the software overhead in control based applications. When port C is being used as status/control for port A or port B, these bits can be set or reset using the Bit Set/Reset operation just as if they were data output ports.

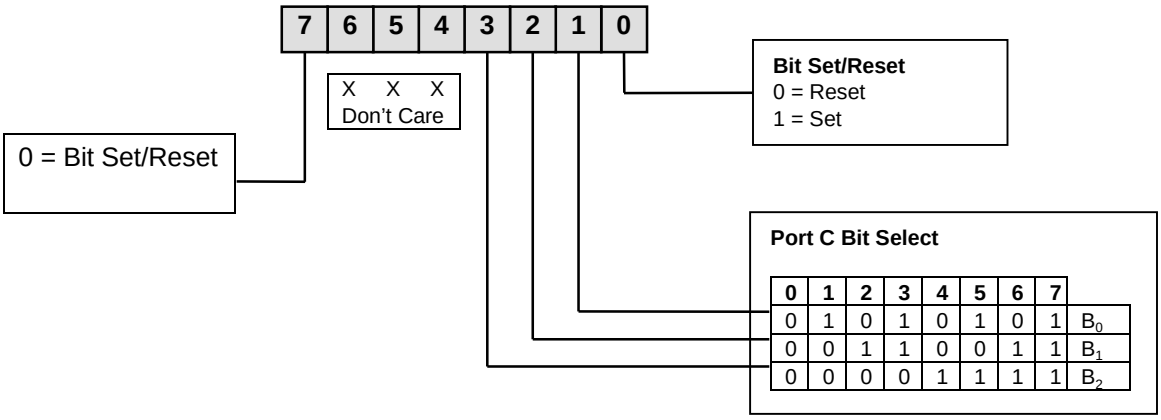
**BIT ASSIGNMENTS**

Bit layouts of the PPI command word register is shown below.

**Command Word for Mode Definition Format**



**Command Word for Bit Set/Reset Format**




### 5.4.2 82C54Counter Timer Registers

The following paragraphs describe the operations of the 82C54 counter timer that is a common element on many of the supported cards. Each 82C54 has three counter timer channels. Full details may be found in the device manufacturer's data sheet in the file 'App82c54.pdf' file in the 'manual' sub-directory of the SOFTMAN CD.

Note that the supported ISA analogue cards (PC24E/PC25E, PC26AT, PC27E and PC30AT) use the similar, but slightly less functional 82C53 counter timer. Most of the details below also apply to the 82C53, full details of which may be found in the 'App82c53.pdf' file in the 'manual' sub-directory of the SOFTMAN CD. The Read Back command which the driver performs before reading counter timer values is only supported by the 82C54 and does not work on the 82C53, so that the values read back will be unreliable. The counter timers on the cards which use the 82C53 are only intended to be used for frequency generation, so this is not much of a problem.

Version 4.23 and later uses the Counter Latch command on the 82C53 to latch a single counter value at a time instead of the unsupported Read Back command which can latch 2 or 3 counter values at a time on the 82C54. This allows the counter timer values on cards that use the 82C53 to be read back more reliably.

Each counter timer has the following register configuration. Offsets are from the start of the counter timer block:

| Port Offset | Description | Access | Bits |
|---|---|---|---|
| $00_{16}$ | Counter Timer 0 Data register | R/W | 8 |
| $01_{16}$ | Counter Timer 1 Data register | R/W | 8 |
| $02_{16}$ | Counter Timer 2 Data register | R/W | 8 |
| $03_{16}$ | Counter Timer Control register | W | 8 |

#### 5.4.2.1 82C54 Counter 0 Data Register

The 82C54 Programmable Timer Counter provides three 16 bit counter/timers which can be independently programmed to operate in any one of six modes with BCD or Binary count functions.

The register definition for Counter 0 Data is as follows.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$00_{16}$** | Write and Read | 8 bits | 82C54 Counter/Timer Counter 0 Data Register | **CT0** |

**FUNCTION**

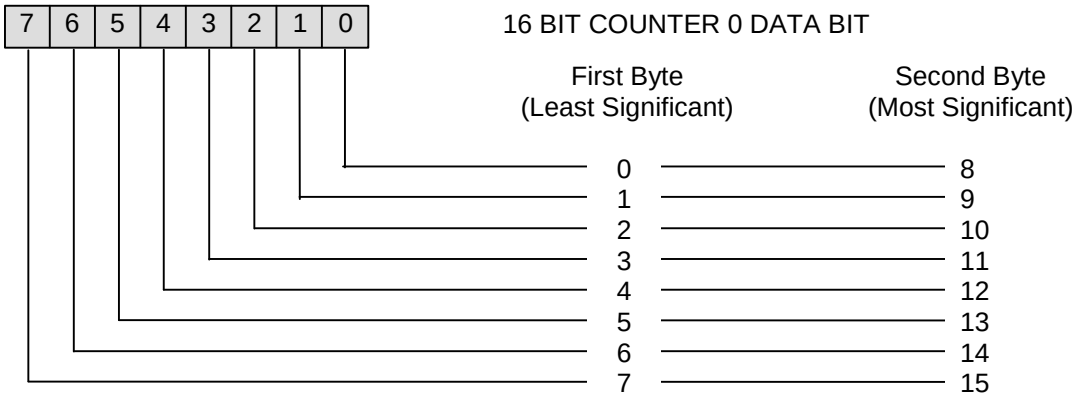
The Counter 0 Data Register is used to write and read 8 bit data to the 82C54 counter/timer 0. The counter is normally configured for 16 bit operation and to ensure validity of the data it is important to always write/read two bytes to the register, least significant byte first. Please note that the 16-bit count values written to this register are not latched into the counting element until the next clock pulse (assuming the gate input is high). Subsequent read operations from this register will therefore not reflect the new count value until this clock pulse has latched the data.

This register is also used to read counter 0 status if the status has been latched using the Read-Back command (not supported on 82C53).

The counter can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C54 (or 82C53) data sheets.

**BIT ASSIGNMENTS**

The bit layout of the counter 0 register is shown below.




### 5.4.2.2 82C54 Counter 1 Data Register

The 82C54 Programmable Timer Counter provides three 16 bit counter/timers which can be independently programmed to operate in any one of six modes with BCD or Binary count functions. The register definition for Counter 1 Data is as follows.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$01_{16}$** | Write and Read | 8 bits | 82C54 Counter/Timer Counter 1 Data Register | **CT1** |

**FUNCTION**

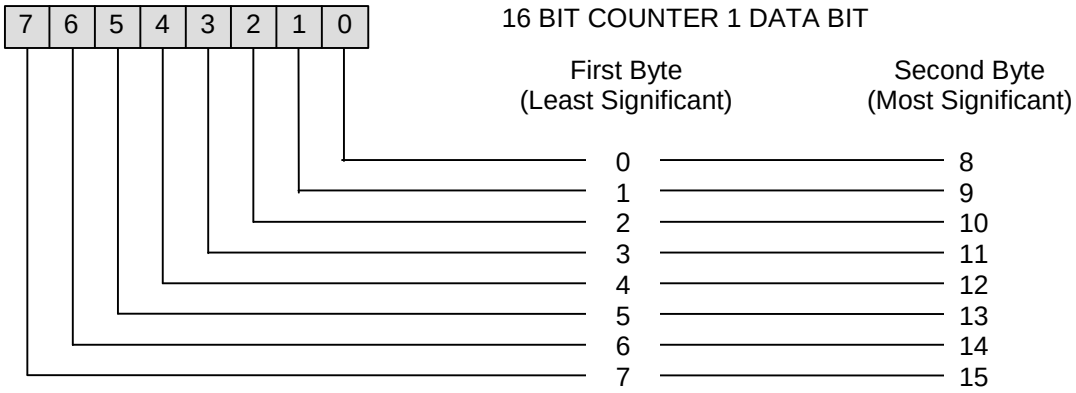
The Counter 1 Data Register is used to write and read 8 bit data to the 82C54 counter/timer 1. The counter is normally configured for 16 bit operation and to ensure validity of the data it is important to always write/read two bytes to the register, least significant byte first. Please note that the 16-bit count values written to this register are not latched into the counting element until the next clock pulse (assuming the gate input is high). Subsequent read operations from this register will therefore not reflect the new count value until this clock pulse has latched the data.

This register is also used to read counter 1 status if the status has been latched using the Read-Back command (not supported on 82C53).

The counter can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C54 (or 82C53) data sheets.

**BIT ASSIGNMENTS**

The bit layout of the counter 1 register is shown below.




#### 5.4.2.3 Counter 2 Data Register

The 82C54 Programmable Timer Counter provides three 16 bit counter/timers which can be independently programmed to operate in any one of six modes with BCD or Binary count functions. The register definition for Counter 2 Data is as follows.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$02_{16}$** | Write and Read | 8 bits | 82C54 Counter/Timer Counter 2 Data Register | **CT2** |

**FUNCTION**

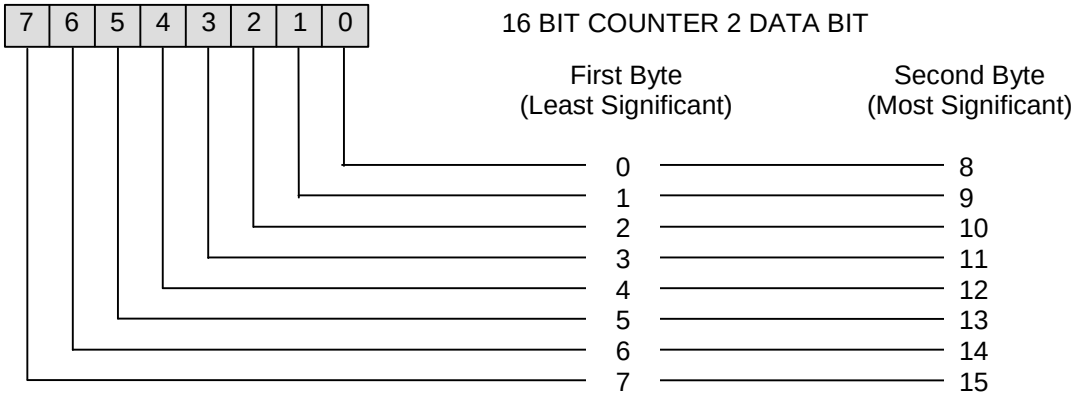
The Counter 2 Data Register is used to write and read 8 bit data to the 82C54 counter/timer 2. The counter is normally configured for 16 bit operation and to ensure validity of the data it is important to always write/read two bytes to the register, least significant byte first. Please note that the 16-bit count values written to this register are not latched into the counting element until the next clock pulse (assuming the gate input is high). Subsequent read operations from this register will therefore not reflect the new count value until this clock pulse has latched the data.

This register is also used to read counter 2 status if the status has been latched using the Read-Back command (not supported on 82C53).

The counter can be configured to operate in several modes. Further details may be found by reference to the device manufacturer's 82C54 (or 82C53) data sheets.

**BIT ASSIGNMENTS**

The bit layout of the counter 2 register is shown below.




#### 5.4.2.4 Counter/Timer Control Register

The control register provides the means to configure the three sixteen bit counter/timers of the 82C54. An outline of its operation is given here, but reference should be made to the 82C54 device manufacturers' data sheets in the appendices on the SOFTMAN CD before programming of the counter is attempted.

The Counter Timer Control register is a WRITE register.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| **$03_{16}$** | Write | 8 bits | 82C54 Counter/Timer Control Register | **CTC** |

**FUNCTION**

The Counter Timer Control register is used to define the operation of the counters 0, 1 and 2, and to latch counter values and/or status of one or more counters.

The programming procedure for the 82C54 is flexible, but the following two conventions must be followed:

- For each counter, the control word must be written before the initial count is loaded.
- The initial count must follow the count format specified in the control word. This format is normally least significant byte followed by most significant byte (control word bits 5 & 4 = 1 & 1) but can be L.S. byte only or M.S. byte only.

As the control register and each counter have separate addresses (offsets 0, 1, 2 and 3) and each control word specifies the counter it applies to (bits 6 and 7) no special instruction sequence is required.

When a control word is written to a counter, all control logic is reset and OUT goes to a known initial state depending on the mode selected.

The six counter modes are:

Mode 0 Interrupt on Terminal Count
Mode 1 Hardware Re-triggerable One-shot
Mode 2 Rate Generator
Mode 3 Square Wave

Mode 4 Software Triggered Mode
Mode 5 Hardware Triggered Strobe (Re-triggerable)

**BIT ASSIGNMENTS**

The bit layout of the counter timer control register is shown below.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| SC1 | SC0 | RW1 | RW0 | M2 | M1 | M0 | BCD |

**SC – Select Counter**

| SC1 | SC0 | |
|---|---|---|
| 0 | 0 | Select Counter 0 |
| 0 | 1 | Select Counter 1 |
| 1 | 0 | Select Counter 2 |
| 1 | 1 | Read-Back Command (See Below) |

**RW – Read/Write**

| RW1 | RW0 | |
|---|---|---|
| 0 | 0 | Counter Latch Command (See Below) |
| 0 | 1 | Read/Write least significant byte only. |
| 1 | 0 | Read/Write most significant byte only. |
| 1 | 1 | Read/Write least significant byte first, then most significant byte. |

**M – Mode**

| M2 | M1 | M0 | |
|---|---|---|---|
| 0 | 0 | 0 | Mode 0 |
| 0 | 0 | 1 | Mode 1 |
| X | 1 | 0 | Mode 2 |
| X | 1 | 1 | Mode 3 |
| 1 | 0 | 0 | Mode 4 |
| 1 | 0 | 1 | Mode 5 |

**BCD – Binary Coded Decimal**

| | |
|---|---|
| 0 | Binary Counter 16-bits |
| 1 | Binary Coded Decimal (BCD) Counter (4 Decades) |

The format of the Counter Latch Command and Read-Back Command are shown below.

**Counter Latch Command**

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| SC1 | SC0 | 0 | 0 | X | X | X | X |

SC1, SC0 specify counter to be latched:

| SC1 | SC0 | COUNTER |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | Read-Back Command |

**Read-Back Command**

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | /COUNT | /STATUS | CNT2 | CNT1 | CNT0 | 0 |

D5: 0 = Latch count of selected Counter(s)
D4 0 = Latch status of selected Counter(s)
D3 1 = Select Counter 2
D4 1 = Select Counter 1
D5 1 = Select Counter 0

N.B. The Read-Back Command is not supported by the 82C53. Prior to AMPDIO v4.23, The driver uses the Read-Back Command to latch counters, so reading counters is not reliable on those cards which use the 82C53.

Latching the count of selected channels with the Read-Back Command has the same effect as the Counter Latch Command on those channels.

If the status of a counter is latched, the next read from that counter's register will read and unlatch the status. Otherwise, if the count of a channel is latched, the next 1 or 2 reads from that counter's register (depending on the Read/Write configuration) will read 1 or 2 halves of the counter value and unlatch the count.

The counter status format is shown below.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| OUTPUT | NULL COUNT | RW1 | RW0 | M2 | M1 | M0 | BCD |

D7 1 = OUT pin is 1
0 = OUT pin is 0
D6 1 = Null count
0 = Count available for reading
D5–D0 Counter programmed mode

Further information on programming the 82C54 Programmable Counter/Timer can be found in chapters 5 and 2. For a full description of the six operating modes and all other features of the 82C54, see the manufacturer's data sheet for the 82C54 in the appendices on the SOFTMAN CD.

### 5.4.3 Clock and Gate Configuration Registers

Clock and counter timer connection registers have the following configuration. Offsets are from the start of the CT Control block in the register map ($18_{16}$ on all supported cards):

| Port Offset | Description | Access | Bits |
|---|---|---|---|
| $00_{16}$ | Clock Connections for group X timers | W | 8 |
| $01_{16}$ | Clock Connections for group Y timers | W | 8 |
| $02_{16}$ | Clock Connections for group Z timers | W | 8 |
| $03_{16}$ | Gate Connections for group X timers | W | 8 |
| $04_{16}$ | Gate Connections for group Y timers | W | 8 |
| $05_{16}$ | Gate Connections for group Z timers | W | 8 |

#### 5.4.3.1 Group Clock Connection Registers

These registers can be used to select the counter/timer clock sources.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| $00_{16}$ | Write | 8 bits | Group X Counter/timer Clock Selection Register | **XCLK_SCE** |
| $01_{16}$ | Write | 8 bits | Group Y Counter/timer Clock Selection Register | **YCLK_SCE** |
| $02_{16}$ | Write | 8 bits | Group Z Counter/timer Clock Selection Register | **ZCLK_SCE** |

**FUNCTION**

Individually selects one of the eight possible Counter/Timer clock sources for each counter/timer channel.

**The Eight Clock Sources**

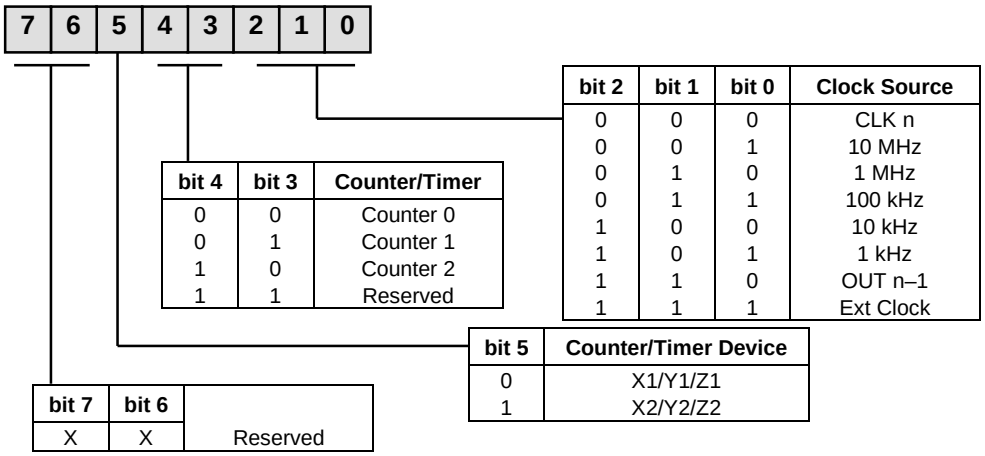
The eight possible clock sources are as follows:

1. The counter/timer's CLK input from the SK1 connector
2. The internal 10 MHz clock
3. The internal 1 MHz clock
4. The internal 100 kHz clock
5. The internal 10 kHz clock
6. The internal 1 kHz clock
7. The output of the preceding counter/timer channel (OUT n–1)
8. The dedicated external clock input for the group (X1/X2, Y1/Y2 or Z1/Z2)

N.B. The preceding counter/timer channel for channel 0 on a particular counter/timer chip is channel 2 of the preceding counter/timer chip. The highest counter/timer chip is considered to precede the lowest counter/timer chip for this purpose. For example, on the PC215E, which has two counter/timer chips, Z1 and Z2, the OUT n–1 clock source for Z2 channel 0 comes from the output of Z1 channel 2, and the OUT n–1 clock source for Z1 channel 0 comes from the output of Z2 channel 2.

**BIT ASSIGNMENTS**

Bit layout of each clock connection register is shown below.




### 5.4.3.2 Group Gate Connection Registers

These registers can be used to select the counter/timer gate input sources for each counter/timer channel.

| Register Offset | Write and/or Read | Register Width | Register Title | Mnemonic |
|---|---|---|---|---|
| $03_{16}$ | Write | 8 bits | Group X Counter/timer Gate Selection Register | **XGAT_SCE** |
| $04_{16}$ | Write | 8 bits | Group Y Counter/timer Gate Selection Register | **YGAT_SCE** |
| $05_{16}$ | Write | 8 bits | Group Z Counter/timer Gate Selection Register | **ZGAT_SCE** |

**FUNCTION**

Individually selects one of the four possible Counter/Timer gate input signal sources for each counter/timer channel.

**The Four Gate Sources**

The four[1] possible gate sources are as follows:

1. VCC (internal +5V d.c.) — i.e. gate permanently enabled
2. GND (internal 0V d.c.) — i.e. gate permanently disabled
3. GAT n — the counter/timer's GAT input from the SK1 connector
4. /OUT n–2 — the inverted output of counter timer n–2

N.B. The n–2 channel for channel 0 on a particular counter/timer chip is channel 1 of the preceding counter/timer chip, and the n–2 channel for channel 1 on a particular counter/timer chip is channel 2 of the preceding counter/timer chip. The highest counter/timer chip is considered to precede the lowest counter/timer chip for this purpose. For example, on the PC215E, which has two counter/timer chips, Z1 and Z2, the /OUT n–2 gate source for Z2 channel 0 comes from the output

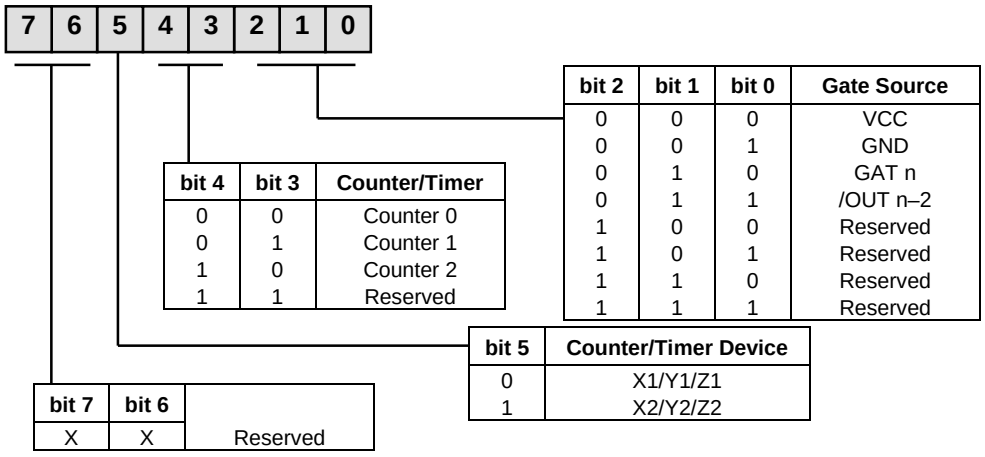
[1] Some cards support up to four additional gate sources.

of Z1 channel 1, and the /OUT n–2 gate source for Z1 channel 0 comes from the output of Z2 channel 1.

For the PCI230 and PCI230+, the Z2 counter/timer's GAT inputs are connected internally to PPI-X C0, C1 and C2. For the PCI260+, all three counter/timer GAT inputs are connected internally to the external trigger input (SK1 pin 17). For the original PCI260, PCI224 and PCI234, the GAT input is not connected.

**BIT ASSIGNMENTS**

Bit layout of each gate connection register is shown below.




**Additional Gate Sources for PCI230+ and PCI260+**

The PCI230+ and PCI260+ cards support the following additional gate sources:

1. Latched GAT n — starts low and goes high on rising edge of timer/counter's GAT input
2. Latched /GAT n — starts low and goes high on falling edge on timer/counter's GAT input
3. /GAT n — inverted timer/counter's GAT input
4. OUT n–2 — the non-inverted output of counter timer n–2

| bit 2 | bit 1 | bit 0 | Gate Source |
|---|---|---|---|
| 0 | 0 | 0 | VCC |
| 0 | 0 | 1 | GND |
| 0 | 1 | 0 | GAT n |
| 0 | 1 | 1 | /OUT n–2 |
| 1 | 0 | 0 | Latched GAT n |
| 1 | 0 | 1 | Latched /GAT n |
| 1 | 1 | 0 | /GAT n |
| 1 | 1 | 1 | OUT n-2 |

# 6 PROGRAMMING WITH THE AMPDIO DRIVER

## 6.1 Windows DLL and Examples

The AMPDIO DIO_TC DLL is a Windows programmer's interface to the analogue and digital IO boards. As long as the compiler/interpreter supports Windows, i.e. uses the Windows calling conventions; all the functions can be called by software written in any language.

The application interface to the driver is defined in a number of header files, for use by different languages.

| Module definition file | Language |
|---|---|
| DIO_TC.H and ADIOCTL.H | Microsoft C/C++ versions 4.x upwards<br>Borland C/C++ versions 4.x and 5.x |
| DIO_TC.BAS | Microsoft Visual Basic version 5.0 |
| DIO_TC.VB | Microsoft Visual Basic .NET |
| DIO_TC_H.CS | Microsoft Visual C# .NET |
| DIO_TC.PAS | Borland Delphi version 3.0 |
| DIO_VEE.H | HP VEE version 4.0 |

For other languages, the user will need to compile a suitable header, in which the DLL functions and constants are declared, using DIO_TC.H as an example. For C programming, some useful macros for constant values may be found in ADIOCTL.H. For Visual Basic, these constants are in the DIO_TC.BAS file. For Delphi programming, they are in the DIO_TC.PAS file.

## 6.2 Support in DOS

The support in DOS is limited to the DOS software supplied separately with individual cards and this is documented in the card manuals. Note, for PCI cards no DOS software is shipped, however a FINDCARD utility is supplied to enable users to develop their own DOS software for these cards. The FINDCARD utility does function under a Windows environment.

The FINDCARD utility will find any installed Amplicon PCI cards and report the plug and play resource allocation. With this information DOS applications can be written for the cards in languages such as Turbo Pascal and Microsoft C.

```
FINDCARD

Card #1 = PCI230 Int = $0B Port 1 = $DC00 Port2 = $E00
```

Where:

Int is the assigned interrupt.
Port1 is the 8-bit register IO space.
Port2 is the 16-bit register IO space.

Please check MEV's web site for up to date DOS PCI Plug and Play libraries and utilities.

### 6.2.1 Windows Library Source Code

The application interface library 'DIO_TC.DLL' is supplied in both executable form and as 'C' source code. The source code can be compiled using Microsoft C/C++ compiler version 4.0 or greater or Borland C++ version 4.0 compiler. The project workspace DIO_TC.MDP, DIO_TC.DSW

or DIO_TC.vcproj is used to build the DLL using Microsoft Visual C/C++ DIO_TC.IDE is used for Borland C++.

## 6.3 Using the Dynamic Link Library

### 6.3.1 C/C++

Section 6.4 describes the library functions available. Please note that in C/C++, the function call examples given should always end with a semi-colon. Where arguments to functions are described as pointers, the address of a user-declared variable is required. This is easily done by using the '&' reference operator. For example, function *TCgetCount* requires a pointer to a variable declared as long, into which the count value result will be placed. A typical 'C' code example for displaying the Z1 Counter 0 count value would be as follows:

```
long count;                          // declare count as long

TCgetCount( h, Z1, 0, &count );      // pass count by reference
printf( "count = %ld", count );      // count now contains new
                                     // value
```

where h is a handle to a registered board. N.B. The large memory model should be used when compiling the library and example programs.

#### 6.3.1.1 Microsoft C/C++

1) Ensure that the library DIO_TC.LIB and the header files DIO_TC.H and ADIOCTL.H can be located by the compiler, either in the project directory or in a path added to the include/library directory paths. Failure to find these files may cause ‘unresolved external’ compilation errors. Note that the location of the preinstalled DIO_TC.LIB file changed in AMPDIO v5.00.

2) At the beginning of the application program, add the following lines:

```
#include <windows.h>
#include "DIO_TC.H"
#include "ADIOCTL.H"
```

3) Add the library link file DIO_TC.LIB into the project workspace.

4) Build the project.

#### 6.3.1.2 Borland C++

1) Ensure that the header files DIO_TC.H and ADIOCTL.H can be located by the compiler, either in the project directory or in a path added to the include/library directory paths. Failure to find these files may cause ‘unresolved external’ compilation errors.

2) At the beginning of the application program, add the following lines:

```
#include <windows.h>
#include "DIO_TC.H"
#include "ADIOCTL.H"
```

3) Generate a Borland C compatible version of the library link file DIO_TC.LIB by using the Borland IMPLIB utility.

4) Add the library newly generated file DIO_TC.LIB to project workspace. Note the DIO_TC.LIB supplied with the AMPDIO package is a Microsoft format library file and is incompatible with Borland compilers.

### 6.3.2 Visual Basic 5.0 and 6.0

The Visual Basic example projects can be opened from within Microsoft Visual Basic by choosing 'File|Open Project…' in the menu and selecting the appropriate .VBP project file within the EX_VB subdirectory of the AMPDIO software directory. The project window will now appear on the desktop. Double-clicking on a .VBP file from within Windows Explorer should also cause Microsoft Visual Basic to run with the selected project file open. Double-click on any file in the project to view the source code, or select Run to run the program.

Note that the .VBP files supplied with AMPDIO v5.00 and later were saved by Visual Basic 6.0. They can still be used with Visual Basic 5.0, but a warning dialog will appear about a 'Retained' key. If you tell Visual Basic to continue loading the project, it will work correctly. Saving the project in Visual Basic 5.0 will stop the warning occuring the next time the project is loaded.

To create your own Visual Basic program from scratch, perform the following steps:

1) From within Microsoft Visual Basic, select 'File|New Project'. A new project window will appear, into an empty Form1 design window will appear.

2) Select 'File|Add file...' and select 'DIO_TC.BAS' from the EX_VB subdirectory of the AmpDIO software directory.

3) Double-click on the empty Form1 design window to bring up the code window for the Form_Load() subroutine. At runtime, this routine will be called when the program first starts up.

4) Type the following lines into the Form_Load() subroutine:

```
Do
    hBoard = registerBoardEx(i)
    If hBoard >= 0 Then
        CardType = GetBoardModel(hBoard)
        If CardType = <Wanted> Then
            Exit Do ' Exit loop as we
                    ' have a valid board
        Else
            ' Free the unwanted board
            FreeBoard(hBoard)
        End If
    End If
    hBoard = -1 ' We don't have a suitable card
    ' Try the next board
    i = i + 1
Loop Until (i >= 8)
```

5) These lines of code will search through installed ADIO cards until the desired card type is found. The limit of 8 in the '`Loop Until (i >= 8)`' condition may be increased to 256 if using DIO_TC.DLL version 4.40 or later.

6) Put away the code window, and select the Form1 design window.

7) Select 'Window|Menu Design...' to bring up the dialog box from which you design the form's menubar. Type 'Exit' as the caption and 'mnuExit' as the name for the first menu bar item, then click on OK to put the dialog box away.

8) The menu bar will now appear in the Form1 design window. Click on the 'Exit' item to bring up the code window for the mnuExit_Click() subroutine. At runtime, this routine will be called whenever the 'Exit' menu is selected.

9) Type the following lines into the mnuExit_Click() subroutine, to un-registers the board from the DLL as the program closes.

```
Dim e As Short
e = FreeBoard( hBoard )
if e <> OK then   Call ReportError( e )
End If
```

These steps will create the shell of a VB application that can now be run. The program at this stage does nothing more than register a board with the DLL on start-up, and free that board on exit.

Section 6.4 describes the library functions available. Where arguments to functions are described as pointers, the address of a user-declared variable is required. Visual Basic normally does this anyway. The function declarations in DIO_TC.BAS uses the 'ByVal' prefix for all function arguments that are not passed as pointers. For example, function *TCgetCount* requires a pointer to a variable declared as long, into which the count value result will be placed. A typical VB code example for displaying the Z1 Counter 0 count value would be as follows:

```
Dim count As Long                ' declare count as long

i = TCgetCount( h, Z1, 0, count )' count now contains new
Text1.Text = Str$( count )       ' value
```

### 6.3.3 Delphi 3.0 Onwards

To open one of the Delphi example projects provided with the DLL, from within Borland Delphi 3.0 select 'File|Open ' and select one of the .DPR project files provided in the EX_DELPH subdirectory of the AmpDIO software directory. The project window will now appear on the desktop. Double-click on any file in the project to view the source code, or select Run to run the program.

To create your own Delphi program from scratch, perform the following steps:

1) From within Borland, select 'File|New Application'. A new project window will appear with an empty Form1.

2) Select View Project Manager and add DIO_TC.PAS from the EX_DELPH subdirectory of the AmpDIO software directory:

   DIO_TC.PAS  - DLL declarations and global constants

3) Double-click on the empty Form1 design window to bring up the code window for the Form_Load() subroutine. At runtime, this routine will get called when the program first starts up.

4) Type the following lines into the Form_Load() subroutine:

```
var
    i:smallint;
    cardtype:smallint;
begin
    // find a board using registerBoardEx which
    // doesn't need to know base address etc etc
    i := 0;
    repeat
        hboard := registerBoardEx(i);
```

```
            if hboard >= 0 then
            begin
                cardtype := GetBoardModel(hBoard);
                if cardType = <Wanted> then exit;
                FreeBoard(hBoard);
            end;
            hBoard := -1;
           inc(i);
        until (i >= 8);
    end;
```

These lines of code will search through installed ADIO cards until the desired card type is found. The limit of 8 in the '`until (i >= 8)`' condition may be increased to 256 if using DIO_TC.DLL version 4.40 or later.

5) Add the following global variable underneath the form class definition and global var statement:

```
var
    Form1: TForm1;
    hBoard:smallInt;
```

6) Type the following lines into the Form1Close() subroutine, to un-register the board from the DLL as the program closes.

```
if hboard >= 0 then
    FreeBoard(hboard);
```

These steps will create the shell of a Delphi application that can now be run. The program at this stage does nothing more than register a board with the DLL on start-up, and free that board on exit.

Section 6.4 describes the library functions available. Where arguments to functions are described as pointers, the address of a user-declared variable is required. This is taken care of automatically by Delphi because DIO_TC.PAS interface file insures that the variables will be passed correctly. For example, function *TCgetCount* requires a pointer to a variable declared as long, into which the count value result will be placed. A typical VB code example for displaying the Z1 Counter 0 count value would be as follows:

```
var
    count:LongInt;
begin
    TCgetCount( h, Z1, 0, count );
    // count now contains new value
    Label1.caption := InttoStr( count );
```

### 6.3.4 Visual Basic .NET

The Visual Basic .NET example projects can be opened from within the Microsoft Visual Studio .NET development environment by choosing 'File|Open|Project…' in the menu and selecting the appropriate .VBPROJ project file within the EX_VBNET subdirectory of the AmpDIO software directory. The project window will now appear on the desktop. Double-clicking on a .VBPROJ file from within Windows Explorer should also cause Microsoft Visual Studio .NET to run with the selected project file open. Double-click on any file in the project to view the source code or form design (if both are present, they are displayed under separate tabs). To run the example, press the F5 key or select 'Debug|Start' in the menu.

To create your own Visual Basic .NET program from scratch within Microsoft Visual Studio .NET, perform the following steps:

1) From within Microsoft Visual Studio .NET, select 'File|New|Project…' in the menu. In the 'New Project' dialog, choose 'Visual Basic Projects' in the 'Project Types' box and 'Windows

Application' in the 'Templates' box. Enter a name for the project and specify a location using the 'Browse…' button. Press the 'OK' button to continue.

2) Select 'File|Add Existing Item' from the menu and add DIO_TC.VB from the the EX_VBNET subdirectory of the AmpDIO software directory.

3) Close the Form1 design window and source code window (if present) and open the properties for the project, 'Project|<Project name> Properties'. Clear the 'Root Namespace' property to allow the DIO_TC.VB file to be used without modification. Press 'OK' to close the project properties dialog.

4) Select 'View|Solution Explorer' from the menu. Double click on the Form1.vb file in the Solution Explorer window to open the form design window. Double-click on the empty Form1 design window to bring up the code window for the Form1_Load() subroutine. At runtime, this routine will be called when the program first starts up.

5) At the top of the source code window, enter the following line to import the DIO_TC.DLL functions:

```
Imports Amplicon.AmpDIO.DIO_TC
```

6) Add the following code under '`Public Class Form1`' after the '`Inherits`' statements:

```
Dim hBoard As Short
```

7) Add the following code to the Form1_Load() subroutine:

```
Dim i As Short
Dim CardType As Short

Do
    hBoard = registerBoardEx(i)
    If hBoard >= 0 Then
        CardType = GetBoardModel(hBoard)
        If CardType = <Wanted> Then
            Exit Do ' Exit main Loop as we
                    ' have valid card
        Else
            ' Free the unwanted card
            FreeBoard (hBoard)
        End If
    End If
    hBoard = -1 ' We don't have a suitable card
    ' Try the Next card
    i = i + 1
Loop Until (i >= NUMBER_CARD_SUPPORTED)
```

8) While the cursor is in the Form1_Load subroutine, select the 'Closed' method from the drop-down selection box at the top right of the source code window and add the following code to the Form1_Closed subroutine:

```
If hBoard >= 0 Then
    FreeBoard(hBoard)
End If
```

These steps will create the shell of a VB.NET application that can now be run. The program at this stage does nothing more than register a board with the DLL on start-up, and free that board on exit.

Section 6.4 describes the library functions available. Where arguments to functions are described as pointers, the address of a user-declared variable is required. Visual Basic normally does this anyway. The function declarations in DIO_TC.VB uses the 'ByVal' prefix for all function arguments

that are not passed as pointers and the 'ByRef' prefix for function arguments that are passed as pointers. For example, function *TCgetCount* requires a pointer to a variable declared as long, into which the count value result will be placed. A typical VB.NET code example for displaying the Z1 Counter 0 count value would be as follows:

```
Dim count As Integer

i = TCgetCount(hBoard, Z1, 0, count)' count now contains new
Text1.Text = Str(count)              ' value
```

### 6.3.5 Visual C# .NET

The Visual C# .NET example projects can be opened from within the Microsoft Visual Studio .NET development environment by choosing 'File|Open|Project…' in the menu and selecting the appropriate .CSPROJ project file within the EX_C# subdirectory of the AmpDIO software directory. The project window will now appear on the desktop. Double-clicking on a .CSPROJ file from within Windows Explorer should also cause Microsoft Visual Studio .NET to run with the selected project file open. Double-click on any file in the project to view the source code or form design (if both are present, they are displayed under separate tabs). To run the example, press the F5 key or select 'Debug|Start' in the menu.

To create your own Visual C# .NET program from scratch within Microsoft Visual Studio .NET, perform the following steps:

1) From within Microsoft Visual Studio .NET, select 'File|New|Project…' in the menu. In the 'New Project' dialog, choose 'Visual C# Projects' in the 'Project Types' box and 'Windows Application' in the 'Templates' box. Enter a name for the project and specify a location using the 'Browse…' button. Press the 'OK' button to continue.

2) Select 'File|Add Existing Item' from the menu and add Dio_tc_h.CS from the the EX_C# subdirectory of the AmpDIO software directory.

3) On the main menu, select 'Project|<Project name> Properties...'. Select 'Configuration Properties|Build' on the left-hand panel. On the right-hand panel, set the 'Code Generation|Allow Unsafe Code Blocks' property to 'True'. Press the 'OK' button to close the project properties dialog.

4) Double-click on the empty Form1 design window to bring up the code window for the Form1_Load() subroutine. At runtime, this routine will be called when the program first starts up.

5) Near the top of the source code window, enter the following line to import the DIO_TC.DLL functions:

```
using Amplicon.AmpDIO;
```

6) Add the following declarations to the Form1 class before the 'main' function:

```
short hBoard;
```

7) Add the following code to the Form1_Load() function:

```
short i;
short CardType;

for (i = 0; i < DIO_TC.NUMBER_CARD_SUPPORTED; i++)
{
    hBoard = DIO_TC.registerBoardEx(i);
    if (hBoard >= 0)
    {
```

```
                CardType = DIO_TC.GetBoardModel(hBoard);
                if (CardType = <Wanted>)
                {
                    break; // Exit loop as we
                           // have valid card
                }
                else
                {
                    // Free the unwanted card
                    DIO_TC.FreeBoard(hBoard);
                }
            }
            hBoard = -1; // We don't have a suitable card
            // Try the Next card
        }
```

8) Go back to the form design window and press the F4 key to open the form's properties window. On the form properties, select 'Events' (represented by a 'lightning strike' icon). Double click on the 'Behavior|Closed' method to add the 'Form1_Closed' function to the source code window and add the following code to the function:

```
if (hBoard >= 0)
{
    DIO_TC.FreeBoard(hBoard);
}
```

These steps will create the shell of a C#.NET application that can now be run. The program at this stage does nothing more than register a board with the DLL on start-up, and free that board on exit.

Section 6.4 describes the library functions available. All functions and constants are part of the 'DIO_TC' class in the 'Amplicon.AmpDIO' namespace. Where arguments to functions are described as pointers, the address of a user-declared variable is required. For a simple variable, the '&' operator may be used, as for the C language. For example, function *TCgetCount* requires a pointer to a variable declared as long, into which the count value result will be placed. A typical C#.NET code example for displaying the Z1 Counter 0 count value would be as follows:

```
int count;

DIO_TC.TCgetCount(hBoard, Z1, 0, &count);
Text1.Text = count.ToString;
```

To pass a pointer to the first element of an array to one of the library functions, it is necessary to use a 'fixed' statement to prevent the array being relocated. For example, using the *TCdriveNCBufferUserInterrupt* function in some non-callback user interrupt code:

```
fixed (int *pData = &MyData[0])
{
    DIO_TC.TCDriveNCBufferUserInterrupt(hBoard, hIntr,
            (uint *)pData, &RetLength);
}
```

## 6.4 DIO_TC.DLL Library Functions

Details are given of each of the functions provided in the supplied Windows Dynamic Link Library (DIO_TC.DLL).

### 6.4.1 Initialization Functions

The board can not be registered with the library unless it has been correctly installed and configured in he system registry, using the configuration tool supplied (in the control panel).

#### 6.4.1.1 Register a Board with the Library — registerBoard

Requests a card matching a model number, base address and IRQ setting. This function returns a Board Handle greater than or equal to 0 on success, which must be used in all subsequent calls to library functions for this board.

```
i = registerBoard (model, ba, irq)
```

| | | |
|---|---|---|
| where | *model* | **short**: board's model number. The following pre-defined constants may be used for the boards supported:<br><br>PC212E = 212<br>PC214E = 214<br>PC215E = 215<br>PC272E = 272<br>PC218E = 218<br>PC263 = 263<br>PC272E = 272<br>PC36AT = 36<br>PC24E = 24<br>PC25E = 24 (25 may also be used in v4.30)<br>PC26AT = 26<br>PC27E = 27<br>PC30AT = 30<br>PCI215 = 215 (same as PC215E)<br>PCI224 = 224<br>PCI230 = 230<br>PCI234 = 234<br>PCI236 = 36 (236 may also be used)<br>PCI260 = 260<br>PCI263 = 263 (same as PC263)<br>PCI272 = 272 (same as PC272E) |
| | *ba* | **short**: board's base address. Factory default is 300 hex. See section 2.3 for details on selecting the board's base address. |
| | *irq* | **short**: board's Interrupt level. Factory default is 5. See section 2.3 for details on selecting the board's interrupt level. The value can be the real IRQ value or one of the following special values:<br><br>IRQ_ANY = –1: Match any (or no) IRQ<br>IRQ_NONE = 255: Match IRQ-less card |
| Returns **short**: | Board handle to be used in all subsequent function calls for that board. | |
| or | ERRSUPPORT<br>ERRBASE<br>ERRIRQ | |

| | |
|---|---|
| Prior Calls | none |
| See Also | registerBoardEx<br>registerBoardPci<br>FreeBoard |

#### 6.4.1.2 Extended Register Board Function — registerBoardEx

Requests use of a card at a specified slot number. This function returns a Board Handle greater than or equal to 0 on success.

SUPPORTED IN VERSION 2.00 ONWARDS.

```
i = registerBoardEx (CardNo);
```

| | | |
|---|---|---|
| where | *CardNo* | **short**: card 'Slot Number'. The slot number can be determined by looking at the order cards are installed in the system. I.E. if there are two cards listed in the device manager or control panel applet and the wanted card is listed first, its slot number is 0. |
| Returns **short**: | Board handle to be used in all subsequent function calls for that board. | |
| or | ERRSUPPORT<br>ERRBASE<br>ERRIRQ | |
| Prior Calls | none | |
| See Also | registerBoard<br>registerBoardPci<br>FreeBoard | |

#### 6.4.1.3 Register a PCI Board by Model, Bus and Slot Position — registerBoardPci

Requests use of a PCI card matching a model number, PCI bus and PCI slot number (determined by which PCI slot the card is plugged into). This function returns a Board Handle greater than or equal to 0 on success.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = registerBoardPci (model, bus, slot);
```

| | | |
|---|---|---|
| where | *model* | **short**: board's model number. The following pre-defined constants may be used for the boards supported:<br><br>PCI215 = 215<br>PCI224 = 224<br>PCI230 = 230<br>PCI234 = 234<br>PCI236 = 36 (236 may also be used)<br>PCI260 = 260<br>PCI263 = 263 |

| | | |
|---|---|---|
| | | PCI272 = 272 |
| | *bus* | **short**: PCI bus number (usually 0). |
| | *slot* | **short**: PCI slot number. |
| Returns **short**: | Board handle to be used in all subsequent function calls for that board. | |
| or | ERRSUPPORT | |
| Prior Calls | none | |
| See Also | registerBoard<br>registerBoardPci<br>FreeBoard | |

#### 6.4.1.4 Get the Model Number of a Board — GetBoardModel

Returns the model number of a registered board. N.B. where a PCI board shares a model number with an ISA board, they may be distinguished by calling the GetBoardPciPosition function, which will return an error for an ISA board.

The only oddities in the returned model numbers are 25 (which can be a PC24E or PC25E) and 36 (which can be a PC36AT or a PCI236).

The PCI230+ and PCI260+ may be distinguished from the older PCI230 and PCI260 models by calling the DIO_TC_hardwareVersion function, supported in verssion 4.42 onwards.

```
i = GetBoardModel (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | Board's model number. Possible values are:-<br><br>212: Amplicon PC212E<br>214: Amplicon PC214E<br>215: Amplicon PC215E or PCI215<br>218: Amplicon PC218E<br>263: Amplicon PC263 or PCI263<br>272: Amplicon PC272E or PCI272<br>36: Amplicon PC36AT or PCI236<br>24: Amplicon PC24E or PC25E<br>26: Amplicon PC26AT<br>27: Amplicon PC27E<br>30: Amplicon PC30AT<br>230: Amplicon PCI230 or PCI230+<br>260: Amplicon PCI260 or PCI260+<br>224: Amplicon PCI224<br>234: Amplicon PCI234 | |
| or | ERRHANDLE | |
| Prior Calls | registerBoardEx | |
| See Also | GetBoardBase | |

| | |
|---|---|
| | GetBoardIRQ<br>GetBoardPciPosition<br>DIO_TC_hardwareVersion |

#### 6.4.1.5 Get Board Base Address — GetBoardBase

Gets the base-address setting of a board as set in the registry.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = GetBoardBase (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | Base address. | |
| or | 0 for invalid board handle | |
| Prior Calls | registerBoardEx | |
| See Also | GetBoardModel<br>GetBoardIRQ<br>GetBoardPciPosition | |

#### 6.4.1.6 Get Board IRQ — GetBoardIRQ

Gets the IRQ setting of a board as set in the registry.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = GetBoardIRQ (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | IRQ setting. | |
| or | 255 (= IRQ_NONE) if board handle invalid or board set-up without IRQ. | |
| Prior Calls | registerBoardEx | |
| See Also | GetBoardModel<br>GetBoardBase<br>GetBoardPciPosition | |

#### 6.4.1.7 Get Board PCI Bus Position — GetBoardPciPosition

Gets the PCI bus and slot number for a PCI card. If the card is an ISA card or the information is not available from the driver, the bus and slot values read will be –1 and the function will return ERRSUPPORT.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = GetBoardPciPosition (h, pbus, pslot)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pbus* | **pointer to short**: pointer to variable where PCI bus number result will be stored. |
| | *pslot* | **pointer to short**: pointer to variable where PCI slot number result will be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRSUPPORT<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | GetBoardModel<br>GetBoardBase<br>GetBoardIRQ | |

#### 6.4.1.8 Unregister a Board — FreeBoard

Frees a previously registered board, allowing it to used by another program.

```
i = FreeBoard (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |
| Prior Calls | registerBoardEx | |
| See Also | | |

#### 6.4.1.9 Get Driver Version — DIO_TC_driverVersion

Gets version number from driver using IOCTL_QUERY_VERSION.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = DIO_TC_driverVersion (h, pver)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | pver | **pointer to unsigned long**: pointer to variable where driver version result will be stored. The driver version result is set according to the driver version number if supported by the driver (driver versions 4.02 onwards) or to 0 if unsupported. The value is formatted as follows: |

| | | |
|---|---|---|
| | | Bits 31 to 24: Major version<br>Bits 23 to 16: Minor version<br>Bits 15 to 0: All zero. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRSUPPORT<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | DIO_TC_dllVersion<br>DIO_TC_hardwareVersion | |

#### 6.4.1.10 Get DLL Version — DIO_TC_dllVersion

Gets value of the DIO_TC_VERSION macro in DIO_TC.H at the time the DLL was compiled.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = DIO_TC_dllVersion ()
```

| | | |
|---|---|---|
| Returns **short**: | (256*major)+minor<br><br>e.g. 4.02 becomes ((4*256)+2) = 1026. | |
| Prior Calls | none | |
| See Also | DIO_TC_driverVersion<br>DIO_TC_hardwareVersion | |

#### 6.4.1.11 Get Hardware Version — DIO_TC_hardwareVersion

Gets the version number of the board from the driver using IOCTL_QUERY_HWVERSION.

SUPPORTED IN VERSION 4.42 ONWARDS.

PCI230+ and PCI260+ can be distinguished from PCI230 and PCI260 by driver version 4.42 or later.

From version 4.44 onwards, the reported hardware version is limited to the maximum value known by the driver and DLL for the card. This allows an application to check which extra card features are supported by the overall system.

```
i = DIO_TC_hardwareVersion (h, pver)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | pver | **pointer to unsigned long**: pointer to variable where hardware version result will be stored. By default, it is set to 0. If the driver detects an enhanced version of a board, it is set to a number greater than 0.<br><br>For PCI230+ and PCI260+, the version is at |

| | | |
|---|---|---|
| | | least 1. For the older PCI230 and PCI260, the version is 0. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | DIO_TC_driverVersion<br>DIO_TC_dllVersion<br>DIO_TC_realHardwareVersion | |

#### 6.4.1.12 Get Real Hardware Version — DIO_TC_realHardwareVersion

Gets the real version number of the board from the driver using IOCTL_QUERY_REALHWVERSION.

SUPPORTED IN VERSION 5.02 ONWARDS.

This is like DIO_TC_hardwareVersion except that the reported value is not limited to the maximum value supported by the driver and DLL for the card. This allows an application to check whether the card has a particular feature or bug-fix that is not necessarily supported by the current driver or DLL.

If the function returns ERRSUPPORT, the real hardware version could not be determined due to the driver version in use, but the reported version will be set to the value reported by DIO_TC_hardwareVersion instead.

```
i = DIO_TC_realHardwareVersion (h, pver)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | pver | **pointer to unsigned long**: pointer to variable where hardware version result will be stored. By default, it is set to 0. If the driver detects an enhanced version of a board, it is set to a number greater than 0.<br><br>For PCI230+ and PCI260+, the version is at least 1. For the older PCI230 and PCI260, the version is 0. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | DIO_TC_hardwareVersion | |

#### 6.4.1.13 Control Hardware Reinitialization — DIO_TC_SetResetOnRegister

Controls whether or not the board registration functions reinitialize the hardware on the board being registered. By default, the board registration functions do reinitialize the hardware.

SUPPORTED IN VERSION 4.40 ONWARDS.

Initialization involves setting PPI ports to input mode (using 8255 mode 0), any other digital outputs to zero, ADC channels to bipolar, single-ended mode and maximum input range, ADC multiplexer to channel 0, ADC conversion trigger source to software trigger, DAC channels to bipolar with minimum output range and all DAC outputs set to 0 (which generally produces 0V output). Note that timer counter channels and counter clock and gate connections are not initialized.

Note that regardless of the DLL reinitializing the hardware on the board being registered, versions of the hardware device driver prior to version 4.40 also reinitialize the hardware on a board everytime it is opened. Version 4.40 of the hardware device driver only initializes the hardware once before it is opened for the first time. The registerBoard and registerBoardPci functions may open several unopen boards temporarily while looking for the one specified by the function's parameters. The DLL only initializes the hardware on at most one of these boards (the one referred to by the returned board handle), but older versions of the device driver will reinitialize the hardware on each of them.

```
DIO_TC_SetResetOnRegister (fROR)
```

| | | |
|---|---|---|
| where | *fROR* | **short**: controls whether or not subsequent calls to the board registration functions will reinitialize the hardware on the board referred to by their returned board handles:<br><br>0 = Do not reinitialize hardware<br>other = Reinitialize hardware (default) |
| Returns **void**. | | |
| Prior Calls | none | |
| See Also | DIO_TC_GetResetOnRegister<br>registerBoard<br>registerBoardEx<br>registerBoardPci | |

#### 6.4.1.14 Check Whether Hardware Will be Reinitialized — DIO_TC_GetResetOnRegister

Indicates whether or not the board registration functions will reinitialize the hardware on the board being registered. See the description of the DIO_TC_SetResetOnRegister function for more details.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = DIO_TC_GetResetOnRegister ()
```

| | |
|---|---|
| Returns **short**: | Value indicating whether or not subsequent calls to the board registration functions will reinitialize the hardware on the board:<br><br>0 = Do not reinitialize hardware<br>1 = Reinitialize hardware (default) |

| | |
|---|---|
| Prior Calls | none |
| See Also | DIO_TC_SetResetOnRegister<br>registerBoard<br>registerBoardEx<br>registerBoardPci |

### 6.4.2 Interrupt Control Functions

#### 6.4.2.1 Enable a Board's Interrupts — enableInterrupts

Enables the card's interrupts to be processed by the driver. This enables interrupts at the first level. A card's interrupt sources are active when enabled at both the first level and the second level and have been set up using one of the interrupt set-up functions such as TCsetEventRecorder or TCsetUserInterrupt.

Interrupts are initially disabled at the first level.

```
i = enableInterrupts (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |
| Prior Calls | registerBoardEx | |
| See Also | disableInterrupts<br>interruptsEnabledP<br>setIntMask<br>getIntMask | |

#### 6.4.2.2 Disable a Board's Interrupts — disableInterrupts

Disables the card's interrupts. This disables interrupts at the first level. Any active interrupt sources which have been enabled at the second level and set-up using one of the interrupt set-up functions such as TCsetEventRecorder or TCsetUserInterrupt will cease to operate.

Interrupts are initially disabled at the first level.

```
i = disableInterrupts (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |
| Prior Calls | registerBoardEx | |
| See Also | enableInterrupts<br>interruptsEnabledP<br>setIntMask | |

| getIntMask |
|---|

#### 6.4.2.3 Check whether a Board's Interrupts are Enabled — interruptsEnabledP

Reports whether the card's interrupts are enabled or not at the first level.

SUPPORTED IN VERSION 4.40 ONWARDS

Interrupts are initially disabled at the first level.

```
i = interruptsEnabledP (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | FALSE (= 0) if interrupts are not enabled at the first level<br>TRUE (= 1) if interrupts are enabled at the first level | |
| Prior Calls | registerBoardEx | |
| See Also | enableInterrupts<br>disableInterrupts | |

#### 6.4.2.4 Enable a Board's Interrupt Source(s) — setIntMask

Enables or disables interrupt sources. This provides a second level of enabling and disabling, the first level being provided by enableInterrupts and disableInterrupts. Each supported card has up to six interrupt sources. This function says which ones should be enabled and which ones disabled. An interrupt source is active when it is enabled at the first level (enableInterrupts) and at the second level and has been set-up (TCsetEventRecorder, TCsetUserInterrupt, etc.).

In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level, but are automatically enabled by the interrupt set-up functions.

In versions of the DLL from version 4.40 onwards, all valid interrupt sources are initially enabled at the second level and are no longer automatically enabled by the interrupt set-up functions. This allows an interrupt source to be set up without enabling it at the second level.

```
i = setIntMask (h, mask)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *mask* | **short**: mask bits. Bits 0 to 5 correspond to the six possible interrupt sources. Set a bit to '1' to enable and to '0' to disable an interrupt source. For cards with an interrupt enable (IE) register, these bits correspond with the matching bits in the IE register. The interrupt source 'chip' parameter used in the interrupt set-up functions corresponds to the interrupt mask bit position multiplied by 4 (e.g. 'chip' = 8 corresponds to mask bit position of 2 and a mask value of $2^2 = 100_2 = 4$). The bit positions for the board's interrupt sources will vary from board to board. Refer to individual card |

| | | |
|---|---|---|
| | | manuals for a description of the interrupt sources, and their functionality. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | getIntMask<br>TCenableInterruptChip<br>TCdisableInterruptChip<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.2.5 Check Which Interrupt Sources are Enabled — getIntMask

Reports which interrupt sources are enabled at the second level. The return value is a bit mask. Interrupt sources that are enabled are not necessarily active. An interrupt source is active if it is enabled at the first and second levels and is set up.

SUPPORTED IN VERSION 4.40 ONWARDS

```
mask = getIntMask (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | Mask bits. Bits 0 to 5 correspond to the six possible interrupt sources. A bit set to '1' indicates that the corresponding interrupt source is enabled, but not necessarily active. For cards with an interrupt enable (IE) register, these bits correspond with the matching bits in the IE register. The interrupt source 'chip' parameter used in the interrupt set-up functions corresponds to the interrupt mask bit position multiplied by 4 (e.g. 'chip' = 8 corresponds to mask bit position of 2 and a mask value component of $2^2 = 100_2 = 4$). Refer to individual card manuals for a description of the interrupt sources, and their functionality. The returned value is the sum of the mask value components for each interupt source enabled at the second level. The bit positions for the board's interrupt sources will vary from board to board. | |
| Prior Calls | registerBoardEx | |
| See Also | setIntMask | |

#### 6.4.2.6 Read Interrupt Status Register — getIntStat

Reads a card's interrupt status register if it has one. This is not very useful.

```
i = getIntStat (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | interrupt status word (>= 0) | |

| | |
|---|---|
| or | ERRHANDLE |
| Prior Calls | registerBoardEx |
| See Also | |

#### 6.4.2.7 Enable an Individual Interrupt Source — TCenableInterruptChip

Enables an interrupt source at the second level if it is a valid interrupt source for the card. An interrupt source is active when it is enabled at the first level (enableInterrupts) and at the second level and has been set-up (TCsetEventRecorder, TCsetUserInterrupt, etc.).

SUPPORTED IN VERSION 4.40 UPWARDS

In versions of the DLL from version 4.40 onwards, all valid interrupt sources are initially enabled at the second level.

```
i = TCenableInterruptChip (h, Chip)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*Chip* — **short**: interrupt source as used in the user interrupt set-up functions. For historic reasons, this corresponds to a bit position in the interrupt enable (IE) mask, multiplied by 4. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

| | |
|---|---|
| Returns **short**: | OK |
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx |
| See Also | TCdisableInterruptChip<br>setIntMask<br>getIntMask<br>TCenableUserInterrupt |

#### 6.4.2.8 Disable an Individual Interrupt Source — TCdisableInterruptChip

Disables an interrupt source at the second level if it is a valid interrupt source for the card. If the interrupt source is active, it will be deactivated.

SUPPORTED IN VERSION 4.40 UPWARDS

In versions of the DLL from version 4.40 onwards, all valid interrupt sources are initially enabled at the second level.

```
i = TCdisableInterruptChip (h, Chip)
```

where *h* — **short**: board handle as issued by the registerBoardEx function.

*Chip* — **short**: interrupt source as used in the user interrupt set-up functions. For historic reasons, this corresponds to a bit position in the interrupt enable (IE) mask, multiplied by 4. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

Returns **short**: OK

or ERRHANDLE
ERRCHAN

Prior Calls: registerBoardEx

See Also: TCenableInterruptChip
setIntMask
getIntMask
TCdisableUserInterrupt

### 6.4.3 Thread Priority Control

#### 6.4.3.1 Set Real Time Priority — DIO_TC_getrealtimepriority

Puts current process and thread into real time priority. Prior to version 4.23, this was only done for Windows NT. For version 4.23 onwards it is also done for Windows 9x.

```
i = DIO_TC_getrealtimepriority ()
```

Returns **BOOLEAN**: TRUE on success

| | |
|---|---|
| or | FALSE on failure |
| Prior Calls | none |
| See Also | |

#### 6.4.3.2 Set Normal Priority — DIO_TC_restorenormalpriority

Puts current process and thread back to normal priority. Prior to version 4.23, this was only done for Windows NT. For version 4.23 onwards it is also done for Windows 9x.

```
i = DIO_TC_restorenormalpriority ()
```

| | |
|---|---|
| Returns **BOOLEAN**: | TRUE on success |
| or | FALSE on failure |
| Prior Calls | none |
| See Also | |

#### 6.4.3.3 Get Priority of User Interrupt Thread — TCgetInterruptThreadPriority

Gets the Win32 thread priority value used for the user interrupt callback thread. May be used even if the user interrupt thread is not currently running (e.g. has not been enabled).

When the interrupt set-up function is called, the user interrupt thread priority is initialized to the priority of the calling thread. Calls to TCsetInterruptThreadPriority change this value. The function gets the value set by the interrupt set-up function or TCsetInterruptThreadPriority, even if the priority of the user interrupt thread has been changed by some other mechanism in the meantime.

SUPPORTED IN VERSION 4.23 ONWARDS.

```
i = TCgetInterruptThreadPriority (h, hUsrInt, pPriority)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| | *pPriority* | **pointer to int**: pointer to (long) integer variable which will be set to the priority of the user interrupt thread. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2 | |

| | |
|---|---|
| | TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2 |
| See Also | TCsetInterruptThreadPriority |

#### 6.4.3.4 Set Priority of User Interrupt Thread — TCsetInterruptThreadPriority

Sets the Win32 thread priority value used for the user interrupt callback thread. May be used even if the user interrupt thread is not currently running (e.g. has not been enabled).

When the interrupt set-up function is called, the user interrupt thread priority is initialized to the priority of the calling thread. This function may be used to change that value. If the user interrupt callback thread is currently running, its priority will be changed immediately. If the user interrupt has not been enabled yet, this priority value will be used when the user interrupt callback thread is created when the user interrupt is enabled (usually by enableInterrupts).

The function may not be used with non-callback user interrupts, as no separate thread is created to handle those. The function returns ERRSUPPORT if used with a non-callback user interrupt or if the function failed to change the priority of a running user interrupt callback thread.

SUPPORTED IN VERSION 4.23 ONWARDS.

```
i = TCsetInterruptThreadPriority (h, hUsrInt, Priority)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| | *Priority* | **int**: standard Win32 thread priority value to use. Values defined in the WIn32 SDK are as follows:<br><br>THREAD_PRIORITY_IDLE = –15<br>THREAD_PRIORITY_LOWEST = –2<br>THREAD_PRIORITY_BELOW_NORMAL = –1<br>THREAD_PRIORITY_NORMAL = 0<br>THREAD_PRIORITY_ABOVE_NORMAL = 1<br>THREAD_PRIORITY_HIGHEST = 2<br>THREAD_PRIORITY_TIME_CRITICAL = 15 |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2 | |

| | |
|---|---|
| See Also | TCgetInterruptThreadPriority<br>enableInterrupts |

### 6.4.4 Data Buffer Functions

#### 6.4.4.1 Allocate a Short Integer Data Buffer — allocateIntegerBuf

Creates a data buffer, by allocating a block of memory of short integer (16-bit) data. The function returns a Buffer Handle (>= 0). The Buffer Handle must be used in any subsequent function calls to identify that particular data buffer.

```
b = allocateIntegerBuf (nItems)
```

| | | |
|---|---|---|
| where | *nItems* | **long**: number of data items to be allocated. If there is insufficient memory available for the size of the buffer, an error is returned. |
| Returns **short**: | Buffer Handle (>= 0). This handle must be used in all subsequent function calls to identify the buffer. | |
| or | ERRSUPPORT<br>ERRMEMORY | |
| Prior Calls | none | |
| See Also | freeIntegerBuf | |

#### 6.4.4.2 Allocate a Long Integer Data Buffer — allocateLongBuf

Creates a data buffer, by allocating a block of memory of long integer (32-bit) data. The function returns a Buffer Handle (>= 0). The Buffer Handle must be used in any subsequent function calls to identify that particular data buffer. A long integer data buffer is required by the function TCsetEventRecorder

```
b = allocateLongBuf (nItems)
```

| | | |
|---|---|---|
| where | *nItems* | **long**: Number of data items to be allocated. If there is insufficient memory available for the size of the buffer, an error is returned. |
| Returns **short**: | Buffer Handle (>= 0). This handle must be used in all subsequent function calls to identify the buffer. | |
| or | ERRSUPPORT<br>ERRMEMORY | |
| Prior Calls | none | |
| See Also | freeLongBuf | |

#### 6.4.4.3 Free up a Short Integer Data Buffer — freeIntegerBuf

Frees a block of memory previously allocated for the given data buffer by the allocateIntegerBuf function.

```
i = freeIntegerBuf (b)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateIntegerBuf function. |
| Returns **short**: | OK | |
| or | ERRBUFFER | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.4 Free up a Long Integer Data Buffer — freeLongBuf

Frees a block of memory previously allocated for the given data buffer by the allocateLongBuf function.

```
i = freeLongBuf (b)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateLongBuf function. |
| Returns **short**: | OK | |
| or | ERRBUFFER | |
| Prior Calls | allocateLongBuf | |
| See Also | | |

#### 6.4.4.5 Read Data from a Short Integer Buffer — readIntegerBuf

Reads a data item from a short integer buffer, returning the item via a user-supplied pointer. The pointer must reference a short integer variable.

```
i = readIntegerBuf (b, item, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle, as issued by the allocateIntegerBuf function |
| | *item* | **long**: index of the data item in the buffer. |
| | *p* | **pointer to short**: points to a short integer variable to be used for the result. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.6 Read Data from a Long Integer Buffer — readLongBuf

Reads a data item from a long integer buffer, returning the item via a user-supplied pointer. The pointer must reference a long integer variable.

```
i = readLongBuf (b, item, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle, as issued by the allocateLongBuf function |
| | *item* | **long**: index of the data item in the buffer. |
| | *p* | **pointer to long**: points to a long integer variable to be used for the result. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE | |
| Prior Calls | allocateLongBuf | |
| See Also | | |

#### 6.4.4.7 Write Data to a Short Integer Buffer — writeIntegerBuf

Writes a single short integer data item to a short integer buffer.

```
i = writeIntegerBuf (b, item, data)
```

| | | |
|---|---|---|
| where | b | **short**: buffer handle, as issued by the allocateIntegerBuf function. |
| | item | **long**: index of item in buffer. |
| | data | **short**: data value. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.8 Write Data to a Long Integer Buffer — writeLongBuf

Writes a single long integer data item to a long integer buffer.

```
i = writeLongBuf (b, item, data)
```

| | | |
|---|---|---|
| where | b | **short**: buffer handle, as issued by the allocateLongBuf function. |
| | item | **long**: index of item in buffer. |

| | | |
|---|---|---|
| | data | **long**: data value. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | allocateLongBuf | |
| See Also | | |

#### 6.4.4.9 Copy a Block of Data to a Short Integer Buffer — copyToIntegerBuf

Copies a block of short integer data to a short integer buffer.

```
i = copyToIntegerBuf (b, start, nItems, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateIntegerBuf function. |
| | *start* | **long**: index of the starting item in the buffer. |
| | *nItems* | **long**: number of items to copy. |
| | *p* | **pointer to short**: pointer to the beginning of the memory block to copy. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.10 Copy a Block of Data to a Long Integer Buffer — copyToLongBuf

Copies a block of long integer data to a long integer buffer.

```
i = copyToLongBuf (b, start, nItems, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateLongBuf function. |
| | *start* | **long**: index of the starting item in the buffer. |
| | *nItems* | **long**: number of items to copy. |
| | *p* | **pointer to long**: pointer to the beginning of the memory block to copy. |
| Returns **short**: | OK | |
| or | ERRBUFFER | |

| | | |
|---|---|---|
| | ERRRANGE<br>ERRDATA | |
| Prior Calls | allocateLongBuf | |
| See Also | | |

#### 6.4.4.11 Copy a Block of Data from a Short Integer Buffer — copyFromIntegerBuf

Copies a segment of a short integer data buffer to a block of memory.

```
i = copyFromIntegerBuf (b, start, nItems, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateIntegerBuf function. |
| | *start* | **long**: index of the starting item in the buffer. |
| | *nItems* | **long**: number of items to copy. |
| | *p* | **pointer to short**: pointer to the beginning of the short integer memory block to which data is to be copied. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.12 Copy a Block of Data from a Long Integer Buffer — copyFromLongBuf

Copies a segment of a long integer data buffer to a block of memory.

```
i = copyFromLongBuf (b, start, nItems, p)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateLongBuf function. |
| | *start* | **long**: index of the starting item in the buffer. |
| | *nItems* | **long**: number of items to copy. |
| | *p* | **pointer to long**: pointer to the beginning of the long integer memory block to which data is to be copied. |
| Returns **short**: | OK | |
| or | ERRBUFFER<br>ERRRANGE<br>ERRDATA | |

| | |
|---|---|
| Prior Calls | allocateLongBuf |
| See Also | |

#### 6.4.4.13 Query Current Interrupt Position within a Short Integer Data Buffer — getIntegerIntItem

Gets the current interrupt position within a short integer data buffer. None of the currently supported interrupt functions in the library use short integer data buffers, so the returned position will be 0.

```
i = getIntegerIntItem (b, pitem)
```

| | | |
|---|---|---|
| where | *b* | **short**: buffer handle as issued by the allocateIntegerBuf function. |
| | *pitem* | **pointer to long**: pointer to a long integer variable, into which the result (the index of the buffer item to be used on the next interrupt) will be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRDATA | |
| Prior Calls | allocateIntegerBuf | |
| See Also | | |

#### 6.4.4.14 Query Current Interrupt Position within a Long Integer Data Buffer — getLongIntItem

Gets the current interrupt position within a long integer data buffer. This function can be called for any data buffer currently being used for Event Recorder data. In this case, data is written to the buffer when the interrupt occurs. This function returns the index within the specified buffer of the data item to be read or written to on the next interrupt, giving an indication of how much of the buffer contains valid data. The position is reset when the Event Recorder is set up.

```
i = getLongIntItem (b, pitem)
```

| | | |
|---|---|---|
| where | b | **short**: buffer handle as issued by the allocateLongBuf function. |
| | *pitem* | **pointer to long**: pointer to a long integer variable into which the result (the index of the buffer item to be used on the next interrupt) will be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRDATA | |
| Prior Calls | allocateLongBuf | |
| See Also | | |

### 6.4.5 Basic Timer/Counter Functions

#### 6.4.5.1 Test if Timer/Counter is Free — TCisAvailable

Checks if a particular timer/counter channel is currently available on a board. A counter/timer may not be available for one of two reasons:
1. the counter/timer is not provided by the board specified, or
2. the counter/timer is being used by some other function.

```
i = TCisAvailable (h, chip, chan)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as previously issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip, i.e. 0, 1, or 2. |
| Returns **short**: | 0 = Timer/counter NOT available, 1 = Available | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreeResource | |

#### 6.4.5.2 Free-up Timer/Counter — TCfreeResource

Frees a timer/counter channel previously reserved for use by one of the following functions:

TCsetMonoShot
TCgenerateAccFreq

```
i = TCfreeResource (h, chip, chan)
```

| | | |
|---|---|---|
| where | h | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12 |

| | | |
|---|---|---|
| | | Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | | |

#### 6.4.5.3 Connect Timer/Counter Clock Source — TCsetClock

Configures a timer/counter clock input source.

```
i = TCsetClock (h, chip, chan, clk)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *clk* | **short**: clock source. One of the following pre-defined constants representing the valid clock sources may be used:<br><br>CLK_CLK = 0: external CLK(*chan*) i/p<br>CLK_10MHZ = 1: internal 10 MHz<br>CLK_1MHZ = 2: internal 1 MHz<br>CLK_100KHZ = 3: internal 100 kHz<br>CLK_10KHZ = 4: internal 10 kHz<br>CLK_1KHZ = 5: internal 1 kHz<br>CLK_OUTN_1 = 6: OUT(*chan*–1)<br>CLK_EXT = 7: external EXTCLK(*chip*) i/p |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |

| | |
|---|---|
| See Also | TCgetClock<br>TCgetLinkedClockChannel<br>TCsetGate |

#### 6.4.5.4 Get Connected Timer/Counter Clock Source — TCgetClock

Gets a timer/counter channel's currently connected clock source, if it has been configured. This is not supported on PC214E or other models that have no clock connection registers.

Another function such as TCsetClock or one of the higher-level timer/counter functions has to set the clock source after registering the board in order for this function to return the current setting. If the current clock setting is not known, the function will return ERRSUPPORT.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCgetClock (h, chip, chan, pclk)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pclk* | **pointer to short**: pointer to a short integer variable into which the current clock source is to be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetClock | |

#### 6.4.5.5 Get Linked Clock Channel — TCgetLinkedClockChannel

Determines the GAT_OUTN_2 source channel for a specified timer counter channel.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCgetLinkedClockChannel (h, chip, chan, pClkChip, pClkChan)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pClkChip* | **pointer to short**: pointer to a short integer variable into which the address offset of the timer/counter chip containing the linked channel is to be stored. |
| | *pClkChan* | **pointer to short**: pointer to a short integer variable into which the channel number of the linked timer/counter channel is to be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCgetLinkedGateChannel<br>TCsetClock | |

#### 6.4.5.6 Connect Timer/Counter Gate Source — TCsetGate

Configures a timer/counter gate input source.

Gate input sources 4 to 7 are implemented on the PCI230+ and PCI260+.

```
i = TCsetGate (h, chip, chan, gate)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16 |

| | | |
|---|---|---|
| | | Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *gate* | **short**: gate source (0 to 7). One of the following pre-defined constants may be used:<br><br>GAT_VCC = 0: Enabled<br>GAT_GND = 1: Disabled<br>GAT_EXT = 2: GAT(*chan*) — external i/p<br>GAT_OUTN_2 = 3: /OUT(*chan*–2)<br><br>Note: for PCI230 and PCI230+, GAT(*chan*) input is PPI-X C(*chan*). For PCI260+, GAT(*chan*) is EXTTRIG. For the original PCI260, GAT(*chan*) is not connected.<br><br>PCI230+ and PCI260+ support the following additional constants:<br><br>GAT_LATCHED_EXT = 4: Latched GAT(*chan*) — goes high on rising edge of GAT(*chan*)<br>GAT_LATCHED_NOT_EXT = 5: Latched /GAT(*chan*) — goes high on falling edge of GAT(*chan*)<br>GAT_NOT_EXT = 6: /GAT(chan) — inverted external i/p<br>GAT_UNINV_OUTN_2 = 7: OUT(*chan*–2) |
| Returns **short**: | OK | |
| or | ERRSUPPORT<br>ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCgetGate<br>TCgetLinkedGateChannel<br>TCsetClock | |

#### 6.4.5.7 Get Connected Timer/Counter gate Source — TCgetGate

Gets a timer/counter channel's currently connected gate source, if it has been configured. This is not supported on PC214E or other models that have no gate connection registers.

Another function such as TCsetGate or one of the higher-level timer/counter functions has to set the gate source after registering the board in order for this function to return the current setting. If the current gate setting is not known, the function will return ERRSUPPORT.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCgetGate (h, chip, chan, pgate)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the |

| | | |
|---|---|---|
| | | registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pgate* | **pointer to short**: pointer to a short integer variable into which the current gate source is to be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetGate | |

#### 6.4.5.8 Get Linked Gate Channel — TCgetLinkedGateChannel

Determines the GAT_OUTN_2 source channel for a specified timer counter channel.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCgetLinkedGateChannel (h, chip, chan, pGatChip, pGatChan)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pGatChip* | **pointer to short**: pointer to a short integer variable into which the address offset of the |

| | | |
|---|---|---|
| | | timer/counter chip containing the linked channel is to be stored. |
| | *pGatChan* | **pointer to short**: pointer to a short integer variable into which the channel number of the linked timer/counter channel is to be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCgetLinkedClockChannel<br>TCsetGate | |

#### 6.4.5.9 Configure Timer/Counter Mode — TCsetMode

Sets a timer counter to one of its six available modes of operation. Reading and loading of count values by LSB followed by MSB is selected, as is a 16-bit binary count.

```
i = TCsetMode (h, chip, chan, mode)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *mode* | **short**: counter mode (0 to 5). See the 82C54 data sheet. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetCount<br>TCsetClock<br>TCsetGate<br>TCgetStatus<br>TCgetMode | |

#### 6.4.5.10 Read Timer/Counter Status — TCgetStatus

Returns the mode and status of a timer/counter by performing a read-back operation on the channel. This does not work on PC24E, PC25E, PC26AT, PC27E or PC30AT as they have the wrong sort of timer/counter chip.

```
i = TCgetStatus (h, chip, chan)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| Returns **short**: | Timer counter status byte. See the 82C54 data sheet for details. | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetMode<br>TCsetCount | |
| See Also | TCgetCount<br>TCgetMode | |

#### 6.4.5.11 Get Timer/Counter Mode — TCgetMode

Gets a timer/counter channel's current mode, if it has been configured.

Normally, this returns the last mode configured by a function such as TCsetMode or one of the higher-level timer/counter functions. If the mode has not been set since the board was registered, the timer/counter chip will be queried if this is supported. (Querying the mode from the timer/counter chip is not supported on PC24E, PC25E, PC26AT, PC27E and PC30AT.) If the timer/counter chip reports anything other than a 16-bit binary counter mode, this is considered invalid. The function returns ERRSUPPORT if no mode has been set and a valid mode cannot be read from the timer/counter chip.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCgetMode (h, chip, chan, pmode)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |

| | | |
|---|---|---|
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pmode* | **pointer to short**: pointer to a short integer variable into which the current counter mode is to be stored. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetMode<br>TCgetStatus | |

#### 6.4.5.12 Set Timer Count Value — TCsetCount

Sends a 16-bit count value to a timer/counter.

```
i = TCsetCount (h, chip, chan, count)
```

| | | |
|---|---|---|
| where | h | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *count* | **long**: 16-bit Count value. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |

| | |
|---|---|
| | ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx<br>TCsetMode |
| See Also | TCgetCount<br>TCsetClock<br>TCsetGate |

#### 6.4.5.13 Set two Timer Count Values — TCsetCounts

Sends two 16-bit count values to two timer/counters.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = TCsetCounts (h, chip1, chan1 count1, chip2, chan2, count2)
```

| | | |
|---|---|---|
| where: | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip1* | **short**: address offset of the timer/counter chip #1. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan1* | **short**: timer/counter #1 channel number within the chip (0, 1 or 2). |
| | *count1* | **long**: 16-bit Count value #1. |
| | *chip2* | **short**: address offset of the timer/counter chip #2. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan2* | **short**: timer/counter #2 channel number within the chip (0, 1 or 2). |
| | *count2* | **long**: 16-bit Count value #2. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |

| | |
|---|---|
| Prior Calls | registerBoardEx<br>TCsetMode |
| See Also | TCsetCount<br>TCgetCounts<br>TCsetClock<br>TCsetGate |

#### 6.4.5.14 Read Timer's current Count Value — TCgetCount

Latches and reads a timer/counter's 16-bit count value, using the counter latch command. (Prior to version 4.23 of the driver, the read-back command was used, but that never worked properly on PC24E, PC25E, PC26AT, PC27E or PC30AT.)

```
i = TCgetCount (h, chip, chan, pcount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pcount* | **pointer to long**: pointer to a long integer variable into which the count value result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetMode<br>TCsetCount | |
| See Also | TCgetUpCount<br>TCsetClock<br>TCsetGate | |

#### 6.4.5.15 Read Timer's current Up-Count — TCgetUpCount

Latches and reads a timer counter value, in the same way as TCgetCount, but returns the actual number of clock pulses received, rather than the count value. Note that the 82C54 timers count down to zero from the initial count value, so this function returns ((initial count) – (current count)). Only counter modes 2 or 3 should be used with this function.

**`i = TCgetUpCount (h, chip, chan, pcount)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pcount* | **pointer to long**: pointer to a long integer variable into which the up-count value result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetMode<br>TCsetCount | |
| See Also | TCgetCount<br>TCsetClock<br>TCsetGate | |

#### 6.4.5.16 Reads Two Timer's current Count Values — TCgetCounts

Latches and reads 16-bit count values from two timer/counter channels. It uses the read-back command if the channels are on the same 82C54 chip. It uses the counter latch command if the channels are on different chips or on an older 82C53 chip. (Prior to version 4.23 of the driver, the read-back command was always used, but that never worked properly on PC24E, PC25E, PC26AT, PC27E or PC30AT.)

SUPPORTED IN VERSION 2.00 ONWARDS

For driver version 5.04 onwards, a second set of readings is always taken. (Previous versions of the driver from version 3.00 onwards took a second set of readings if the counters were on different chips.) If the value of the second timer/counter changes between the two sets of readings, the second set of readings is used, otherwise the first set of readings is used. This is useful when the second timer/counter channel is clocked by the output of the first channel.

**`i = TCgetCounts (h, chip1, chan1, pcount1, chip2, chan2, pcount2)`**

| | | |
|---|---|---|
| where: | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip1* | **short**: address offset of the timer/counter chip |

| | | |
|---|---|---|
| | | #1. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan1* | **short**: timer/counter #1 channel number within the chip (0, 1 or 2). |
| | *pcount1* | **pointer to long**: pointer to a long integer variable into which the count value #1 result will be placed. |
| | *chip2* | **short**: address offset of the timer/counter chip #2. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan2* | **short**: timer/counter #2 channel number within the chip (0, 1 or 2). |
| | *pcount2* | **pointer to long**: pointer to a long integer variable into which the count value #2 result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetMode<br>TCsetCount<br>TCsetCounts | |
| See Also | TCgetCounts<br>TCsetClock<br>TCsetGate | |

#### 6.4.5.17 Gets a Timer's Initial Count Value — TCgetInitialCount

Gets a timer/counter channel's initial count value, if it has been set.

Another function such as TCsetCount or one of the higher-level timer/counter functions has to set the initial count after registering the board in order for this function to return the current initial count value. If the current initial count value is not known, the function will return ERRSUPPORT.

SUPPORTED IN VERSION 4.40 ONWARDS.

**`i = TCgetInitialCount (h, chip, chan, pcount)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *pcount* | **pointer to long**: pointer to a long integer variable into which the initial count value will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetCount | |

### 6.4.6 Differential Counter Functions

#### 6.4.6.1 Set-up Differential Counter Pair — TCsetDiffCounters

Sets up two counter/timers for a differential count operation. If the gate sources specified are both GAT_VCC, counting will start immediately. Otherwise the user must provide the gate signals or set the gates high by a call to TCsetGate. Note that the PC214E does not support software-configurable clock and gate settings, and the *clk1*, *clk2*, *gat1*, and *gat2* arguments will have no effect. See section 5.2.2 for details on the clock and gate sources available. See section 3.1.1 for more details on the Differential Counter application.

**`i = TCsetDiffCounters (h, chip1, chan1, clk1, gat1, chip2, chan2, clk2, gat2)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip1* | **short**: address offset of timer/counter chip #1. One of the following pre-defined constants may be used: |

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20

*chan1* **short**: timer/counter #1 channel number within the chip (0, 1 or 2).

*clk1* **short**: timer #1 clock source. One of the following pre-defined constants may be used:

CLK_CLK = 0: external CLK(*chan1*) i/p
CLK_10MHZ = 1: 10 MHz
CLK_1MHZ = 2: 1 MHz
CLK_100KHZ = 3: 100 kHz
CLK_10KHZ = 4: 10 kHz
CLK_1KHZ = 5: 1 kHz
CLK_OUTN_1 = 6: OUT(*chan1*–1)
CLK_EXT = 7: external EXTCLK(*chip1*) i/p

*gat1* **short**: timer #1 gate source. One of the following pre-defined constants may be used:

GAT_VCC = 0: Enabled
GAT_GND = 1: Disabled
GAT_EXT = 2: GAT(*chan1*) external i/p
GAT_OUTN_2 = 3: /OUT(*chan1*–2)

*chip2* **short**: address offset of timer/counter chip #2. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20.

*chan2* **short**: timer/counter #2 channel number within the chip (0, 1 or 2).

*clk2* **short**: timer #2 clock source. One of the following pre-defined constants may be used:

CLK_CLK = 0: external CLK(*chan2*) i/p
CLK_10MHZ = 1: 10 MHz
CLK_1MHZ = 2: 1 MHz
CLK_100KHZ = 3: 100 kHz
CLK_10KHZ = 4: 10 kHz
CLK_1KHZ = 5: 1 kHz
CLK_OUTN_1 = 6: OUT(*chan2*–1)
CLK_EXT = 7: external EXTCLK(chip2) i/p

*gat2* **short**: timer #2 gate source. One of the following pre-defined constants may be used:

| | | |
|---|---|---|
| | | GAT_VCC = 0: Enabled<br>GAT_GND = 1: Disabled<br>GAT_EXT = 2: GAT(*chan2*) external i/p<br>GAT_OUTN_2 = 3: /OUT(*chan2*–2) |
| Returns **short**: | Differential counter handle (>= 0). Use this handle as the *hD* parameter in calls to TCgetDiffCount, TCgetRatio and TCfreeDiffCounters when referring to this particular differential counter pair. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetGate<br>TCgetDiffCount<br>TCgetRatio<br>TCfreeDiffCounters | |

#### 6.4.6.2 Read Differential Count — TCgetDiffCount

Reads the difference between the count values of the two counters specified in the TCsetDiffCounters function.

```
i = TCgetDiffCount (h, hD, pdiff)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hD* | **short**: differential counter handle as issued by the TCsetDiffCounters function. |
| | *pdiff* | **pointer to long**: pointer to a long integer variable into which the 16-bit count value representing (Count#2 – Count#1) will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetDiffCounters | |
| See Also | TCgetRatio<br>TCfreeDiffCounters | |

#### 6.4.6.3 Read Differential Ratio — TCgetRatio

Reads the ratio of the count values of the two counter/timers specified in function TCsetDiffCounters.

**`i = TCgetRatio (h, hD, pratio)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hD* | **short**: differential counter handle as issued by the TCsetDiffCounters function. |
| | *pratio* | **pointer to float**: pointer to a 32-bit floating point variable into which the value representing the ratio of counts (Counter#2 / Counter#1) will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetDiffCounters | |
| See Also | TCgetDiffCount<br>TCfreeDiffCounters | |

#### 6.4.6.4 Free Differential Counter Pair — TCfreeDiffCounters

Frees the counter/timers associated with a differential pair, as set up by function TCsetDifferentialCounters. Call this function when finished with the differential counter.

**`i = TCfreeDiffCounters (h, hD)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hD* | **short**: differential counter handle as issued by the TCsetDiffCounters function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetDiffCounters | |
| See Also | TCgetDiffCount<br>TCgetRatio | |

### 6.4.7 Millisecond Stopwatch, Event Recorder and Event Counting Functions

#### 6.4.7.1 Prepare a Millisecond Stopwatch — TCsetStopwatch

Sets up a stopwatch, which uses two timer/counters to count in milliseconds for about 50 days. See section 3.1.4 for more details on the Stopwatch application.

**`i = TCsetStopwatch (h, chip, chan)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). This channel and the next channel (*chan*+1) are used. The second channel may be on the next timer/counter chip. |
| Returns **short**: | Stopwatch handle (>= 0). Use this in calls to the other stopwatch functions to refer to this stopwatch. | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetEventRecorder<br>TCstartStopwatch<br>TCfreeStopwatch | |

#### 6.4.7.2 Start a Millisecond Stopwatch — TCstartStopwatch

Starts a stopwatch that has been previously set up by the TCsetStopwatch function.

**`i = TCstartStopwatch (h, hS)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hS* | **short**: handle to stopwatch as issued by the TCsetStopwatch function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetStopwatch | |
| See Also | TCgetElapsedTime<br>TCfreeStopwatch | |

#### 6.4.7.3 Get Stopwatch Elapsed Time — TCgetElapsedTime

Gets the elapsed time, in milliseconds, since a given stopwatch was started.

```
i = TCgetElapsedTime (h, hS, ptime)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hS* | **short**: handle to stopwatch as issued by the TCsetStopwatch function. |
| | *ptime* | **pointer to long**: pointer to a long integer variable into which the elapsed time result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetStopwatch<br>TCstartStopwatch | |
| See Also | TCgetTimeStr<br>TCsetEventRecorder | |

#### 6.4.7.4 Prepare an Event Time Recorder — TCsetEventRecorder

Sets up an event recorder that records the times of positive edges on a PPI Port C bit 0 digital input (DI) line. The times recorded are the elapsed time since the given stopwatch was started). This is performed by using a stopwatch, previously set up by a call to TCsetStopwatch, and enabling the DI line to generate an interrupt. An interrupt service routine (ISR) stores the elapsed time from the stopwatch into a previously allocated data buffer for each event. See section 3.4.1 for more details on the Event Recorder application.

```
i = TCsetEventRecorder (h, hS, chip, hB)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hS* | **short**: stopwatch handle as issued by the TCsetStopwatch function. |
| | *chip* | **short**: address offset of the digital input chip from which Port C bit 0 will be used as the event input. Use one of the following pre-defined constants:-<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *hB* | **short**: buffer handle as issued by the allocateLongBuf function. |

| | |
|---|---|
| Returns **short**: | Event recorder handle (>= 0). Use this handle to call the TCfreeEventRecorder function when finished. |
| or | ERRHANDLE<br>ERRCHAN<br>ERRBUFFER |
| Prior Calls | registerBoardEx<br>TCsetStopwatch<br>allocateLongBuf |
| See Also | TCfreeEventRecorder<br>getLongIntItem<br>readLongBuf<br>copyFromLongBuf<br>enableInterrupts |

#### 6.4.7.5 Free-up Event Recorder Timer and Digital Input Channels — TCfreeEventRecorder

Frees up the event recorder.

```
i = TCfreeEventRecorder (h, hE)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hE* | **short**: event recorder handle as issued by the TCsetEventRecorder function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetStopwatch<br>allocateLongBuf<br>TCsetEventRecorder | |
| See Also | disableInterrupts<br>TCfreeStopwatch<br>freeLongBuf | |

#### 6.4.7.6 Convert Milliseconds into Time String — TCgetTimeStr

Converts a 32-bit word representing an elapsed time in milliseconds to a time string in the format “DD HH:MM:SS.TTT”. Such 32-bit elapsed times are produced by the stopwatch functions.

```
i = TCgetTimeStr (ms, strPtr)
```

| | | |
|---|---|---|
| where | *ms* | **long**: elapsed time in milliseconds. |
| | *strPtr* | **pointer to char**: pointer to buffer where null-terminated string for result is to be written. |
| Returns **short**: | OK | |

| | |
|---|---|
| or | ERRDATA |
| Prior Calls | none |
| See Also | TCgetElapsedTime |

#### 6.4.7.7 Free-up Stopwatch Counter/Timers — TCfreeStopwatch

Frees the timer/counters used by a stopwatch, as previously set up by TCsetStopwatch. Call this function when the stopwatch is no longer required.

```
i = TCfreeStopwatch (h, hS)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hS* | **short**: stopwatch handle as issued by the TCsetStopwatch function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetStopwatch | |
| See Also | | |

#### 6.4.7.8 Prepare a 32-Bit Event Counter — TCsetEventCounter

Sets up a 32-bit event counter, which uses 2 timer/counters to count events on the clock input of the specified timer/counter channel.

An 'event' is a rising edge followed by a falling edge on the clock input.

If an internal clock source is used to provide the events, this is similar to the stopwatch function.

SUPPORTED IN VERSION 4.42 ONWARDS.

See section 3.1.10 for more information on the 32-bit event counter application.

```
i = TCsetEventCounter (h, chip, chan, clock)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12 |

| | | |
|---|---|---|
| | | Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). This channel and the next channel (*chan*+1) are used. The second channel may be on the next timer/counter chip. |
| | *clock* | **short**: clock source (source of events) for the event counter. One of the following pre-defined constants representing the valid clock sources may be used (usually CLK_CLK or CLK_EXT):<br><br>CLK_CLK = 0: external CLK(*chan*) i/p<br>CLK_10MHZ = 1: internal 10 MHz<br>CLK_1MHZ = 2: internal 1 MHz<br>CLK_100KHZ = 3: internal 100 kHz<br>CLK_10KHZ = 4: internal 10 kHz<br>CLK_1KHZ = 5: internal 1 kHz<br>CLK_OUTN_1 = 6: OUT(*chan*–1)<br>CLK_EXT = 7: external EXTCLK(*chip*) i/p |
| Returns **short**: | Event counter handle (>= 0). Use this in calls to the other event counter functions to refer to this event counter. | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | TCresetEventCounter<br>TCgetEventCount<br>TCfreeEventCounter | |

#### 6.4.7.9 Reset a 32-bit Event Counter — TCresetEventCounter

Resets to zero a 32-bit event counter as previously set up by the *TCsetEventCounter* function.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCresetEventCounter (h, hE)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hE* | **short**: event counter handle as issued by the TCsetEventCounter function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetEventCounter | |
| See Also | TCgetEventCount<br>TCfreeEventCounter | |

#### 6.4.7.10 Read a 32-bit Event Counter — TCgetEventCount

Gets the number of events since a given 32-bit event counter, as previously set up by the *TCsetEventCounter* function, was set up or reset.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCgetEventCount (h, hE, pcount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hE* | **short**: event counter handle as issued by the TCsetEventCounter function. |
| | *pcount* | **pointer to unsigned long**: points to a variable into which the 32-bit event count is written. |
| Returns **short**: | 0 = counter not overflowed<br>1 = counter overflowed | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetEventCounter | |
| See Also | TCresetEventCounter<br>TCfreeEventCounter | |

#### 6.4.7.11 Free up 32-bit Event Counter — TCfreeEventCounter

Frees up a 32-bit event counter as previously set up by the *TCsetEventCounter* function. Call this function when the event counter is no longer required.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCfreeEventCounter (h, hE)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hE* | **short**: event counter handle as issued by the TCsetEventCounter function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetEventCounter | |
| See Also | TCgetEventCount | |

| TCresetEventCounter |
|---|

### 6.4.8 Frequency/Pulse Generation Functions

#### 6.4.8.1 Send Monostable Pulse — TCsetMonoShot

Creates a single pulse of specified duration on the output of a timer/counter, using the timer's 'Hardware Re-triggerable One-Shot' mode. In this mode, the timer output will go low for the duration specified on the clock pulse following a gate trigger. Subsequent gate triggers will re-trigger the pulse. See section 3.1.2 for more details on the Monostable application.

For cards without a Counter Connections Register block (e.g. PC214E) an input clock frequency of 1 MHz is assumed (4 MHz is assumed for PC27E).

For cards with a Counter Connections Register block, the highest internal clock frequency in 1 kHz, 10 kHz, 100 kHz, 1 MHz or 10 MHz that will support the specified duration is used.

```
i = TCsetMonoShot (h, chip, chan, duration)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *duration* | **float**: pulse duration time, in seconds.<br><br>The minimum duration is 1 s divided by the highest available input clock frequency. For cards with clock connection registers, the minimum duration is 100 ns.<br><br>The maximum duration is 65536 s divided by the lowest available input clock frequency. For cards with clock connection registers, the maximum duration is 65.536 s. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetGate | |

TCfreeResource

#### 6.4.8.2 Generate Astable Multivibrator Waveform — TCsetAstable

Generates a clock signal of specified frequency and mark-to-space ratio. This is implemented on two counters, both in mode 1 (digital one-shot). One counter counts the mark time and the other counts the space time. The outputs of each counter/timer control the gate of the other, so that when the mark times-out, the space counter is triggered and vice versa. N.B. the user must connect each counter's gate to the other's output on the user connector SK1. See section 3.1.3 for more details on the Astable application.

For cards without a Counter Connections Register block (e.g. PC214E) an input clock frequency of 1 MHz is assumed (4 MHz is assumed for PC27E).

For cards with a Counter Connections Register block, the highest internal clock frequency in 1 kHz, 10 kHz, 100 kHz, 1 MHz or 10 MHz that will support the required mark or space duration is used and chosen individually for each counter.

```
i = TCsetAstable (h, chip, chan, chipS, chanS, freq, msratio)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*chip* — **short**: address offset of timer/counter chip. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20

*chan* — **short**: timer/counter channel number within the chip (0, 1 or 2).

*chipS* — **short**: address offset of secondary timer/counter chip. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20

*chanS* — **short**: secondary timer/counter channel number within the chip (0, 1 or 2).

*freq* — **float**: desired frequency, in Hertz.

The frequency must be at least 0.005 Hz.

*msratio* — **float**: desired mark-to-space ratio, defined as (mark time/period), i.e. 0 is D.C. 0V, 1 is D.C. 5V, 0.5 is symmetrical square wave, i.e. high

| | | |
|---|---|---|
| | | for 1 and low for 1. |
| Returns **short**: | Handle to the astable multivibrator (>= 0). Use this handle to call the TCfreeAstable function when finished, in order to free up the counter/timers for re-use. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreeAstable | |

#### 6.4.8.3 Free-up Astable Multivibrator Counter/Timers — TCfreeAstable

Frees the two timer counters used for an astable multivibrator, as set-up by the TCsetAstable function.

```
i = TCfreeAstable (h, hA)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoard function. |
| | *hA* | **short**: astable multivibrator handle as issued by the TCsetAstable function |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetAstable | |
| See Also | | |

#### 6.4.8.4 Generate a Frequency — TCgenerateFreq

Generates a square wave of specified frequency on a single timer/counter. See section 3.1.6 for more details on the Frequency Generation application.

For cards without a Counter Connections Register block (e.g. PC214E) an input clock frequency of 1 MHz is assumed (4 MHz is assumed for PC27E).

For cards with a Counter Connections Register block, the highest internal clock frequency in 1 kHz, 10 kHz, 100 kHz, 1 MHz or 10 MHz that will support the desired output frequency is chosen.

```
i = TCgenerateFreq (h, chip, chan, freq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of timer/counter chip. One of the following pre-defined constants may |

| | | |
|---|---|---|
| | | be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |
| | *freq* | **float**: desired frequency in Hertz.<br><br>The frequency must be at least 0.01 Hz.<br><br>The maximum frequency is the highest available internal clock frequency divided by 2. For cards with clock connection registers, this is 5000000 Hz. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE | |
| Prior Calls | registerBoardEx | |
| See Also | TCgenerateAccFreq<br>TCgeneratePulse | |

#### 6.4.8.5 Generate an Accurate Frequency — TCgenerateAccFreq

Generates a square wave of specified frequency accurate to 0.1% using two cascaded timer/counters. See section 3.1.6 for more details on the Frequency Generation application.

For cards without a Counter Connections Register block (e.g. PC214E) an input clock frequency of 1 MHz is assumed (4 MHz is assumed for PC27E). This affects the accuracy.

For cards with a Counter Connections Register block an input clock frequency of 10 MHz is used.

The TCfreeResource function should be used to free up the timer/counter channels for use by other functions.

```
i = TCgenerateAccFreq (h, chip, chan, freq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of timer/counter chip #2. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8 |

| | | |
|---|---|---|
| | | Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). Another timer/counter (*chan* – 1) will be used by this function. This timer/counter may be on the previous chip. |
| | *freq* | **float**: desired frequency in Hertz.<br><br>The frequency must be at least 0.002328 Hz. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE | |
| Prior Calls | registerBoardEx | |
| See Also | TCgenerateFreq<br>TCfreeResource | |

#### 6.4.8.6 Generate a Pulse — TCgeneratePulse

Generates a negative-going pulse at a specified frequency on a single timer/counter. The pulse width is the period of the timer/counter clock input, which may be hard-wired or software-configured, depending on the card.

SUPPORTED IN VERSION 4.00 ONWARDS.

For cards without a Counter Connections Register block (e.g. PC214E) an input clock frequency of 1 MHz is assumed (4 MHz is assumed for PC27E).

For cards with a Counter Connections Register block, the highest internal clock frequency in 1 kHz, 10 kHz, 100 kHz, 1 MHz or 10 MHz that will support the desired output frequency is chosen.

```
i = TCgeneratePulse (h, chip, chan, freq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). |

| | | |
|---|---|---|
| | *freq* | **float**: desired frequency in Hertz.<br><br>The frequency must be at least 0.01 Hz.<br><br>The maximum frequency is the highest available internal clock frequency divided by 2. For cards with clock connection registers, this is 5000000 Hz.<br><br>For cards with clock connection registers, the internal input clock frequency is set to the highest of 1 kHz, 10 kHz, 100 kHz, 1 MHz or 10 Mhz that is no more than the desired frequency multiplied by 65536.<br><br>The pulse width is 1 s divided by the internal input clock frequency. |
| Returns *short*: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE | |
| Prior Calls | registerBoardEx | |
| See Also | TCgenerateFreq | |

#### 6.4.8.7 Set up a Periodic Pulse Train Generator — TCsetPeriodicPulseTrain

Sets up a periodic pulse train on the specified timer channel. It sets up a hardware-retriggerable one-shot on another timer channel to set the duration of the pulse train and sets up a periodic rate generator on a third timer channel (the 'train' channel) to retrigger the one-shot periodically.

SUPPORTED IN VERSION 4.32 ONWARDS.

The 'one-shot' channel is offset by -2 from the specified 'pulse' channel. The 'train' channel is offset by -2 from the 'one-shot' channel. All the timer channels on all timer chips are grouped together to determine a negative offset channel, so this may be on the next lower timer chip. If there is no next lower timer chip, a wraparound to the highest timer chip will be used to determine the offset channel.

If the board lacks clock connection and gate connection registers, the clock sources must be wired up manually. The output of the 'one-shot' channel must be inverted and wired up manually to the gate input of the 'pulse' channel. The output of the 'train' channel must be wired up manually to the gate input of the 'one-shot' channel.

For PC214E: Timer Z1 channel 0 provides an inverted output on connector SK1 so this is best used as the 'one-shot' channel, using Z1 channel 2 as the 'pulse' channel and Z1 channel 1 as the 'train' channel. Wire Ctr Z1 /OUT0 O/P (pin 54) to Ctr Z1 GAT2 I/P (pin 75), wire Ctr Z1 OUT1 O/P (pin 55) to Ctr Z1 GAT0 I/P (pin 73), set J2 to the 'one-shot' clock source, set J3 to the 'train' clock source and set J4 to the 'pulse' clock source. The pulse train output is on Ctr Z1 OUT2 O/P (pin 17).

```
i = TCsetPeriodicPulseTrain (h, pulseChip, pulseChan, pulseClock,
pulseCount, pulseShape, oneshotClock, oneshotTime, trainClock,
trainGate, trainFreq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pulseChip* | **short**: address offset of timer/counter chip for the pulse generator. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *pulseChan* | **short**: timer/counter channel number within the chip for the pulse generator (0, 1 or 2). Two other timer/counter channels (*pulseChan*-2 and *pulseChan*-4) will also be used. These timer/counter channels may be on previous chips or wrap around to the highest chips. |
| | *pulseClock* | **short**: clock source for the pulse generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *pulseCount* | **short**: number of pulses to output in each train. |
| | *pulseShape* | **short**: shape of pulses:<br><br>0 (negative-going for 1 clock period)<br>1 (negative-going square pulses) |
| | *oneshotClock* | **short**: clock source for the one-shot train duration. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *oneshotTime* | **double**: desired one-shot train duration in seconds. |
| | *trainClock* | **short**: clock source for the train rate generator. One of the following pre-defined constants may |

| | | |
|---|---|---|
| | | be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br><br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input)<br>GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral division of the train clock frequency. |
| Returns **short**: | Handle to the periodic pulse train generator (>= 0). Use this handle to call the TCfreePeriodicPulseTrain function when finished, in order to free up the counter/timers for re-use. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainCount<br>TCchangePeriodicPulseTrainDuration<br>TCcontrolPeriodicPulseTrain<br>TCsetRestrictedPulseTrain<br>TCsetOneShotPulseTrain | |

#### 6.4.8.8 Change Periodic Pulse Train's Gate Input — TCchangePeriodicPulseTrainGate

Sets the gate input for a periodic pulse train generator (as set up by TCsetPeriodicPulseTrain) to the specified value if the board has timer/counter gate connection registers. For example, if the periodic pulse train generator was set up with an initial gate value of GAT_GND to disable pulse generation, then a gate value of GAC_VCC may be used here to begin generating pulses at a later time.

SUPPORTED IN VERSION 4.32 ONWARDS.

Has no effect on boards without gate connection registers.

**`i = TCchangePeriodicPulseTrainGate (h, hPPT, trainGate)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input)<br>GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainCount<br>TCchangePeriodicPulseTrainDuration<br>TCcontrolPeriodicPulseTrain | |

#### 6.4.8.9 Change Periodic Pulse Train's Train Frequency — TCchangePeriodicPulseTrainFreq

Changes the train frequency for the periodic pulse train generator (as set up by TCsetPeriodicPulseTrain) to the specified value.

SUPPORTED IN VERSION 4.32 ONWARDS.

**`i = TCchangePeriodicPulseTrainFreq (h, hPPT, trainFreq)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral |

| | | |
|---|---|---|
| | | division of the train clock frequency. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainCount<br>TCchangePeriodicPulseTrainDuration<br>TCcontrolPeriodicPulseTrain | |

#### 6.4.8.10 Change Periodic Pulse Train's Pulse Count — TCchangePeriodicPulseTrainCount

Changes the pulse count for the periodic pulse train generator (as set up by TCsetPeriodicPulseTrain) to the specified value by altering the pulse frequency and the one-shot duration. For a one-shot currently in progress, the wrong number of pulses may be generated.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCchangePeriodicPulseTrainCount (h, hPPT, pulseCount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *pulseCount* | **short**: number of pulses to output in each train. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainDuration<br>TCcontrolPeriodicPulseTrain | |

#### 6.4.8.11 Change Periodic Pulse Train's Train Duration — TCchangePeriodicPulseTrainDuration

Changes the one-shot duration for the periodic pulse train generator (as set up by TCsetPeriodicPulseTrain) to the specified value by altering the pulse frequency and the one-shot duration. For a one-shot currently in progress, the wrong number of pulses may be

generated.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCchangePeriodicPulseTrainDuration (h, hPPT, oneshotTime)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *oneshotTime* | **double**: desired one-shot train duration in seconds. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainCount<br>TCcontrolPeriodicPulseTrain | |

#### 6.4.8.12 Control a Periodic Pulse Train Generator's Timer Channels — TCcontrolPeriodicPulseTrain

Stops or starts the 'pulse' channel and/or the 'one-shot' channel and/or the 'train' channel of a periodic pulse train generator. If the 'pulse' channel is stopped, its output will go high immediately. If the 'one-shot' channel is stopped, the outputs of both the 'one-shot' channel and the 'pulse' channel will go high immediately. If the 'train' channel is stopped, its output will go high immediately but there will be no immediate effect on the 'one-shot' and 'pulse' channels.

On setting up the periodic pulse train generator, all three channels are started. For pulses to be generated, all channels must be started and the 'train' channel's gate input must be high. If the 'train' channel is stopped or its gate input goes low during a pulse train, the current pulse train will complete unless the 'one-shot' channel and/or the 'pulse' channel is stopped.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCcontrolPeriodicPulseTrain (h, hPPT, runPulse, runOneshot,
runTrain)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *runPulse* | **short**: controls the 'pulse' channel: |

| | | |
|---|---|---|
| | | 0 (stop the pulse channel [if started])<br>1 (start the pulse channel [if stopped]) |
| | *runOneshot* | **short**: controls the 'one-shot' channel:<br><br>0 (stop the one-shot channel [if started])<br>1 (start the one-shot channel [if stopped]) |
| | *runTrain* | **short**: controls the 'train' channel:<br><br>0 (stop the train channel [if started])<br>1 (start the train channel [if stopped]) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCfreePeriodicPulseTrain<br>TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainCount<br>TCchangePeriodicPulseTrainDuration | |

#### 6.4.8.13 Free a Periodic Pulse Train Generator — TCfreePeriodicPulseTrain

Frees resources used by a periodic pulse train generator as set up by TCsetPeriodicPulseTrain.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCfreePeriodicPulseTrain (h, hPPT)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetPeriodicPulseTrain | |
| See Also | TCchangePeriodicPulseTrainGate<br>TCchangePeriodicPulseTrainFreq<br>TCchangePeriodicPulseTrainCount<br>TCchangePeriodicPulseTrainDuration<br>TCcontrolPeriodicPulseTrain | |

#### 6.4.8.14 Set up a Restricted Periodic Pulse Train Generator — TCsetRestrictedPulseTrain

Sets up a restricted periodic pulse train on the specified timer channel (the 'pulse' channel). Each train of pulses is output within a restricted period equal to the clock period of a second timer channel (the 'train' channel).

SUPPORTED IN VERSION 4.32 ONWARDS.

The 'train' channel is offset by -2 from the specified 'pulse' channel. If the 'pulse' channel is timer channel 0 or 1 on a timer chip, the 'train' channel will be channel 1 or 2 on the next lower timer chip or on the highest timer chip (due to wrap-around).

If the board lacks clock connection and gate connection registers, the clock sources must be wired up manually and the output of the 'train' channel must be inverted and wired up manually to the gate input of the 'pulse' channel.

For PC214E: Timer Z1 channel 0 provides an inverted output on connector SK1 so this is best used as the 'train' channel, using Z1 channel 2 as the 'pulse' channel. Wire Ctr Z1 /OUT0 O/P (pin 54) to Ctr Z1 GAT2 I/P (pin 75), set J2 to the 'train' clock source and set J4 to the 'pulse' clock source. The pulse train output is on Ctr Z1 OUT2 O/P (pin 17).

```
i = TCsetRestrictedPulseTrain (h, pulseChip, pulseChan, pulseClock,
pulseCount, pulseShape, trainClock, trainGate, trainFreq)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*pulseChip* — **short**: address offset of timer/counter chip for the pulse generator. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20

*pulseChan* — **short**: timer/counter channel number within the chip for the pulse generator (0, 1 or 2). Another timer/counter channel (*pulseChan*-2) will also be used, and may be on the previous chip or wrap around to the highest chip.

*pulseClock* — **short**: clock source for the pulse generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:

-1 (choose automatically)
CLK_10MHZ = 1 (fixed 10 MHz clock)
CLK_1MHZ = 2 (fixed 1 MHz clock)
CLK_100KHZ = 3 (fixed 100 kHz clock)
CLK_10KHZ = 4 (fixed 10 kHz clock)
CLK_1KHZ = 5 (fixed 1 kHz clock)

*pulseCount* — **short**: number of pulses to output in each train.

| | | |
|---|---|---|
| | *pulseShape* | **short**: shape of pulses:<br><br>0 (negative-going for 1 clock period)<br>1 (negative-going square pulses) |
| | *trainClock* | **short**: clock source for the train rate generator. One of the following pre-defined constants may be used:<br><br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br><br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input)<br>GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral division of the train clock frequency. |
| Returns **short**: | Handle to the restricted periodic pulse train generator (>= 0) ). Use this handle to call the TCfreeRestrictedPulseTrain function when finished, in order to free up the counter/timers for re-use. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreeRestrictedPulseTrain<br>TCchangeRestrictedPulseTrainGate<br>TCchangeRestrictedPulseTrainFreq<br>TCchangeRestrictedPulseTrainCount<br>TCcontrolRestrictedPulseTrain<br>TCsetPeriodicPulseTrain | |

#### 6.4.8.15 Change Restricted Periodic Pulse Train's Gate Input — TCchangeRestrictedPulseTrainGate

Sets the gate input for a restricted periodic pulse train generator (as set up by TCsetRestrictedPulseTrain) to the specified value if the board has timer/counter gate connection registers. For example, if the pulse train generator was set up with an initial gate value of GAT_GND to disable pulse generation, then a gate value of GAC_VCC may be used here to begin generating pulses at a later time.

SUPPORTED IN VERSION 4.32 ONWARDS.

Has no effect on boards without gate connection registers.

**`i = TCchangeRestrictedPulseTrainGate (h, hRPT, trainGate)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hRPT* | **short**: restricted periodic pulse train generator handle, as issued by the TCsetRestrictedPulseTrain function. |
| | *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input)<br>GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetRestrictedPulseTrain | |
| See Also | TCfreeRestrictedPulseTrain<br>TCchangeRestrictedPulseTrainFreq<br>TCchangeRestrictedPulseTrainCount<br>TCcontrolRestrictedPulseTrain | |

#### 6.4.8.16 Change Restricted Periodic Pulse Train's Frequency — TCchangeRestrictedPulseTrainFreq

Changes the frequency of trains for the restricted periodic pulse train generator (as set up by TCsetRestrictedPulseTrain) to the specified value.

SUPPORTED IN VERSION 4.32 ONWARDS.

**`i = TCchangeRestrictedPulseTrainFreq (h, hRPT, trainFreq)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hRPT* | **short**: restricted periodic pulse train generator handle, as issued by the TCsetRestrictedPulseTrain function. |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral |

| | | |
|---|---|---|
| | | division of the train clock frequency. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetRestrictedPulseTrain | |
| See Also | TCfreeRestrictedPulseTrain<br>TCchangeRestrictedPulseTrainGate<br>TCchangeRestrictedPulseTrainCount<br>TCcontrolRestrictedPulseTrain | |

#### 6.4.8.17 Change Restricted Periodic Pulse Train's Pulse Count — TCchangeRestrictedPulseTrainCount

Changes the pulse count for the restricted periodic pulse train generator (as set up by TCsetRestrictedPulseTrain) to the specified value by altering the pulse frequency. For a pulse train currently in progress, the wrong number of pulses may be generated.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCchangeRestrictedPulseTrainCount (h, hRPT, pulseCount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hRPT* | **short**: restricted periodic pulse train generator handle, as issued by the TCsetRestrictedPulseTrain function. |
| | *pulseCount* | **short**: number of pulses to output in each train. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetRestrictedPulseTrain | |
| See Also | TCfreeRestrictedPulseTrain<br>TCchangeRestrictedPulseTrainGate<br>TCchangeRestrictedPulseTrainFreq<br>TCcontrolRestrictedPulseTrain | |

#### 6.4.8.18 Control a Restricted Periodic Pulse Train Generator's Timer Channels — TCcontrolRestrictedPulseTrain

Stops or starts the 'pulse' channel and/or the 'train' channel of the restricted periodic pulse train generator. If the 'pulse' channel is stopped, its output will go high immediately. If the 'train' channel is stopped, the outputs of both the 'train' channel and the 'pulse' channel will go high immediately.

On setting up the restricted periodic pulse train generator, both channels are started. For pulses to be generated, both channels must be started and the 'train' channel's gate input must be high.

SUPPORTED IN VERSION 4.40 ONWARDS.

**`i = TCcontrolRestrictedPulseTrain (h, hRPT, runPulse, runTrain)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPPT* | **short**: periodic pulse train generator handle, as issued by the TCsetPeriodicPulseTrain function. |
| | *runPulse* | **short**: controls the 'pulse' channel:<br>0 (stop the pulse channel [if started])<br>1 (start the pulse channel [if stopped]) |
| | *runTrain* | **short**: controls the 'train' channel:<br>0 (stop the train channel [if started])<br>1 (start the train channel [if stopped]) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetRestrictedPulseTrain | |
| See Also | TCfreeRestrictedPulseTrain<br>TCchangeRestrictedPulseTrainGate<br>TCchangeRestrictedPulseTrainFreq<br>TCchangeRestrictedPulseTrainCount | |

#### 6.4.8.19 Free a Restricted Periodic Pulse Train Generator — TCfreeRestrictedPulseTrain

Frees resources used by a restricted periodic pulse train generator as set up by TCsetRestrictedPulseTrain.

SUPPORTED IN VERSION 4.32 ONWARDS.

**`i = TCfreeRestrictedPulseTrain (h, hRPT)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hRPT* | **short**: restricted periodic pulse train generator handle, as issued by the TCsetRestrictedPulseTrain function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |

| | |
|---|---|
| Prior Calls | registerBoardEx<br>TCsetRestrictedPulseTrain |
| See Also | TCchangeRestrictedPulseTrainGate<br>TCchangeRestrictedPulseTrainFreq<br>TCchangeRestrictedPulseTrainCount<br>TCcontrolRestrictedPulseTrain |

#### 6.4.8.20 Set up a Hardware-Triggered One-Shot Pulse Train Generator — TCsetOneShotPulseTrain

Sets up a pulse train within a hardware-retriggerable one-shot period. The pulses are generated on the specified 'pulse' chip and channel. The hardware-retriggerable one-shot period is generated by the 'one-shot' channel. The 'one-shot' channel has a trigger input (on the timer channel's gate input). A low-to-high transition on the trigger input will trigger (or retrigger) the one-shot period.

SUPPORTED IN VERSION 4.32 ONWARDS.

The 'one-shot' channel is offset by -2 from the specified 'pulse' channel. If the 'pulse' channel is timer channel 0 or 1 on a timer chip, the 'one-shot' channel will be channel 1 or 2 on the next lower timer chip or on the highest timer chip.

If the board lacks clock connection and gate connection registers, the clock sources must be wired up manually and the output of the 'one-shot' channel must be inverted and wired up manually to the gate input of the 'pulse' channel. Also, the one-shot trigger input must be wired up manually.

If a software trigger is desired for the one-shot, this may be accomplished by controlling the trigger input directly if the board has gate connection registers, or by connecting the trigger input to a digital output that is settable by software.

For PC214E: Timer Z1 channel 0 provides an inverted output on connector SK1 so this is best used as the 'one-shot' channel, using Z1 channel 2 as the 'pulse' channel. Wire Ctr Z1 /OUT0 O/P (pin 54) to Ctr Z1 GAT2 I/P (pin 75), set J2 to the 'one-shot' clock source and set J4 to the 'pulse' clock source. The pulse train output is on Ctr Z1 OUT2 O/P (pin 17) and the trigger input is on Ctr Z1 GAT0 I/P (pin 73).

```
i = TCsetOneShotPulseTrain (h, pulseChip, pulseChan, pulseClock,
pulseCount, pulseShape, oneshotClock, oneshotTrigger, oneshotTime)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pulseChip* | **short**: address offset of timer/counter chip for the pulse generator. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *pulseChan* | **short**: timer/counter channel number within the chip for the pulse generator (0, 1 or 2). Another timer/counter channel (*pulseChan*-2) will also |

| | | |
|---|---|---|
| | | be used, and may be on the previous chip or wrap around to the highest chip. |
| | *pulseClock* | **short**: clock source for the pulse generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *pulseCount* | **short**: number of pulses to output in each train. |
| | *pulseShape* | **short**: shape of pulses:<br><br>0 (negative-going for 1 clock period)<br>1 (negative-going square pulses) |
| | *oneshotClock* | **short**: clock source for the one-shot train duration. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock) |
| | *oneshotTrigger* | **short**: trigger input setting for the hardware-retriggerable one-shot. One of the following pre-defined constants may be used:<br><br>GAT_VCC = 0 (trigger input high)<br>GAT_GND = 1 (trigger input low)<br>GAT_EXT = 2 (trigger from external gate input)<br>GAT_OUTN_2 = 3 (trigger from inverted output of a timer channel offset by -2 from the one-shot channel) |
| | *oneshotTime* | **double**: desired one-shot train duration in seconds. |
| Returns **short**: | Handle to the one-shot pulse train generator (>= 0). Use this handle to call the TCfreeOneShotPulseTrain function when finished, in order to free up the counter/timers for re-use. | |
| or | ERRHANDLE<br>ERRCHAN | |

| | |
|---|---|
| | ERRDATA |
| Prior Calls | registerBoardEx |
| See Also | TCfreeOneShotPulseTrain<br>TCchangeOneShotPulseTrainTrigger<br>TCchangeOneShotPulseTrainCount<br>TCchangeOneShotPulseTrainDuration<br>TCcontrolOneShotPulseTrain<br>TCsetPeriodicPulseTrain |

#### 6.4.8.21 Change One-Shot Pulse Train's Trigger Input — TCchangeOneShotPulseTrainTrigger

Sets the trigger input for a hardware-retriggerable one-shot pulse train generator (as set up by TCsetOneShotPulseTrain) to the specified value if the board has counter timer gate connection registers.

SUPPORTED IN VERSION 4.32 ONWARDS.

Has no effect on boards without gate connection registers.

```
i = TCchangeOneShotPulseTrainTrigger (h, hOSPT, oneshotTrigger)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hOSPT* | **short**: one-shot pulse train generator handle, as issued by the TCsetOneShotPulseTrain function. |
| | *oneshotTrigger* | **short**: trigger input setting for the hardware-retriggerable one-shot. One of the following pre-defined constants may be used:<br>GAT_VCC = 0 (trigger input high)<br>GAT_GND = 1 (trigger input low)<br>GAT_EXT = 2 (trigger from external gate input)<br>GAT_OUTN_2 = 3 (trigger from inverted output of a timer channel offset by -2 from the one-shot channel) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetOneShotPulseTrain | |
| See Also | TCfreeOneShotPulseTrain<br>TCchangeOneShotPulseTrainCount<br>TCchangeOneShotPulseTrainDuration<br>TCcontrolOneShotPulseTrain | |

#### 6.4.8.22 Change One-Shot Pulse Train's Pulse Count — TCchangeOneShotPulseTrainCount

Changes the pulse count for the hardware-retriggerable one-shot pulse train generator (as set up by TCsetOneShotPulseTrain) to the specified value by altering the pulse frequency and the one-shot duration. For a one-shot currently in progress, the wrong number of pulses may be generated.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCchangeOneShotPulseTrainCount (h, hOSPT, pulseCount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hOSPT* | **short**: one-shot pulse train generator handle, as issued by the TCsetOneShotPulseTrain function. |
| | *pulseCount* | **short**: number of pulses to output in each train. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetOneShotPulseTrain | |
| See Also | TCfreeOneShotPulseTrain<br>TCchangeOneShotPulseTrainTrigger<br>TCchangeOneShotPulseTrainDuration<br>TCcontrolOneShotPulseTrain | |

#### 6.4.8.23 Change One-Shot Pulse Train's Train Duration — TCchangeOneShotPulseTrainDuration

Changes the one-shot duration for the hardware-retriggerable one-shot pulse train generator (as set up by TCsetOneShotPulseTrain) to the specified value by altering the pulse frequency and the one-shot duration. For a one-shot currently in progress, the wrong number of pulses may be generated.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCchangeOneShotPulseTrainDuration (h, hOSPT, oneshotTime)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hOSPT* | **short**: one-shot pulse train generator handle, as issued by the TCsetOneShotPulseTrain function. |
| | *oneshotTime* | **double**: desired one-shot train duration in seconds. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |

| | |
|---|---|
| | ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx<br>TCsetOneShotPulseTrain |
| See Also | TCfreeOneShotPulseTrain<br>TCchangeOneShotPulseTrainTrigger<br>TCchangeOneShotPulseTrainCount<br>TCcontrolOneShotPulseTrain |

#### 6.4.8.24 Control a Hardware-Triggered One-Shot Pulse Train Generator's Timer Channels — TCcontrolOneShotPulseTrain

Stops or starts the 'pulse' channel and/or the 'one-shot' channel of the hardware retriggerable one-shot pulse train generator. If the 'pulse' channel is stopped, its output will go high immediately. If the 'one-shot' channel is stopped, the outputs of both the 'one-shot' channel and the 'pulse' channel will go high immediately.

On setting up the hardware retriggerable one-shot pulse train generator, both channels are started. For the pulse train generator to be armed, both channels must be started.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = TCcontrolOneShotPulseTrain (h, hOSPT, runPulse, runOneshot)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hOSPT* | **short**: one-shot pulse train generator handle, as issued by the TCsetOneShotPulseTrain function. |
| | *runPulse* | **short**: controls the 'pulse' channel:<br>0 (stop the pulse channel [if started])<br>1 (start the pulse channel [if stopped]) |
| | *runOneshot* | **short**: controls the 'one-shot' channel:<br>0 (stop the one-shot channel [if started])<br>1 (start the one-shot channel [if stopped]) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetOneShotPulseTrain | |
| See Also | TCfreeOneShotPulseTrain<br>TCchangeOneShotPulseTrainTrigger<br>TCchangeOneShotPulseTrainCount<br>TCchangeOneShotPulseTrainDuration | |

#### 6.4.8.25 Free a Hardware-Triggered One-Shot Pulse Train Generator — TCfreeOneShotPulseTrain

Frees resources used by a hardware-retriggerable one-shot pulse train generator as set up by TCsetOneShotPulseTrain.

SUPPORTED IN VERSION 4.32 ONWARDS.

```
i = TCfreeOneShotPulseTrain (h, hOSPT)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hOSPT* | **short**: one-shot pulse train generator handle, as issued by the TCsetOneShotPulseTrain function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetOneShotPulseTrain | |
| See Also | TCchangeOneShotPulseTrainTrigger<br>TCchangeOneShotPulseTrainCount<br>TCchangeOneShotPulseTrainDuration<br>TCcontrolOneShotPulseTrain | |

#### 6.4.8.26 Set up a Programmable Width Pulse Generator — TCsetPWPulse

Sets up a programmable width pulse generator. Generates a hardware-retriggerable low-going pulse (mono-shot), where the 'low' period is a specified proportion of a specified period. This is similar to TCsetMonoShot. The timer output will go low for dutyCycle * period seconds following a rising edge trigger on the gate input. The timer uses an internal clock source either specified explicitly or chosen automatically. The automatically chosen clock source depends on the specified period.

N.B. Use TCsetGate to change the trigger source.

The programmable width pulse generator is also used internally by the pulse-width modulated pulse train generator (see TCsetPWMTrain).

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCsetPWPulse (h, chip, chan, clock, duty, period)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of timer/counter chip for the pulse generator. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12 |

| | | |
|---|---|---|
| | | Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip for the pulse generator (0, 1 or 2). |
| | *clock* | **short**: clock source for the pulse generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock)<br><br>For PC214E, a clock source of –1 is treated as CLK_1MHZ irrespective of the actual jumper settings. |
| | *duty* | **double**:.proportion of the specified period to spend with the output low, range 0 to 1. |
| | *period* | **double**: assumed period between hardware triggers, used to calculate the output low time. This is used to choose the internal clock source when the clock source is chosen automatically. This is just an assumed period. The function has no control over the actual period between triggers.<br><br>The maximum period is 65536s divided by the frequency of the clock source, e.g. for CLK_1KHZ the maximum period is 65.356s. If the clock source is chosen automatically and the card has clock connection registers, the maximum period is as for CLK_1KHZ, i.e. 65.536s. |
| Returns **short**: | Handle to the programmable width pulse generator (>= 0). Use this handle to call the TCfreePWPulse function when finished, in order to free up the counter/timer for re-use. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreePWPulse<br>TCchangePWPulseDutyCycle<br>TCchangePWPulsePeriod<br>TCcontrolPWPulse<br>TCsetMonoShot<br>TCsetPWMTrain | |

#### 6.4.8.27 Change Programmable Width Pulse Generator's Duty Cycle — TCchangePWPulseDutyCycle

Changes the output low period of the programmable width pulse generator by specifying a new duty cycle. This is the proportion of the previously specified period to spend with the output low when the pulse generator is triggered. This function has no direct control over the output high period.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCchangePWPulseDutyCycle (h, hPWP, duty)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWP* | **short**: programmable width pulse generator handle, as issued by the TCsetPWPulse function. |
| | *duty* | **double**:.proportion of the specified period to spend with the output low, range 0 to 1. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPWPulse | |
| See Also | TCfreePWPulse<br>TCchangePWPulsePeriod<br>TCcontrolPWPulse | |

#### 6.4.8.28 Change Programmable Width Pulse Generator's Period — TCchangePWPulsePeriod

Changes the assumed period of the programmable width pulse generator, which affects the length of the output low period when the pulse generator is triggered. There is no direct control over the output high period.

Note that if the programmable width pulse generator was set up to use an automatically chosen internal clock source, then a new internal clock source may be chosen by this function. This will result in the programmable width pulse generator generating output pulses of incorrect length while this function is running.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCchangePWPulsePeriod (h, hPWP, period)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWP* | **short**: programmable width pulse generator handle, as issued by the TCsetPWPulse function. |
| | *period* | **double**: assumed period between hardware triggers, used to calculate the output low time. |

| | | |
|---|---|---|
| | | This is used to choose the internal clock source when the clock source is chosen automatically. This is just an assumed period. The function has no control over the actual period between triggers.<br><br>The maximum period is 65536 s divided by the frequency of the clock source, e.g. for CLK_1KHZ the maximum period is 65.356 s. If the clock source is chosen automatically and the card has clock connection registers, the maximum period is as for CLK_1KHZ, i.e. 65.536 s. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPWPulse | |
| See Also | TCfreePWPulse<br>TCchangePWPulseDutyCycle<br>TCcontrolPWPulse | |

#### 6.4.8.29 Control a Programmable Width Pulse Generator's Timer Channel — TCcontrolPWPulse

Stops or starts the programmable width pulse generator. When the pulse generator is stopped, its output will go high immediately and it will no longer respond to triggers on its gate input. When the pulse generator is started, it will respond to triggers on its gate input.

When initially set up by TCsetPWPulse, the pulse generator is started.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCcontrolPWPulse (h, hPWP, run)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWP* | **short**: programmable width pulse generator handle, as issued by the TCsetPWPulse function. |
| | *run* | **short**: controls the programmable width pulse generator:<br><br>0 (stop the pulse channel [if started])<br>1 (start the pulse channel [if stopped]) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |

| | |
|---|---|
| | TCsetPWPulse |
| See Also | TCfreePWPulse<br>TCchangePWPulseDutyCycle<br>TCchangePWPulsePeriod |

#### 6.4.8.30 Free a Programmable Width Pulse Generator — TCfreePWPulse

Frees the timer counter resources used by a programmable width pulse generator, as previously set up by TCsetPWPulse.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCfreePWPulse (h, hPWP)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWP* | **short**: programmable width pulse generator handle, as issued by the TCsetPWPulse function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetPWPulse | |
| See Also | TCchangePWPulseDutyCycle<br>TCchangePWPulsePeriod<br>TCcontrolPWPulse | |

#### 6.4.8.31 Set up a Pulse Width Modulated Pulse Train Generator — TCsetPWMTrain

Sets up a continuous pulse-width modulated pulse train on the specified timer channel (the 'pulse' channel) using a hardware-retriggerable one-shot. It sets up a periodic rate generator on a second timer channel (the 'train' channel) to retrigger the one-shot periodically.

SUPPORTED IN VERSION 4.42 ONWARDS.

The 'train' channel is offset by –2 from the 'pulse' channel. If the 'pulse' channel is timer channel 0 or 1 on a timer chip, the 'train' channel will be channel 1 or 2 on the next lower timer chip or on the highest timer chip.

If the board lacks clock connection and gate connection registers, the clock sources must be wired up manually. The output of the 'train' channel must be wired up manually to the gate input of the 'pulse' channel.

```
i = TCsetPWMTrain (h, pulseChip, pulseChan, pulseClock, trainClock,
trainGate, duty, trainFreq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pulseChip* | **short**: address offset of timer/counter chip for |

| | |
|---|---|
| | the pulse generator. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| *pulseChan* | **short**: timer/counter channel number within the chip for the pulse generator (0, 1 or 2). Another timer/counter channel (*pulseChan*-2) will also be used, and may be on the previous chip or wrap around to the highest chip. |
| *pulseClock* | **short**: clock source for the pulse generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock)<br><br>For PC214E, a clock source of –1 is treated as CLK_1MHZ irrespective of the actual jumper settings. |
| *trainClock* | **short**: clock source for the train rate generator. One of the following pre-defined constants may be used, or use -1 to choose a value automatically for boards with clock connection registers:<br><br>-1 (choose automatically)<br>CLK_10MHZ = 1 (fixed 10 MHz clock)<br>CLK_1MHZ = 2 (fixed 1 MHz clock)<br>CLK_100KHZ = 3 (fixed 100 kHz clock)<br>CLK_10KHZ = 4 (fixed 10 kHz clock)<br>CLK_1KHZ = 5 (fixed 1 kHz clock)<br><br>For PC214E, a clock source of –1 is treated as CLK_1MHZ irrespective of the actual jumper settings. |
| *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br><br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input) |

| | | |
|---|---|---|
| | | GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| | *duty* | **double**:.proportion of the specified period to spend with the output low, range 0 to 1. |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral division of the train clock frequency.<br><br>The minimum train frequency is the frequency of the train clock source divided by 65536, or the frequency of the pulse clock source divided by 65536, whichever is the greatest. E.g. for CLK_1KHZ the minimum train frequency is about 0.01526 Hz. If both the train clock source and pulse clock source are chosen automatically, the minimum train frequency is as for CLK_1KHZ, i.e. about 0.01526 Hz.<br><br>The maximum train frequency is the frequency of the train clock source divided by 2, e.g. for CLK_10MHZ the maximum train frequency is 5000000 Hz. If the train clock source is chosen automatically, the maximum train frequency is as for CLK_10MHZ, i.e. 5000000 Hz. |
| Returns **short**: | Handle to the pulse width modulated pulse train generator (>= 0). Use this handle to call the TCfreePWMTrain function when finished, in order to free up the counter/timers for re-use. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCfreePWMTrain<br>TCchangePWMTrainGate<br>TCchangePWMTrainFreq<br>TCchangePWMTrainDutyCycle<br>TCcontrolPWMTrain | |

#### 6.4.8.32 Change Pulse Width Modulated Pulse Train Generator's Gate — TCchangePWMTrainGate

Sets the gate for the pulse width modulated pulse train generator to the specified value. For example, if the PWM pulse train generator was set up with an initial gate value of GAT_GND to disable pulse generation, then a gate value of GAC_VCC may be used here to begin generating pulses at a later time.

SUPPORTED IN VERSION 4.42 ONWARDS.

Has no effect on boards without gate connection registers.

```
i = TCchangePWMTrainGate (h, hPWMT, trainGate)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWMT* | **short**: pulse width modulated pulse train generator handle, as issued by the TCsetPWMTrain function. |
| | *trainGate* | **short**: gate input source for the train rate generator (if the board has gate connection registers). One of the following pre-defined constants may be used:<br>GAT_VCC = 0 (gate on/high)<br>GAT_GND = 1 (gate off/low)<br>GAT_EXT = 2 (gate from external input)<br>GAT_OUTN_2 = 3 (gate from inverted output of a timer channel offset by -2 from train channel) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPWMTrain | |
| See Also | TCfreePWMTrain<br>TCchangePWMTrainFreq<br>TCchangePWMTrainDutyCycle<br>TCcontrolPWMTrain | |

#### 6.4.8.33 Change Pulse Width Modulated Pulse Train Generator's Frequency — TCchangePWMTrainFreq

Changes the frequency and duration of the pulses for the pulse width modulated pulse train generator, preserving the duty cycle.

SUPPORTED IN VERSION 4.42 ONWARDS.

If the 'pulse' channel uses an automatically chosen internal clock, there may be a glitch when the frequency is changed.

```
i = TCchangePWMTrainFreq (h, hPWMT, trainFreq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWMT* | **short**: pulse width modulated pulse train generator handle, as issued by the TCsetPWMTrain function. |
| | *trainFreq* | **double**: desired train frequency in Hz. The actual frequency will be rounded to an integral division of the train clock frequency. |

| | | |
|---|---|---|
| | | The minimum train frequency is the frequency of the train clock source divided by 65536, or the frequency of the pulse clock source divided by 65536, whichever is the greatest. E.g. for CLK_1KHZ the minimum train frequency is about 0.01526 Hz. If both the train clock source and pulse clock source are chosen automatically, the minimum train frequency is as for CLK_1KHZ, i.e. about 0.01526 Hz.<br><br>The maximum train frequency is the frequency of the train clock source divided by 2, e.g. for CLK_10MHZ the maximum train frequency is 5000000 Hz. If the train clock source is chosen automatically, the maximum train frequency is as for CLK_10MHZ, i.e. 5000000 Hz. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetPWMTrain | |
| See Also | TCfreePWMTrain<br>TCchangePWMTrainGate<br>TCchangePWMTrainDutyCycle<br>TCcontrolPWMTrain | |

#### 6.4.8.34 Change Pulse Width Modulated Pulse Train Generator's Duty Cycle — TCchangePWMTrainDutyCycle

Changes the duration of the pulses for the pulse width modulated pulse train generator, preserving the frequency.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCchangePWMTrainDutyCycle (h, hPWMT, duty)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWMT* | **short**: pulse width modulated pulse train generator handle, as issued by the TCsetPWMTrain function. |
| | *duty* | **double**:.proportion of the specified period to spend with the output low, range 0 to 1. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |

| | |
|---|---|
| Prior Calls | registerBoardEx<br>TCsetPWMTrain |
| See Also | TCfreePWMTrain<br>TCchangePWMTrainGate<br>TCchangePWMTrainFreq<br>TCcontrolPWMTrain |

#### 6.4.8.35 Control a Pulse Width Modulated Pulse Train Generator's Timer Channels — TCcontrolPWMTrain

Stops or starts the 'pulse' channel and/or the 'train' channel of the pulse-width modulated pulse train generator. If the 'pulse' channel is stopped, its output will go high immediately. If the 'train' channel is stopped, its output will go high immediately; if its output was low,this can trigger a pulse on the 'pulse' channel.

On setting up the pulse width modulated pulse train generator, both channels are started. For pulses to be generated, both channels must be started and the 'train' channel's gate input must be high. If the 'train' channel is stopped or its gate input goes low, the current pulse will complete unless the 'pulse' channel is stopped. Stopping the 'train' channel while its output is low can trigger a pulse on the pulse channel unless the pulse channel is stopped.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCcontrolPWMTrain (h, hPWMT, runPulse, runTrain)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWMT* | **short**: pulse width modulated pulse train generator handle, as issued by the TCsetPWMTrain function. |
| | *runPulse* | **short**: controls the 'pulse' channel:<br>0 (stop the pulse channel [if started])<br>1 (start the pulse channel [if stopped]) |
| | *runTrain* | **short**: controls the 'train' channel:<br>0 (stop the train channel [if started])<br>1 (start the train channel [if stopped]) |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetPWMTrain | |
| See Also | TCfreePWMTrain<br>TCchangePWMTrainGate<br>TCchangePWMTrainFreq<br>TCchangePWMTrainDutyCycle | |

#### 6.4.8.36 Free a Pulse Width Modulated Pulse Train Generator — TCfreePWMTrain

Frees the timer counter resources used by a pulse width modulated pulse train generator, as previously setup by TCsetPWMTrain.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = TCfreePWMTrain (h, hPWMT)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hPWMT* | **short**: pulse width modulated pulse train generator handle, as issued by the TCsetPWMTrain function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetPWMTrain | |
| See Also | TCchangePWMTrainGate<br>TCchangePWMTrainFreq<br>TCchangePWMTrainDutyCycle<br>TCcontrolPWMTrain | |

### 6.4.9 Frequency Input and Regeneration Functions

#### 6.4.9.1 Measure Period of an External Signal — TCgetExtPeriod

Returns the period of an external signal, measured in microseconds. The external signal must be connected to the clock input of the timer channel specified by the *chip* and *chan* arguments. See section 3.1.5 for more details on the Frequency/Period Measurement application.

```
i = TCgetExtPeriod (h, chip, chan, pper)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). Another timer/counter (*chan*–2) will also be used to provide the gate signal. |

| | | |
|---|---|---|
| | | This second timer/counter may be on the previous chip. |
| | *pper* | **pointer to f**loat: Pointer to a 32-bit floating-point variable into which the period result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCgetExtFreq | |

#### 6.4.9.2 Measure Frequency of an External Signal — TCgetExtFreq

Returns the frequency of an external signal, in Hertz. The external signal must be connected to the clock input of the timer specified by the *chip* and *chan* arguments. See section 3.1.5 for more details on the Frequency/Period Measurement application.

```
i = TCgetExtFreq (h, chip, chan, pfreq)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). Another timer/counter (*chan*–2) will also be used to provide the gate signal. This second timer/counter may be on the previous chip. |
| | *pfreq* | **pointer to f**loat: Pointer to a 32-bit floating-point variable into which the frequency result will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE<br>ERRDATA | |
| Prior Calls | registerBoardEx | |

| | |
|---|---|
| See Also | TCgetExtPeriod<br>TCgetExtFreqRestricted |

#### 6.4.9.3 Measure Frequency of an External Signal Over a Fixed Period — TCgetExtFreqRestricted

Returns the frequency of an external signal, in Hertz, as measured over a specified period. The external signal must be connected to the clock input of the timer specified by the *chip* and *chan* arguments. See section 3.1.5 for more details on the Frequency/Period Measurement application.

SUPPORTED IN VERSION 4.42 ONWARDS

```
i = TCgetExtFreqRestricted (h, chip, chan, width, pfreq, poverflow)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the timer/counter chip. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *chan* | **short**: timer/counter channel number within the chip (0, 1 or 2). Another timer/counter (*chan*–2) will also be used to provide the gate signal. This second timer/counter may be on the previous chip. |
| | *width* | **double**: width of the measurement gate in seconds. |
| | *pfreq* | **pointer to double**: pointer to a 64-bit floating-point variable into which the frequency result will be placed. |
| | *poverflow* | **pointer to short**: pointer to a variable used to store an overflow indication. A 16-bit counter is used to measure the frequency. This variable indicates whether or not the counter overflowed during the measurement period:<br><br>0 = not overflowed<br>1 = overflowed |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE<br>ERRDATA | |

| | |
|---|---|
| Prior Calls | registerBoardEx |
| See Also | TCgetExtFreq |

#### 6.4.9.4 Multiply an External Frequency — TCmultiplyFreq

Measures an external signal's frequency, then generates another signal whose frequency is the external frequency multiplied by a specified number. N.B. this function is not on-going, and must be called at a regular interval to keep the generated frequency tracking the external signal. Note that the output signal will be a square wave. See section 3.1.7 for more details on the Frequency Multiplication application.

```
i = TCmultiplyFreq (h, ipChip, ipChan, opChip, opChan, factor)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *ipChip* | **short**: address offset of the timer/counter chip on which the input frequency will be measured. One of the following pre-defined constants may be used:<br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *ipChan* | **short**: input timer/counter channel number within the chip on which to perform the frequency measurement (0, 1 or 2). Another timer/counter (*ipChan*–2) will also be used to provide the gate signal. This may be on the previous chip. |
| | *opChip* | **short**: address offset of the timer/counter chip on which to generate the output frequency. One of the following pre-defined constants may be used:<br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *opChan* | **short**: output timer/counter channel number within the chip (0, 1 or 2). |
| | *factor* | **float**: multiplication factor. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE | |

| | |
|---|---|
| Prior Calls | registerBoardEx |
| See Also | TCdivideFreq |

#### 6.4.9.5 Divide an External Frequency — TCdivideFreq

Measures an external signal's frequency, then generates another signal whose frequency is the external frequency divided by a specified number. N.B. this function is not on-going, and must be called at a regular interval to keep the generated frequency tracking the external signal. Note the output signal will be a square wave.

```
i = TCdivideFreq (h, ipChip, ipChan, opChip, opChan, divisor)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *ipChip* | **short**: address offset of the timer/counter chip on which the input frequency will be measured. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *ipChan* | **short**: input timer/counter channel number within the chip on which to perform the frequency measurement (0, 1 or 2). Another timer/counter (*ipChan*–2) will also be used to provide the gate signal. This may be on the previous chip. |
| | *opChip* | **short**: address offset of the timer/counter chip on which to generate the output frequency. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *opChan* | **short**: output timer/counter channel number within the chip (0, 1 or 2). |
| | *divisor* | **float**: division factor. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |

| | |
|---|---|
| Prior Calls | registerBoardEx |
| See Also | TCmultiplyFreq |

### 6.4.10 Digitally Controlled Oscillator Functions

#### 6.4.10.1 Prepare a Digitally-Controlled Oscillator — TCsetDCO

Implements a digitally controlled oscillator (DCO) which periodically reads a data value from a digital input channel and generates an external frequency based on the value. The digital input channel can be 1, 4, 8, 12, 16, or 24-bits wide, as specified by a previous call to function DIOsetChanWidth. The digital channel must have already been set up as an input with a call to function DIOsetMode. See section 3.4.2 for more details on the Digitally Controlled Oscillator application.

```
i = TCsetDCO (h, diChip, diChan, opChip, opChan, udFreq, udChip,
MinF, MaxF)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*diChip* — **short**: address offset of the digital input chip. One of the following pre-defined constants may be used:

PPIX = 0
PPIY = 8
PPIZ = 16

*diChan* — **short**: digital input channel.

*opChip* — **short**: address offset of the timer/counter chip to be used for frequency output. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12
Z1 = 16
Z2 = 20

*opChan* — **short**: frequency output timer/counter channel number within the chip (0, 1 or 2).

*udFreq* — **float**: update frequency in Hertz.

*udChip* — **short**: address offset of a timer/counter chip of which counter channel 1 will be used to generate the update interrupts. One of the following pre-defined constants may be used:

X1 = 0
X2 = 4
Y1 = 8
Y2 = 12

| | | |
|---|---|---|
| | | Z1 = 16<br>Z2 = 20 |
| | *MinF* | **float**: output frequency corresponding to DI data value 0. |
| | *MaxF* | **float**.: frequency corresponding to the maximum digital input data value, which itself depends on the channel width specified in DIOsetChanWidth. |
| Returns **short**: | DCO handle (>= 0). Use this handle to call TCfreeDCO when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCsetUserCO<br>enableInterrupts<br>TCfreeDCO | |

#### 6.4.10.2 Prepare a User-Controlled Oscillator — TCsetUserCO

Implements a user-controlled oscillator that periodically calls a user-supplied function. The user function may set the frequency of the oscillator using TCsetUserCOLevel.

```
i = TCsetUserCO (h, pfn, opChip, opChan, udFreq, udChip, MinF, MaxF)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long) returning void**: A pointer to a function implemented in the user's code that has the format of a TCUserCOCallback as defined below. |
| | *opChip* | **short**: address offset of the timer/counter chip to be used for frequency output. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *opChan* | **short**: frequency output timer/counter channel number within the chip (0, 1 or 2). |
| | *udFreq* | **float**: update frequency in Hertz. |
| | *udChip* | **short**: address offset of a timer/counter chip of which counter channel 1 will be used to |

| | | |
|---|---|---|
| | | generate the update interrupts. One of the following pre-defined constants may be used:<br><br>X1 = 0<br>X2 = 4<br>Y1 = 8<br>Y2 = 12<br>Z1 = 16<br>Z2 = 20 |
| | *MinF* | **float**: output frequency corresponding to user value 0. |
| | *MaxF* | **float**.: output frequency corresponding to user value 2147483647 ($7FFFFFFF_{16}$). |
| Returns **short**: | User CO handle (>= 0). Use this handle to call TCfreeDCO when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCUserCOCallback<br>enableInterrupts<br>TCfreeDCO | |

#### 6.4.10.3 User Controlled Oscillator Callback — TCUserCOCallback

Function to be implemented in the user's code. The user will need to pass a pointer to the function (which has a user-supplied name) to TCsetUserCO. It must be declared as 'CALLBACK' . It is called following an update timer interrupt. This function can be used to adjust the frequency output of the user-controlled oscillator using TCsetUserCOLevel.

**`TCUserCOCallback (h, hCO, count)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hCO* | **short**: handle of user controlled oscillator as issued by the TCsetUserCO function. |
| | *count* | **unsigned long**: counter value read from the timer/counter channel used as the update counter. This value is likely to be of no use whatsoever. |
| Returns **void**. | | |
| Prior Calls | registerBoardEx<br>TCsetUserCO<br>enableInterrupts | |
| See Also | TCsetUserCOLevel | |

#### 6.4.10.4 Set User Controlled Oscillator Output Level — TCsetUserCOLevel

This function allows the user to set the frequency of the controlled oscillator declared using TCsetUserCO. Provided so that the output frequency of the controlled oscillator can be set as part of the user callback function; however, it can be called at any time.

```
i = TCsetUserCOLevel (h, hCO, value)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hCO* | **short**: handle of user controlled oscillator as issued by the TCsetUserCO function. |
| | *value* | **unsigned long**: value between 0 and 2147483647 ($7FFFFFFF_{16}$) representing the desired frequency. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |
| Prior Calls | registerBoardEx<br>TCsetUserCO | |
| See Also | TCUserCOCallback<br>TCfreeDCO | |

#### 6.4.10.5 Free-up a DCO or User CO's Timer/Counters — TCfreeDCO

Frees the timer/counter & DIO resources used by a DCO, or User CO, as previously set up by the TCsetDCO or TCsetUserCO function. Call this function when you've finished using the DCO or User CO.

```
i = TCfreeDCO (h, hO)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hO* | **short**: DCO handle or User CO handle, as issued by the TCsetDCO or TCsetUserCO function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetDCO<br>TCsetUserCO | |
| See Also | disableInterrupts | |

## 6.4.11 Digital Input/Output Functions

### 6.4.11.1 Test if Digital I/O Chip is Free — DIOisAvailable

Checks if a particular Digital I/O (DIO) chip is available on a board. A DIO chip may not be available for one of two reasons:

1. the DIO chip is not provided by the board specified, or
2. the DIO chip is being used by some other function.

```
i = DIOisAvailable (h, chip)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| Returns **short**: | 0 = DIO Chip NOT Available, 1 = Available | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | | |

### 6.4.11.2 Configure a Digital I/O Port for Input or Output — DIOsetMode

Sets up a digital I/O port for basic input or output.

Note that all output ports on the chip will be set to logic level 0, including those not directly configured by the function call. This is a feature of the 82C55 chip.

```
i = DIOsetMode (h, chip, port, isInput)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *port* | **short**: DIO port within the chip. Port C is split into two 4-bit nibbles, which can be programmed independently. One of the following pre-defined constants may be used:<br><br>PORTA = 0 |

| | | |
|---|---|---|
| | | PORTB = 1<br>PORTC_L = 2<br>PORTC_U = 3 |
| | *isInput* | **short**: non-zero if port is to be set as input, zero if port is to be set as output. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | DIOsetChanWidth<br>DIOsetData<br>DIOgetData<br>DIOgetMode | |

#### 6.4.11.3 Check Digital I/O Port Direction — DIOgetMode

Indicates whether a digital I/O port is an input or an output.

SUPPORTED IN VERSION 4.40 ONWARDS.

```
i = DIOgetMode (h, chip, port, pIsInput)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *port* | **short**: DIO port within the chip. Port C is split into two 4-bit nibbles, which can be programmed independently. One of the following pre-defined constants may be used:<br><br>PORTA = 0<br>PORTB = 1<br>PORTC_L = 2<br>PORTC_U = 3 |
| | *pIsInput* | **pointer to short**: pointer to a variable into which a port direction value will be placed. The port direction value is as follows:<br><br>0: output<br>1: input |
| Returns **short**: | OK | |
| or | ERRHANDLE | |

| | |
|---|---|
| | ERRCHAN |
| Prior Calls | registerBoardEx |
| See Also | DIOsetMode |

#### 6.4.11.4 Re-define Channel Width within a Digital I/O Chip — DIOsetChanWidth

Redefines the number of bits per DIO channel to be used in subsequent calls to the DIOsetData and DIOgetData functions. The default channel width is 8-bits, and this can be changed to 1, 4, 8, 12, 16, or 24. After calling this function, the chan argument in the DIOsetData and DIOgetData functions refers to the group of bits of width *numBits*, starting at Port A bit 0. For a channel width of 12, port C-Upper forms the upper 4 bits of channel 0 and port C-Lower forms the upper 4 bits of channel 1. Note that the three ports (A, B, C-Upper and C-Lower) must be set up correctly for input or output accordingly by calling the DIOsetMode function for each.

```
i = DIOsetChanWidth (h, chip, numBits)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*chip* — **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:

PPIX = 0
PPIY = 8
PPIZ = 16

*numBits* — **short**: bit width to be used in subsequent calls to functions DIOsetData, DIOgetData and TCsetDCO. Valid widths are 1, 4, 8, 12, 16, or 24.

| *numBits* | *channels per chip* |
|---|---|
| 1 | 24 |
| 4 | 6 |
| 8 | 3 |
| 12 | 2 |
| 16 | 1 |
| 24 | 1 |

| | |
|---|---|
| Returns **short**: | OK |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx |
| See Also | DIOsetMode<br>DIOsetData<br>DIOgetData |

#### 6.4.11.5 Send Digital Output Data — DIOsetData

Writes a data value to a DIO channel. It is assumed that the channel has already been set as an output by a call to function DIOsetMode.

```
i = DIOsetData (h, chip, chan, data)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *chan* | **short**: DIO channel. Note the channel numbering depends on the channel width as set by DIOsetChanWidth (default is three 8-bit channels). |
| | *data* | **long**: digital data word. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>DIOsetMode<br>DIOsetChanWidth | |
| See Also | DIOgetData | |

#### 6.4.11.6 Read Digital Input Data — DIOgetData

Reads a data value from a DIO channel. It is assumed that the channel has already been set as an input by a call to function DIOsetMode.

```
i = DIOgetData (h, chip, chan, pdata)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *chan* | **short**: DIO channel. Note the channel numbering depends on the channel width as set by DIOsetChanWidth (default is three 8-bit channels). |

| | | |
|---|---|---|
| | *pdata* | **pointer to long**: pointer to a long integer variable into which the digital data word will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>DIOsetMode<br>DIOsetChanWidth | |
| See Also | DIOsetData | |

#### 6.4.11.7 Configure a Digital I/O Port Mode — DIOsetModeEx

Writes directly to the digital I/O port control register. The value written may be a mode-setting command (bit 7 = 1) or a single bit set/reset command (bit 7 = 0).

SUPPORTED IN VERSION 2.00 ONWARDS.

Note that writing a mode-setting command (bit 7 = 1) causes all output ports on the chip to be set to logic level 0 (except for certain PORT C bits in modes 1 and 2). This is a feature of the 82C55 chip.

```
i = DIOsetModeEx (h, chip, ctrl)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | ctrl | **short**: value to write to PPI control port. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | DIOsetDataEx<br>DIOgetDataEx<br>DIOgetModeEx | |

#### 6.4.11.8 Check a Digital I/O Port's Mode — DIOgetModeEx

Checks the last mode-setting command written to a digital I/O port's control register.

SUPPORTED IN VERSION 4.40 ONWARDS.

Note: if this is used with a hardware device driver prior to version 4.40, the function will get the last mode-setting command or single bit set/reset command sent to the digital I/O port's control register. To be on the safe side, the application should check that bit 7 of the returned data is set to 1 before interpreting it as a mode value.

```
i = DIOgetModeEx (h, chip, pdata)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *pdata* | **pointer to short**: pointer to a short integer variable into which the mode value will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | DIOsetModeEx | |

#### 6.4.11.9 Write to Digital Output Port — DIOsetDataEx

Writes a data value to a DIO port directly.

SUPPORTED IN VERSION 2.00 ONWARDS.

```
i = DIOsetDataEx (h, chip, port, data)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *port* | **short**. DIO Port, 0,1 or 2. |
| | *data* | **short**: digital data value. |
| Returns **short**: | OK | |

| | |
|---|---|
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx<br>DIOsetMode<br>DIOsetModeEx |
| See Also | DIOgetDataEx |

#### 6.4.11.10 Read Digital Input Data Port — DIOgetDataEx

Reads a data value from a DIO port.

SUPPORTED IN VERSION 2.00 ONWARDS.

```
i = DIOgetDataEx (h, chip, port, pdata)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *chip* | **short**: address offset of the DIO chip. One of the following pre-defined constants may be used:<br><br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16 |
| | *port* | **short**. DIO Port, 0,1 or 2. |
| | *pdata* | **pointer to short**: pointer to a short integer variable which the digital data value will be placed. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>DIOsetMode<br>DIOsetModeEx | |
| See Also | DIOsetDataEx | |

### 6.4.12 Switch Scanner Matrix Functions

#### 6.4.12.1 Set up a Switch Scanner Matrix — DIOsetSwitchMatrix

Sets up one, two or three 82C55 DIO chips as a switch matrix scanning device. The *order* of the matrix specified can be 12 (for a 12 X 12 matrix scanning 144 switches, using PPIX), 24 (for a 24 X 24 matrix scanning 576 switches, using PPIX and PPIY), or 36 (for a 36 X 36 matrix scanning 1296 switches, using PPIX, PPIY and PPIZ). Group A (ports A and C-upper) are set for output, to send test patterns to the matrix, and group B (port B and C-lower) are set for input to read the switch status information back in. The user must ensure that the switch

array is wired correctly with suitable diodes and resistors, otherwise the board could get damaged. See section 3.2.2 for details. Only one switch matrix implementation is available per board.

**`i = DIOsetSwitchMatrix (h, order)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function<br>. |
| | *order* | **short**: order of the matrix (12, 24 or 36). |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | DIOgetSwitchStatus<br>DIOfreeSwitchMatrix | |

#### 6.4.12.2 Query Status of a Switch within the Scan Matrix — DIOgetSwitchStatus

Queries the status of a particular switch in the switch matrix set up by the DIOsetSwitchMatrix function. The grid reference of the switch is given, and the function performs a test on that switch and returns 1 for switch on (closed) or 0 for switch off (open).

**`i = DIOgetSwitchStatus (h, xcoord, ycoord)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *xcoord* | **short**: X-co-ordinate of the position of the switch in the matrix (origin is at port A0/B0 of PPIX). Valid values should be in the range 0 – *order* (as specified in DIOsetSwitchMatrix). |
| | *ycoord* | **short**: Y-co-ordinate of the position of the switch in the matrix (origin is at port A0/B0 of PPIX). Valid values should be in the range 0 – *order* (as specified in DIOsetSwitchMatrix). |
| Returns **short**: | Zero, if switch was OFF (open). Non-zero if switch was ON (closed). | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>DIOsetSwitchMatrix | |
| See Also | DIOfreeSwitchMatrix | |

#### 6.4.12.3 Free-up the Digital I/O Chip(s) from a Switch Matrix — DIOfreeSwitchMatrix

Frees the DIO resources used by the switch matrix as set up in function DIOsetSwitchMatrix.

**`i = DIOfreeSwitchMatrix (h)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>DIOsetSwitchMatrix | |
| See Also | | |

### 6.4.13 Basic User Interrupt Callbacks

#### 6.4.13.1 Prepare a Basic User Interrupt — TCsetUserInterrupt

Used to register a callback function that will be called when a particular interrupt event occurs.

This does not support write events due to the interface to the user callback function. For write events, use the buffered user interrupt functions instead.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

**`i = TCsetUserInterrupt (h, pfn, wParam, Chip, ISRDATA, Chip1, Chan1, Chip2, Chan2)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long) returning void**: a pointer to a function implemented in the user's code that has the format of a TCUserCCallback as defined below. |
| | *wParam* | **unsigned integer**: user-supplied value passed to the user's callback function. |
| | *Chip* | **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined |

| | | |
|---|---|---|
| | | constants may be used:<br><br>X1 = 0 PPIXC0 = 0<br>X2 = 4 PPIXC3 = 4<br>Y1 = 8 PPIYC0 = 8<br>Y2 = 12 PPIYC3 = 12<br>Z1 = 16 PPIZC0 = 16<br>Z2 = 20 PPIZC3 = 20<br>PPIX = 0 PPIYC7 = 8<br>PPIY = 8 EXT0 = 0<br>PPIZ = 16 ADC0 = 0<br>ADC2 = 8<br>DAC2 = 8<br>DAC4 = 16<br>SATRIG = 12 |
| | *ISRDATA* | **short**: type of data to fetch on interrupt. The following pre-defined constants may be used:<br><br>ISR_NODATA = –1<br>ISR_READ_16COUNT = 0<br>ISR_READ_16COUNTSTAT = 16<br>ISR_READ_32COUNT = 1<br>ISR_READ_PPIABC = 5<br>ISR_READ_PPIC = 6<br>ISR_PC27 = 7<br>ISR_READ_DATA8 = 8<br>ISR_READ_DATA16 = 9 |
| | *Chip1* | **short**: address offset of the first timer/counter or PPI chip to be interrogated:<br><br>X1 = 0 X2 = 4 PPIX = 0<br>Y1 = 8 Y2 = 12 PPIY = 8<br>Z1 = 16 Z2 = 20 PPIZ = 16 |
| | *Chan1* | **short**: first timer/counter channel or PPI port to interrogate (0, 1, 2). |
| | *Chip2* | **short**: address offset of the second timer/counter or PPI chip to be interrogated:<br><br>X1 = 0 X2 = 4 PPIX = 0<br>Y1 = 8 Y2 = 12 PPIY = 8<br>Z1 = 16 Z2 = 20 PPIZ = 16 |
| | *Chan2* | **short**: second timer/counter channel or PPI port to interrogate (0, 1, 2) |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCUserCCallback | |

| | |
|---|---|
| | TCfreeUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetNCBufferUserInterrupt<br>enableInterrupts<br>disableInterrupts |

#### 6.4.13.2 Prepare a Basic User Interrupt for Analogue Input — TCsetUserInterruptAIO

Used to register a callback function that will be called when a particular interrupt event occurs. This variant is used to support reading from analogue channels.

SUPPORTED IN VERSION 4.00 ONWARDS.

This does not support write events due to the interface to the user callback function. For write events, use the buffered user interrupt functions instead.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetUserInterruptAIO (h, pfn, wParam, Chip, ISRDATA, Group, ChMask)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long) returning void**: a pointer to a function implemented in the user's code that has the format of a TCUserCCallback as defined below. |
| | *wParam* | **unsigned integer**: user-supplied value passed to the user's callback function. |
| | *Chip* | **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:<br><br>X1 = 0 PPIXC0 = 0<br>X2 = 4 PPIXC3 = 4<br>Y1 = 8 PPIYC0 = 8<br>Y2 = 12 PPIYC3 = 12<br>Z1 = 16 PPIZC0 = 16 |

| | | |
|---|---|---|
| | | Z2 = 20<br>PPIX = 0<br>PPIY = 8<br>PPIZ = 16<br>PPIZC3 = 20<br>PPIYC7 = 8<br>EXT0 = 0<br>ADC0 = 0<br>ADC2 = 8<br>DAC2 = 8<br>DAC4 = 16<br>SATRIG = 12 |
| | *ISRDATA* | **short**: type of data to fetch on interrupt. The following pre-defined constants may be used:<br><br>ISR_READ_ADCS = 10<br>ISR_READ_ADCSNOFIFO = 11<br>ISR_READ_ADCSFIFO = 12<br>ISR_READ_ADCSASAP = 15 |
| | *Group* | **short**: channel group of ADC channels to read. |
| | *ChMask* | **unsigned long**: bit-mask of ADC channels to read; LSB corresponds to channel 0; MSB corresponds to channel 32; bit value '1' means the corresponding channel will be read.<br><br>N.B. channels will be read cyclically, one channel each time. |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCUserCCallback<br>TCfreeUserInterrupt<br>TCsetUserInterrupt<br>TCsetUserInterrupt2<br>TCsetBufferUserInterruptAIO<br>TCsetNCBufferUserInterruptAIO<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.13.3 Prepare a Basic User Interrupt for Miscellaneous Input — TCsetUserInterrupt2

Used to register a callback function that will be called when a particular interrupt event occurs. This variant is reserved for future use.

SUPPORTED IN VERSION 4.00 ONWARDS.

This does not support write events due to the interface to the user callback function. For write events, use the buffered user interrupt functions instead.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are

automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetUserInterrupt2 (h, pfn, wParam, Chip, ISRDATA, Block1,
Port1, Block2, Port2)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*pfn* — **pointer to function (short, unsigned int, unsigned long) returning void**: a pointer to a function implemented in the user's code that has the format of a TCUserCCallback as defined below.

*wParam* — **unsigned integer**: user-supplied value passed to the user's callback function.

*Chip* — **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*ISRDATA* — **short**: type of data to fetch on interrupt.

*Block1* — **unsigned long**: usage depends on *ISRDATA* value.

*Port1* — **unsigned long**: usage depends on *ISRDATA* value.

*Block2* — **unsigned long**: usage depends on *ISRDATA* value.

*Port2* — **unsigned long**: usage depends on *ISRDATA* value.

| | |
|---|---|
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx |
| See Also | TCUserCCallback<br>TCfreeUserInterrupt<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts<br>disableInterrupts |

#### 6.4.13.4 Basic User Interrupt Callback — TCUserCCallback

Function to be implemented in the user's code. The user will need to pass a pointer to the function (which has a user-supplied name) to TCsetUserInterrupt, TCsetUserInterruptAIO or TCsetUserInterrupt2. It must be declared as 'CALLBACK'. It is called following an interrupt.

```
TCUserCCallback (h, wParam, lParam)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *wParam* | **unsigned integer**: value as supplied by user to TCsetUserInterrupt function. |
| | *lParam* | **unsigned long**: value read on interrupt. |

Returns **void**.

| | |
|---|---|
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>enableInterrupts |
| See Also | |

#### 6.4.13.5 Free up a User Interrupt — TCfreeUserInterrupt

Frees up a user-interrupt set with TCsetUserInterrupt, TCsetUserInterruptAIO, TCsetUserInterrupt2, TCsetBufferUserInterrupt, TCsetBufferUserInterruptAIO, TCsetBufferUserInterrupt2, TCsetNCBufferUserInterrupt, TCsetNCBufferUserInterruptAIO, TCsetNCBufferUserInterrupt2 functions.

```
i = TCfreeUserInterrupt (h, hUsrInt)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |

| | | |
|---|---|---|
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2 | |
| See Also | | |

### 6.4.14 Buffered User Interrupt Callbacks

#### 6.4.14.1 Prepare a Buffered User Interrupt — TCsetBufferUserInterrupt

Used to register a callback function that will be called to process a buffers-worth of data that has been read from or is to be written to the card over a number of interrupts.

SUPPORTED IN VERSION 3.00 ONWARDS

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetBufferUserInterrupt (h, pfn, wParam, Chip, SizeReq, fContin,
ISRDATA, Chip1, Chan1, Chip2, Chan2)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long, pointer to unsigned long) returning void**: a pointer to a function implemented in the user's code that has the format of a TCUserCBCallback as defined below. |
| | *wParam* | **unsigned integer**: user-supplied value passed to the user's callback function. |
| | *Chip* | **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter |

interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*ISRDATA* — **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:

ISR_NODATA = –1
ISR_READ_16COUNT = 0
ISR_READ_16COUNTSTAT = 16
ISR_READ_32COUNT = 1
ISR_READ_32COUNTSTAT = 17
ISR_READ_PPIABC = 5
ISR_READ_PPIC = 6
ISR_PC27 = 7
ISR_READ_DATA8 = 8
ISR_READ_DATA16 = 9
ISR_READ_2PPIABC = 13
ISR_READ_3PPIABC = 14
ISR_WRITE_DATA8 = 32
ISR_WRITE_DATA16 = 33
ISR_WRITE_PPIABC = 34
ISR_WRITE_16COUNT = 39
ISR_WRITE_32COUNT = 40
ISR_WRITE_2PPIABC = 41
ISR_WRITE_3PPIABC = 42

*SizeReq* — **unsigned long**: number of data values in each buffer.

*fContin* — **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero.

*Chip1* — **short**: address offset of the first timer/counter or PPI chip to be read or written:

| | | |
|---|---|---|
| X1 = 0 | X2 = 4 | PPIX = 0 |
| Y1 = 8 | Y2 = 12 | PPIY = 8 |
| Z1 = 16 | Z2 = 20 | PPIZ = 16 |

| | | |
|---|---|---|
| | *Chan1* | **short**: first timer/counter channel or PPI port to read or write (0, 1, 2). |
| | *Chip2* | **short**: address offset of the second timer/counter or PPI chip to be read or written:<br>X1 = 0 X2 = 4 PPIX = 0<br>Y1 = 8 Y2 = 12 PPIY = 8<br>Z1 = 16 Z2 = 20 PPIZ = 16 |
| | *Chan2* | **short**: second timer/counter channel or PPI port to read or write (0, 1, 2) |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCUserCBCallback<br>TCfreeUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetUserInterrupt<br>TCsetNCBufferUserInterrupt<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.14.2 Prepare a Buffered User Interrupt for Analogue I/O — TCsetBufferUserInterruptAIO

Used to register a callback function that will be called to process a buffers-worth of data that has been read from or is to be written to the card over a number of interrupts. This variant is used to support reading from or writing to analogue channels.

SUPPORTED IN VERSION 4.00 ONWARDS.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetBufferUserInterruptAIO (h, pfn, wParam, Chip, SizeReq,
fContin, ISRDATA, Group, ChMask)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long, pointer to unsigned long) returning void**: a pointer to a function implemented in the user's code that has the |

format of a TCUserCBCallback as defined below.

*wParam* — **unsigned integer**: user-supplied value passed to the user's callback function.

*Chip* — **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*SizeReq* — **unsigned long**: number of data values in each buffer.

*fContin* — **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero.

*ISRDATA* — **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:

ISR_READ_ADCS = 10
ISR_READ_ADCSNOFIFO = 11
ISR_READ_ADCSFIFO = 12
ISR_READ_ADCSASAP = 15
ISR_WRITE_DACS = 35
ISR_WRITE_DACSNOFIFO = 37
ISR_WRITE_DACSFIFO = 38

*Group* — **short**: channel group of ADC channels to read or DAC channels to be written.

*ChMask* — **unsigned long**: bit-mask of ADC channels to read or DAC channels to be written; LSB corresponds to channel 0; MSB corresponds to channel 32; bit value '1' means the corresponding channel will be read.

| | | |
|---|---|---|
| | | N.B. ADC channels will be read cyclically, one channel each time; DAC channels will be written a whole frame (all selected channels) at a time. |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCUserCBCallback<br>TCfreeUserInterrupt<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterrupt2<br>TCsetUserInterruptAIO<br>TCsetNCBufferUserInterruptAIO<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.14.3 Prepare a Buffered User Interrupt for Miscellaneous I/O — TCsetBufferUserInterrupt2

Used to register a callback function that will be called to process a buffers-worth of data that has been read from or is to be written to the card over a number of interrupts. This variant is used to support types of data transfer that do not fit the parameters of TCsetBufferUserInterrupt or TCsetBufferUserInterruptAIO.

SUPPORTED IN VERSION 4.00 ONWARDS.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetBufferUserInterrupt2 (h, pfn, wParam, Chip, SizeReq,
fContin, ISRDATA, Block1, Port1, Block2, Port2)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *pfn* | **pointer to function (short, unsigned int, unsigned long, pointer to unsigned long) returning void**: a pointer to a function implemented in the user's code that has the format of a TCUserCBCallback as defined below. |
| | *wParam* | **unsigned integer**: user-supplied value passed to the user's callback function. |
| | *Chip* | **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied |

by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*SizeReq* — **unsigned long**: number of data values in each buffer.

*fContin* — **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero.

*ISRDATA* — **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:

ISR_WRITE_2DACS = 36

*Block1* — **unsigned long**: usage depends on *ISRDATA* value.

For ISR_WRITE_2DACS: the channel group number of the first DAC to write to.

*Port1* — **unsigned long**: usage depends on *ISRDATA* value.

For ISR_WRITE_2DACS: the channel number (within the group) of the first DAC to write to.

*Block2* — **unsigned long**: usage depends on *ISRDATA* value.

For ISR_WRITE_2DACS: the channel group number of the second DAC to write to.

*Port2* — **unsigned long**: usage depends on *ISRDATA* value.

For ISR_WRITE_2DACS: the channel number (within the group) of the second DAC to write

| | |
|---|---|
| | to. |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx |
| See Also | TCUserCBCallback<br>TCfreeUserInterrupt<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts<br>disableInterrupts |

#### 6.4.14.4 Buffered User Interrupt Callback — TCUserCBCallback

Function to be implemented in the user's code. The user will need to pass a pointer to the function (which has a user-supplied name) to TCsetBufferUserInterrupt, TCsetBufferUserInterruptAIO or TCsetBufferUserInterrupt2. It must be declared as 'CALLBACK'. It is called when a buffers-worth of data has been read from the card during interrupt processing, or another buffers-worth of data is required from the user to be written to the card during interrupt processing.

SUPPORTED IN VERSION 3.00 ONWARDS.

**`TCUserCBCallback (h, wParam, BufSize, pBuffer)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *wParam* | **unsigned integer**: value as supplied by user to TCsetBufferUserInterrupt function. |
| | *BufSize* | **unsigned long**: number of data values which can be read from the buffer, or which must be written to the buffer by the user. |
| | *pBuffer* | **pointer to unsigned long**: points to start of buffer. |
| Returns **void**. | | |
| Prior Calls | registerBoardEx<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>enableInterrupts | |
| See Also | | |

### 6.4.15 Non-Callback Buffered User Interrupts

#### 6.4.15.1 Prepare a Non-Callback Buffered User Interrupt — TCsetNCBufferUserInterrupt

Called to set up buffered user interrupts without callbacks and without creating any new threads. Instead of callbacks, data is transferred by the application calling the TCdriveNCBufferUserInterrupt function. This function may be used by a HP VEE application.

SUPPORTED IN VERSION 4.00 ONWARDS.

N.B. An incompatible version of this function was included in the unreleased Version 3.00.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetNCBufferUserInterrupt (h, Chip, SizeReq, fContin, ISRDATA,
Chip1, Chan1, Chip2, Chan2)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*Chip* — **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*ISRDATA* — **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:

ISR_NODATA = –1
ISR_READ_16COUNT = 0

ISR_READ_16COUNTSTAT = 16
ISR_READ_32COUNT = 1
ISR_READ_32COUNTSTAT = 17
ISR_READ_PPIABC = 5
ISR_READ_PPIC = 6
ISR_PC27 = 7
ISR_READ_DATA8 = 8
ISR_READ_DATA16 = 9
ISR_READ_2PPIABC = 13
ISR_READ_3PPIABC = 14
ISR_WRITE_DATA8 = 32
ISR_WRITE_DATA16 = 33
ISR_WRITE_PPIABC = 34
ISR_WRITE_16COUNT = 39
ISR_WRITE_32COUNT = 40
ISR_WRITE_2PPIABC = 41
ISR_WRITE_3PPIABC = 42

*SizeReq* — **unsigned long**: number of data values in each buffer.

*fContin* — **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero.

*Chip1* — **short**: address offset of the first timer/counter or PPI chip to be read or written:

| | | |
|---|---|---|
| X1 = 0 | X2 = 4 | PPIX = 0 |
| Y1 = 8 | Y2 = 12 | PPIY = 8 |
| Z1 = 16 | Z2 = 20 | PPIZ = 16 |

*Chan1* — **short**: first timer/counter channel or PPI port to read or write (0, 1, 2).

*Chip2* — **short**: address offset of the second timer/counter or PPI chip to be read or written:

| | | |
|---|---|---|
| X1 = 0 | X2 = 4 | PPIX = 0 |
| Y1 = 8 | Y2 = 12 | PPIY = 8 |
| Z1 = 16 | Z2 = 20 | PPIZ = 16 |

*Chan2* — **short**: second timer/counter channel or PPI port to read or write (0, 1, 2)

Returns **short**: User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished.

or
ERRHANDLE
ERRCHAN
ERRDATA

Prior Calls: registerBoardEx

See Also:
TCdriveNCBufferUserInterrupt
TCwaitNCBufferReady
TCwaitMultiNCBufferReady
TCfreeUserInterrupt
TCsetNCBufferUserInterruptAIO
TCsetNCBufferUserInterrupt2

| | |
|---|---|
| | TCsetUserInterrupt<br>TCsetBufferUserInterrupt<br>enableInterrupts<br>disableInterrupts |

#### 6.4.15.2 Prepare a Non-Callback Buffered User Interrupt for Analogue I/O — TCsetNCBufferUserInterruptAIO

Called to set up buffered user interrupts without callbacks and without creating any new threads. Instead of callbacks, data is transferred by the application calling the TCdriveNCBufferUserInterrupt function. This function may be used by a HP VEE application. This variant is used to support reading from or writing to analogue channels.

SUPPORTED IN VERSION 4.00 ONWARDS.

N.B. An incompatible version of this function was included in the unreleased Version 3.00.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetNCBufferUserInterruptAIO (h, Chip, SizeReq, fContin,
ISRDATA, Group, ChMask)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*Chip* — **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*SizeReq* — **unsigned long**: number of data values in each

| | | |
|---|---|---|
| | | buffer. |
| | *fContin* | **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero. |
| | *ISRDATA* | **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:<br><br>ISR_READ_ADCS = 10<br>ISR_READ_ADCSNOFIFO = 11<br>ISR_READ_ADCSFIFO = 12<br>ISR_READ_ADCSASAP = 15<br>ISR_WRITE_DACS = 35<br>ISR_WRITE_DACSNOFIFO = 37<br>ISR_WRITE_DACSFIFO = 38 |
| | *Group* | **short**: channel group of ADC channels to read or DAC channels to be written. |
| | *ChMask* | **unsigned long**: bit-mask of ADC channels to read or DAC channels to be written; LSB corresponds to channel 0; MSB corresponds to channel 32; bit value '1' means the corresponding channel will be read.<br><br>N.B. ADC channels will be read cyclically, one channel each time; DAC channels will be written a whole frame (all selected channels) at a time. |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCdriveNCBufferUserInterrupt<br>TCwaitNCBufferReady<br>TCwaitMultiNCBufferReady<br>TCfreeUserInterrupt<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterrupt2<br>TCsetUserInterruptAIO<br>TCsetBufferUserInterruptAIO<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.15.3 Prepare a Non-Callback Buffered User Interrupt for Miscellaneous I/O — TCsetNCBufferUserInterrupt2

Called to set up buffered user interrupts without callbacks and without creating any new threads. Instead of callbacks, data is transferred by the application calling the TCdriveNCBufferUserInterrupt function. This function may be used by a HP VEE application. This variant is used to support types of data transfer that do not fit the parameters of

TCsetNCBufferUserInterrupt or TCsetNCBufferUserInterruptAIO.

SUPPORTED IN VERSION 4.00 ONWARDS.

If the interrupt source is enabled at the first and second levels, it will be activated. Interrupts are initially disabled at the first level; use enableInterrupts to enable them. In versions of the DLL up to version 4.39, all interrupt sources are initially disabled at the second level but are automatically enabled by this interrupt set-up function. In versions of the DLL from 4.40 onwards, all valid interrupt sources are initially enabled at the second level but are no longer automatically enabled by this interrupt set-up function. In either case, if interrupt sources have not been explicitly disabled at the second level, there is no need to explicitly enable them.

```
i = TCsetNCBufferUserInterrupt2 (h, Chip, SizeReq, fContin, ISRDATA,
Block1, Port1, Block2, Port2)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*Chip* — **short**: determines interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For timer/counter interrupt or first interrupt line on a PPI chip it is the address offset of the chip. For second interrupt line on a PPI chip it is the address offset of the PPI chip plus 4. Corresponds to bit positions in interrupt enable register if the card has one, otherwise use the address offset of the interrupting chip. The following pre-defined constants may be used:

| | |
|---|---|
| X1 = 0 | PPIXC0 = 0 |
| X2 = 4 | PPIXC3 = 4 |
| Y1 = 8 | PPIYC0 = 8 |
| Y2 = 12 | PPIYC3 = 12 |
| Z1 = 16 | PPIZC0 = 16 |
| Z2 = 20 | PPIZC3 = 20 |
| PPIX = 0 | PPIYC7 = 8 |
| PPIY = 8 | EXT0 = 0 |
| PPIZ = 16 | ADC0 = 0 |
| | ADC2 = 8 |
| | DAC2 = 8 |
| | DAC4 = 16 |
| | SATRIG = 12 |

*SizeReq* — **unsigned long**: number of data values in each buffer.

*fContin* — **short**: use single buffer if zero, use double buffering (continuous mode) if non-zero.

*ISRDATA* — **short**: type of data transfer to be performed on interrupt. The following pre-defined constants may be used:

ISR_WRITE_2DACS = 36

*Block1* — **unsigned long**: usage depends on *ISRDATA* value.

| | | |
|---|---|---|
| | | For ISR_WRITE_2DACS: the channel group number of the first DAC to write to. |
| | *Port1* | **unsigned long**: usage depends on *ISRDATA* value.<br><br>For ISR_WRITE_2DACS: the channel number (within the group) of the first DAC to write to. |
| | *Block2* | **unsigned long**: usage depends on *ISRDATA* value.<br><br>For ISR_WRITE_2DACS: the channel group number of the second DAC to write to. |
| | *Port2* | **unsigned long**: usage depends on *ISRDATA* value.<br><br>For ISR_WRITE_2DACS: the channel number (within the group) of the second DAC to write to. |
| Returns **short**: | User Interrupt handle (>= 0). Use this to free the user interrupt with TCfreeUserInterrupt when finished. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx | |
| See Also | TCdriveNCBufferUserInterrupt<br>TCwaitNCBufferReady<br>TCwaitMultiNCBufferReady<br>TCfreeUserInterrupt<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt2<br>enableInterrupts<br>disableInterrupts | |

#### 6.4.15.4 Transfer Data for Non-Callback Buffered User Interrupt — TCdriveNCBufferUserInterrupt

Called to drive data through the interface that has been set up using TCsetNCBufferUserInterrupt, TCsetNCBufferUserInterruptAIO or TCsetNCBufferUserInterrupt2. The function will perform a blocking wait, if necessary, until one of the interrupt data buffers used for data transfer becomes available. The TCwaitNCBufferReady or TCwaitMultiNCBufferReady functions may be used to detect when a blocking wait would be performed. This function may be used by a HP VEE application.

SUPPORTED IN VERSION 4.00 ONWARDS.

N.B. An incompatible version of this function was included in the unreleased Version 3.0.

```
i = TCdriveNCBufferUserInterrupt (h, hUsrInt, pBuffer, pRetLen)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the |

| | | |
|---|---|---|
| | | registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by the TCsetNCBufferUserInterrupt function. |
| | *pBuffer* | **pointer to unsigned long**: pointer to the start of user's buffer with which to transfer data to or from one of the interrupt data buffers. The amount of data to be transferred is given by the value of the *SizeReq* parameter which was passed to the user interrupt set-up function. |
| | *pRetLen* | **pointer to unsigned long**: points to an unsigned long variable used for returned length result value. The returned length value is valid if the function returns OK. The returned length value is the same as *SizeReq* parameter which was passed to the user interrupt set-up function unless a failure occurred when reading data from the card, in which case the returned length value will be 0. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts | |
| See Also | TCwaitNCBufferReady<br>TCwaitMultiNCBufferReady | |

#### 6.4.15.5 Poll or Wait for Interrupt Data Buffer Ready for Non-Callback Buffered User Interrupt — TCwaitNCBufferReady

Called to poll non-callback buffered user interrupt to see if an interrupt data buffer is ready for data transfer using TCdriveNCBufferUserInterrupt. Also used to wait until an interrupt data buffer is ready with a timeout facility (a poll is implemented as a zero-length timeout). To wait until more than one non-callback buffered user interrupt has an interrupt data buffer available, use the TCwaitMultiNCBufferReady function instead. This function may be used by a HP VEE application.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = TCwaitNCBufferReady (h, hUsrInt, TOutMs)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by the TCsetNCBufferUserInterrupt function. |
| | *TOutMs* | **unsigned long**: maximum amount of time in |

| | | |
|---|---|---|
| | | milliseconds to wait for an interrupt data buffer to become available for transfer. Can be set to 0 for a poll or to INFINITE ($FFFFFFFF_{16}$) to wait indefinitely. |
| Returns **short**: | 0 if timed out; 1 if ready for data transfer. | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts | |
| See Also | TCdriveNCBufferUserInterrupt<br>TCwaitMultiNCBufferReady | |

#### 6.4.15.6 Poll or Wait for Interrupt Data Buffer Ready for Multiple Non-Callback Buffered User Interrupts — TCwaitMultiNCBufferReady

Called to wait until one of multiple non-callback buffered user interrupts has an interrupt data buffer that is ready for data transfer using TCdriveNCBufferUserInterrupt, with timeout facility. May also be used to poll without waiting (a poll is implemented as a zero-length timeout) and may also be used for a single non-callback buffered user interrupt, but it may be easier to use the TCwaitNCBufferReady function for those cases. This function may be used by a HP VEE application.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = TCwaitMultiNCBufferReady (nPairs, phBInArr, phUIInArr, phBout,
phUIOut, TOutMs)
```

| | | |
|---|---|---|
| where | *nPairs* | **unsigned long**: number of non-callback buffered user interrupts being checked. |
| | *phBInArr* | **pointer to short**: points to the first of an array of *nPairs* board handles as issued by the registerBoardEx function. Each index corresponds to one of the *nPairs* non-callback buffered user interrupts being checked. The function does not modify the contents of the array. |
| | *phUIInArr* | **pointer to short**: points to the first of an array of *nPairs* user interrupt handles as issued by the TCsetNCBufferUserInterrupt function. Each index corresponds to one of the *nPairs* non-callback buffered user interrupts being checked. The function does not modify the contents of the array. |
| | *phBOut* | **pointer to short**: pointer to short integer variable which will be set to –1 on timeout, or to the board handle of the first user interrupt which is ready for data transfer. |

| | | |
|---|---|---|
| | *phUIOut* | **pointer to short**: pointer to short integer variable which will be set to –1 on timeout, or to the user interrupt handle of the first user interrupt which is ready for data transfer. |
| | *TOutMs* | **unsigned long**: maximum amount of time in milliseconds to wait for an interrupt data buffer to become available for transfer on any of the user interrupts being checked. Can be set to 0 for a poll or to INFINITE ($FFFFFFFF_{16}$) to wait indefinitely. |
| Returns **short**: | 0 if timed out; 1 if ready for data transfer (variables pointed to by *phBOut* and *phUIOut* set to board handle and user interrupt handle, respectively). | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRMEMORY | |
| Prior Calls | registerBoardEx<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts | |
| See Also | TCdriveNCBufferUserInterrupt<br>TCwaitNCBufferReady | |

### 6.4.16 Miscellaneous Interrupt Handling Functions

#### 6.4.16.1 Check User Interrupt for Occurrence of Error — TCcheckUserInterruptError

Checks a previously set-up and enabled user interrupt to see if an overflow or under-run has occurred and clears the condition afterwards.

An overflow condition occurs when a user interrupt is reading data from a port or an ADC channel and a data sample fetched on a trigger could not be handled due to a FIFO full condition or lack of room in a user interrupt data buffer.

An under-run condition occurs when a user interrupt is writing data to a port or a DAC channel and no data is available when a trigger occurs, due to an empty FIFO condition or no data available in a user interrupt data buffer.

Enabling a user interrupt or checking the overflow or underflow condition with this function causes any such condition to be cleared (but not before checking the condition).

SUPPORTED IN VERSION 4.23 ONWARDS.

```
i = TCcheckUserInterruptError (h, hUsrInt)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |

| | |
|---|---|
| Returns **short**: | 0 = overflow or under-run NOT detected; 1 = overflow or under-run detected and cleared. |
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts |
| See Also | |

#### 6.4.16.2 Flush (Discard) User Interrupt Data — TCflushUserInterrupt

Flushes (discards) any user interrupt data that is stored in the driver including any data in FIFOs.

If the interrupt reads data into the user buffer from a device, the driver will reset to the start of the current buffer and reset the FIFO (if used).

If the interrupt writes data from the user buffer to a device, the driver will discard any remaining data in all queued buffers and reset the FIFO (if used).

When reading multiple ADC channels or writing multiple DAC channels, the correspondence between buffer positions and channels will be preserved subject to the following warnings.

**WARNING FOR PCI230 AND PCI260**: When using the ADC FIFO, it is possible for a sample to enter the FIFO just after it has been reset by this call. This would cause the correspondence between buffer positions and channels to be lost (they are shifted by one position). It is impossible for the driver to detect or prevent this condition. Therefore, if reading multiple channels from the ADC FIFO, use of this function must be avoided if there is any possibility that a conversion trigger could occur during the operation.

SUPPORTED IN VERSION 4.35 ONWARDS.

```
i = TCflushUserInterrupt (h, hUsrInt)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |

| | |
|---|---|
| | TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts |
| See Also | |

#### 6.4.16.3 Expedite Read User Interrupt — TCexpediteReadUserInterrupt

Cause current or following user interrupt data buffer to complete as soon as possible with as much data as possible without waiting for the buffer to fill.

SUPPORTED IN VERSION 5.02 ONWARDS.

The amount of data returned in the interrupt data buffer will be between 0 and the length of the buffer inclusive.

There is no point using this function with non-buffered user interrupts.

The function will fail for user interrupts that write data to the device, or if the driver version is too old, or if the user interrupt event is not currently enabled.

If the user interrupt reads multiple ADC channels and the buffer length is a multiple of the number of enabled ADC channels, the amount of data returned in the buffer will also be a multipls of the number of ADC channels. Any remaining ADC channel data will be saved for the next buffer in continuous mode, but will be discarded in non-continuous mode, preserving the channel sequence.

```
i = TCexpediteReadUserInterrupt (h, hUsrInt)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts | |
| See Also | TCcheckUserInterruptDataAvailable | |

#### 6.4.16.4 Check User Interrupt Data Available — TCcheckUserInterruptDataAvailable

Check the amount of user interrupt data available to be read.

SUPPORTED IN VERSION 5.02 ONWARDS.

Counts the amount of data in the current user interrupt data buffer and the amount of data int the driver's small, internal buffer. It also tries to count the amount of data available in hardware FIFOs.

The check for the amount of ADC data available in the FIFO is not very accurate for the original PCI230 and PCI260, but is more accurate for the PCI230+ and PCI260+.

The function will fail for user interrupts that write data to the device, or if the driver version is too old, or if the user interrupt event is not currently enabled.

```
i = TCcheckUserInterruptDataAvailable (h, hUsrInt, pDataAvail)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| | *pDataAvail* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to the amount of user interrupt data available. |
| Returns **short**: | OK. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2<br>enableInterrupts | |
| See Also | TCexpediteReadUserInterrupt | |

#### 6.4.16.5 Enable a User Interrupt — TCenableUserInterrupt

Enables a previously set-up interrupt source at the second level. An interrupt source is active when it is enabled at the first level (enableInterrupts) and at the second level and has been set-up.

SUPPORTED IN VERSION 4.40 UPWARDS

In versions of the DLL from version 4.40 onwards, all valid interrupt sources are initially enabled at the second level, so there is no need to call this function unless the interrupt

source has been explicitly disabled.

**`i = TCenableUserInterrupt (h, hUsrInt)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2<br>TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2 | |
| See Also | TCdisableUserInterrupt<br>setIntMask<br>getIntMask<br>TCenableInterruptChip | |

#### 6.4.16.6 Disable a User Interrupt — TCdisableUserInterrupt

Disables a previously set-up interrupt source at the second level. If the interrupt source is active, it will be deactivated.

SUPPORTED IN VERSION 4.40 UPWARDS

In versions of the DLL from version 4.40 onwards, all valid interrupt sources are initially enabled at the second level.

**`i = TCdisableUserInterrupt (h, hUsrInt)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *hUsrInt* | **short**: user interrupt handle as issued by user interrupt set-up function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>TCsetUserInterrupt<br>TCsetUserInterruptAIO<br>TCsetUserInterrupt2 | |

| | |
|---|---|
| | TCsetBufferUserInterrupt<br>TCsetBufferUserInterruptAIO<br>TCsetBufferUserInterrupt2<br>TCsetNCBufferUserInterrupt<br>TCsetNCBufferUserInterruptAIO<br>TCsetNCBufferUserInterrupt2 |
| See Also | TCenableUserInterrupt<br>setIntMask<br>getIntMask<br>TCdisableInterruptChip |

### 6.4.17 Analogue I/O Resource Management

#### 6.4.17.1 Test if ADC Interrupt Source is Free — AIOADCisAvailable

Called to determine whether an ADC chip is available at a particular base address offset for use as an interrupt source. For some cards, the base address offset is just a placeholder for the ADC chip.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOADCisAvailable (h, Chip)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Chip* | **short**: determines ADC interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). For some ADC cards this is an actual base address offset. For others it depends on the bit position in the card's interrupt enable register. The following pre-defined constants may be used:<br><br>ADC0 = 0<br>ADC2 = 8<br><br>For PC26AT, PC27E and PC30AT use ADC0. For PCI230 and PCI260 use ADC2. |
| Returns **short**: | 0 = ADC chip interrupt source NOT available, 1 = Available; | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | AIOcountADCgroups<br>AIOADCgroupIntChip<br>AIOcountADCchans | |

#### 6.4.17.2 Determine Number of ADC Channel Groups — AIOcountADCgroups

Called to determine the number of ADC channel groups supported on a card. Generally, ADC channels that share the same multiplexer will be in the same channel group. If a card has

ADC channel groups they are numbered from 0 to number of groups–1.

N.B. all currently supported cards have at most one ADC channel group.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOcountADCgroups (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | number of ADC channel groups on the card. | |
| Prior Calls | registerBoardEx | |
| See Also | AIOcountADCchans<br>AIOADCgroupIntChip<br>AIOADCgroupHasFIFO | |

#### 6.4.17.3 Determine Number of ADC Channels in a Group — AIOcountADCchans

Called to determine the number of ADC channels in a particular channel group on a card. If the specified group exists, channels are numbered from 0 to number of channels–1.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOcountADCchans (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| Returns **short**: | number of ADC channels in the group; this will be 0 if *Group* is out of range for the card. | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | | |

#### 6.4.17.4 Determine ADC Channel Group's Interrupt Source — AIOADCgroupIntChip

Called to determine the *Chip* value to use when using an ADC channel group's 'conversion complete' interrupt as the interrupt source when calling one of the user interrupt set-up functions. This value may also be used for the AIOADCisAvailable function.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOADCgroupIntChip (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| Returns **short**: | the *Chip* value to use for user interrupt set-up (>= 0); | |

| | |
|---|---|
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups |
| See Also | AIOADCisAvailable |

#### 6.4.17.5 Determine whether ADC Channel Group has a FIFO — AIOADCgroupHasFIFO

Called to determine whether an ADC channel group has a FIFO.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIOADCgroupHasFIFO (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| Returns **short**: | 0 = NO FIFO, 1 = has FIFO; | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOgetADCgroupFIFOsize | |

#### 6.4.17.6 Determine whether ADC Channel Group has a FIFO and Get its Size — AIOgetADCgroupFIFOsize

Called to determine whether an ADC channel group has a FIFO and to determine the size of the FIFO.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOgetADCgroupFIFOsize (h, Group, pSize)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pSize* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to the size of the FIFO. |
| Returns **short**: | 0 = NO FIFO, 1 = has FIFO | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |

| | |
|---|---|
| See Also | AIOADCgroupHasFIFO |

#### 6.4.17.7 Test if DAC Interrupt Source is Free — AIODACisAvailable

Called to determine whether a DAC chip is available at a particular base address offset for use as an interrupt source. For some cards, the base address offset is just a placeholder for the DAC chip. Generally, only cards with DAC FIFOs have an interrupt source.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIODACisAvailable (h, Chip)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Chip* | **short**: determines DAC interrupt source. The interrupt source number from 0 to 5 multiplied by 4 (0, 4, 8, 12, 16, 20). It depends on the bit position in the card's interrupt enable register. The following pre-defined constants may be used:<br><br>DAC2 = 8<br>DAC4 = 16<br><br>For PCI224 and PCI234 use DAC2.<br><br>For PCI230+ hardware version 2 use DAC4. |
| Returns **short**: | 0 = DAC chip interrupt source NOT available, 1 = Available; | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | AIOcountDACgroups<br>AIODACgroupIntChip<br>AIOcountDACchans | |

#### 6.4.17.8 Determine Number of DAC Channel Groups — AIOcountDACgroups

Called to determine the number of DAC channel groups supported on a card. Generally, non-multiplexed DAC channels will be in the same channel group and DAC channels that share the same multiplexer will be in the same channel group. If a card has DAC channel groups they are numbered from 0 to number of groups–1.

N.B. all currently supported cards have at most one DAC channel group.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOcountDACgroups (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |

| | |
|---|---|
| Returns **short**: | number of DAC channel groups on the card. |
| Prior Calls | registerBoardEx |
| See Also | AIOcountDACchans<br>AIODACgroupIntChip<br>AIODACgroupHasFIFO |

#### 6.4.17.9 Determine Number of DAC Channels in a Group — AIOcountDACchans

Called to determine the number of DAC channels in a particular channel group on a card. If the specified group exists, channels are numbered from 0 to number of channels–1.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOcountDACchans (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| Returns **short**: | number of DAC channels in the group; this will be 0 if *Group* is out of range for the card. | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | | |

#### 6.4.17.10 Determine DAC Channel Group's Interrupt Source — AIODACgroupIntChip

Called to determine the *Chip* value to use when using a DAC channel group's interrupt as the interrupt source when calling one of the user interrupt set-up functions. This value may also be used for the AIODACisAvailable function.

N.B. Only DAC channel groups with a FIFO may be used as an interrupt source. For FIFO-less DAC channel groups an error (ERRCHAN) is returned.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIODACgroupIntChip (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| Returns **short**: | The *Chip* value to use for user interrupt set-up (>= 0) if the DAC channel group can be used as an interrupt source; | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |

| See Also | AIODACisAvailable |
|---|---|

#### 6.4.17.11 Determine whether DAC Channel Group has a FIFO — AIODACgroupHasFIFO

Called to determine whether a DAC channel group has a FIFO.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIODACgroupHasFIFO (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| Returns **short**: | 0 = NO FIFO, 1 = has FIFO; | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOgetDACgroupFIFOsize | |

#### 6.4.17.12 Determine whether DAC Channel Group has a FIFO and Get its Size — AIOgetDACgroupFIFOsize

Called to determine whether a DAC channel group has a FIFO and to determine the size of the FIFO.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOgetDACgroupFIFOsize (h, Group, pSize)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *pSize* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to the size of the FIFO. |
| Returns **short**: | 0 = NO FIFO, 1 = has FIFO; | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIODACgroupHasFIFO | |

## 6.4.18 Analogue I/O Configuration

### 6.4.18.1 Query ADC Software Bipolar/Unipolar Settings — AIOgetADCchanMode

Gets the software bipolar/unipolar mode of each ADC channel in a group. This indicates the way raw data from each ADC channel is cooked, but does not reflect the actual settings in the hardware.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOgetADCchanMode (h, Group, pModes)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pModes* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to bit vector value with 1's for unipolar channels and 0's for bipolar or unsupported channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOsetADCchanMode<br>AIOgetHWADCchanMode | |

### 6.4.18.2 Query ADC Hardware Bipolar/Unipolar Settings — AIOgetHWADCchanMode

Gets the hardware bipolar/unipolar mode of each ADC channel in a group. This is only supported on certain cards. It does not indicate the way raw data from each ADC channel is cooked.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIOgetHWADCchanMode (h, Group, pModes)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pModes* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to bit vector value with 1's for unipolar channels and 0's for bipolar or unsupported channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE | |

| | |
|---|---|
| | ERRCHAN<br>ERRDATA |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups |
| See Also | AIOgetADCchanMode<br>AIOsetHWADCchanMode |

#### 6.4.18.3 Configure ADC Software Bipolar/Unipolar Settings — AIOsetADCchanMode

Sets the software bipolar/unipolar mode of each ADC channel in a group. This affects the way raw data from each ADC channel is cooked, but has no effect on the underlying hardware settings.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOsetADCchanMode (h, Group, ChMask, Modes)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to change and 0's for channels to leave alone. |
| | *Modes* | **unsigned long**: bit vector with 1's for unipolar channels and 0's for bipolar channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOgetADCchanMode<br>AIOgetHWADCchanMode<br>AIOsetHWADCchanMode<br>AIOsetAllADCchanMode | |

#### 6.4.18.4 Configure ADC Hardware Bipolar/Unipolar Settings — AIOsetHWADCchanMode

Sets the hardware bipolar/unipolar mode for ADC channels in a group. This is only supported on certain cards. It has no affect on the way raw data from each ADC channel is cooked.

SUPPORTED IN VERSION 4.10 ONWARDS

```
i = AIOsetHWADCchanMode (h, Group, ModeVal)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |

| | | |
|---|---|---|
| | *ModeVal* | **unsigned long**: bipolar/unipolar mode value may depend on card type.<br><br>In general: 0 = set all channels to bipolar in hardware; $FFFFFFFF_{16}$ = set all channels to unipolar in hardware.<br><br>For PCI230 and PCI260: all non-zero values set all channels to unipolar in hardware. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOsetADCchanMode<br>AIOgetHWADCchanMode<br>AIOsetAllADCchanMode | |

#### 6.4.18.5 Configure ADC All Channels Bipolar or Unipolar — AIOsetAllADCchanMode

Sets the software bipolar/unipolar mode of all ADC channels in a group to all bipolar or all unipolar. This affects the way raw data from each ADC channel is cooked. Also sets the hardware bipolar/unipolar mode on cards that support this.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = AIOsetAllADCchanMode (h, Group, Mode)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *Mode* | **short**: Zero (0) sets all channels to bipolar; Non-Zero set all channels to unipolar. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOsetADCchanMode<br>AIOsetHWADCchanMode | |

#### 6.4.18.6 Query ADC Hardware Single-ended/Differential Settings — AIOgetHWADCchanDiff

Gets the hardware single-ended/differential mode of each ADC channel in a group. This is only supported on certain cards.

SUPPORTED IN VERSION 4.10 ONWARDS.

**`i = AIOgetHWADCchanDiff (h, Group, pDiffs)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pDiffs* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to bit vector value set with 1's for differential channels and 0's for single-ended channels or unsupported channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOsetHWADCchanDiff | |

#### 6.4.18.7 Configure ADC Hardware Single-ended/Differential Settings — AIOsetHWADCchanDiff

Sets the hardware single-ended/differential mode for ADC channels in a group. This is only supported on certain cards.

SUPPORTED IN VERSION 4.10 ONWARDS.

**`i = AIOsetHWADCchanDiff (h, Group, DiffVal)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *DiffVal* | **unsigned long**: single-ended/differential mode value may depend on card type.<br><br>In general: 0 = set all channels to single-ended; $FFFFFFFF_{16}$ = set all channels to differential.<br><br>For PCI230 and PCI260: all non-zero values set all channels to differential. Single-ended channels are paired up in differential mode so only even channels (0, 2, etc.) are valid. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |

| See Also | AIOgetHWADCchanDiff |
|---|---|

#### 6.4.18.8 Query ADC Hardware Gain Settings — AIOgetHWADCchanGain

Gets the hardware gain settings for a group of ADC channels. This is only supported on certain cards.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIOgetHWADCchanGain (h, Group, pGains)
```

where

*h* — **short**: board handle as issued by the registerBoardEx function.

*Group* — **short**: ADC channel group.

*pGains* — **pointer to unsigned long**: pointer to unsigned long variable which will be set to channel gains value which depends on the card type.

For PCI230 and PCI260:

bits 1 – 0: gain for channels 0 and 1
bits 3 – 2: gain for channels 2 and 3
bits 5 – 4: gain for channels 4 and 5
bits 7 – 6: gain for channels 6 and 7
bits 9 – 8: gain for channels 8 and 9
bits 11 –10: gain for channels 10 and 11
bits 13 – 12: gain for channels 12 and 13
bits 15 – 14: gain for channels 15 and 14

| Value | Unipolar range | Bipolar range |
|---|---|---|
| 0 | 0 – 20V* | ±10V |
| 1 | 0 – 10V | ±5V |
| 2 | 0 – 5V | ±2.5V |
| 3 | 0 – 2.5V | ±1.25V |

E. . Saturates at about 12V.

Returns **short**: OK

or ERRHANDLE
ERRCHAN

Prior Calls: registerBoardEx
AIOcountADCgroups

See Also: AIOsetHWADCchanGain

#### 6.4.18.9 Configure ADC Hardware Gain Settings — AIOsetHWADCchanGain

Sets the hardware gain for ADC channels in a group. This only has an affect on certain cards.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIOsetHWADCchanGain (h, Group, ChMask, Gains)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to change and 0's for channels to leave alone. |
| | *Gains* | **unsigned long**: channel gains value which depends on the card type.<br><br>For PCI230 and PCI260:<br><br>bits 1 – 0: gain for channels 0 and 1<br>bits 3 – 2: gain for channels 2 and 3<br>bits 5 – 4: gain for channels 4 and 5<br>bits 7 – 6: gain for channels 6 and 7<br>bits 9 – 8: gain for channels 8 and 9<br>bits 11 –10: gain for channels 10 and 11<br>bits 13 – 12: gain for channels 12 and 13<br>bits 15 – 14: gain for channels 15 and 14<br><br>Value Unipolar range Bipolar range<br>0 0 – 20V* ±10V<br>1 0 – 10V ±5V<br>2 0 – 5V ±2.5V<br>3 0 – 2.5V ±1.25V<br><br>E. . Saturates at about 12v.<br><br>For any bit in *ChMask* set to '1', both channels in the pair will be affected by the corresponding pair of bits in Gains. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOgetHWADCchanGain | |

#### 6.4.18.10 Query DAC Software Bipolar/Unipolar Settings — AIOgetDACchanMode

Gets the software bipolar/unipolar mode of each DAC channel in a group. This indicates the way cooked data is converted to raw data for each DAC channel, but does not reflect the actual settings in the hardware.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOgetDACchanMode (h, Group, pModes)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |

| | | |
|---|---|---|
| | *pModes* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to bit vector value with 1's for unipolar channels and 0's for bipolar or unsupported channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOsetDACchanMode<br>AIOgetHWDACchanMode | |

#### 6.4.18.11 Query DAC Hardware Bipolar/Unipolar Settings — AIOgetHWDACchanMode

Gets the hardware bipolar/unipolar mode of each DAC channel in a group. This is only supported on certain cards. It does not indicate the way cooked data is uncooked for each DAC channel.

SUPPORTED IN VERSION 4.10 ONWARDS.

```
i = AIOgetHWDACchanMode (h, Group, pModes)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *pModes* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to bit vector value with 1's for unipolar channels and 0's for bipolar or unsupported channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOgetDACchanMode<br>AIOsetHWDACchanMode | |

#### 6.4.18.12 Configure DAC Software Bipolar/Unipolar Settings — AIOsetDACchanMode

Sets the software bipolar/unipolar mode of each DAC channel in a group. This affects the way cooked data is converted to raw data for each DAC channel, but has no effect on the underlying hardware settings.

SUPPORTED IN VERSION 4.00 ONWARDS.

**`i = AIOsetDACchanMode (h, Group, ChMask, Modes)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to change and 0's for channels to leave alone. |
| | *Modes* | **unsigned long**: bit vector with 1's for unipolar channels and 0's for bipolar channels. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOgetDACchanMode<br>AIOgetHWDACchanMode<br>AIOsetHWDACchanMode<br>AIOsetAllDACchanMode | |

#### 6.4.18.13 Configure DAC Hardware Bipolar/Unipolar Settings — AIOsetHWDACchanMode

Sets the hardware bipolar/unipolar mode for DAC channels in a group. This only has an affect on certain cards. It has no affect on the way cooked data is uncooked for each DAC channel.

SUPPORTED IN VERSION 4.10 ONWARDS.

**`i = AIOsetHWDACchanMode (h, Group, ModeVal)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *ModeVal* | **unsigned long**: bipolar/unipolar mode value may depend on card type.<br><br>In general: 0 = set all channels to bipolar in hardware; $FFFFFFFF_{16}$ = set all channels to unipolar in hardware.<br><br>For PCI230: all non-zero values set all channels to unipolar in hardware.<br><br>For PCI224: all non-zero values set all channels to unipolar in hardware.<br><br>For PCI234: only value 0 is allowed, as there is no hardware unipolar mode. |
| Returns **short**: | OK | |

| | |
|---|---|
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups |
| See Also | AIOsetDACchanMode<br>AIOgetHWDACchanMode<br>AIOsetAllDACchanMode |

#### 6.4.18.14 Configure DAC All Channels Bipolar or Unipolar — AIOsetAllDACchanMode

Sets the software bipolar/unipolar mode of all DAC channels in a group to all bipolar or all unipolar. This affects the way cooked data is converted to raw data for each DAC channel. Also sets the hardware bipolar/unipolar mode on cards that support this.

SUPPORTED IN VERSION 4.02 ONWARDS.

```
i = AIOsetAllDACchanMode (h, Group, Mode)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *Mode* | **short**: Zero (0) sets all channels to bipolar; Non-Zero set all channels to unipolar. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOsetDACchanMode<br>AIOsetHWDACchanMode | |

#### 6.4.18.15 Query DAC Hardware Output Range Settings — AIOgetHWDACchanRange

Gets the hardware output range settings for a group of DAC channels. This is only supported on certain cards.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOgetHWDACchanRange (h, Group, pRanges)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *pRanges* | **pointer to unsigned long**: pointer to unsigned long variable which will be set to channel output ranges value which depends on the |

| | | |
|---|---|---|
| | | card type.<br><br>For PCI224: Value applies to all channels.<br><br>Value Unipolar range Bipolar range<br>0 0 – 1.25V ±1.25V<br>1 0 – 2.5V ±2.5V<br>2 0 – 5V ±5V<br>3 0 – 10V ±10V<br><br>For PCI234: Value applies to all channels. 0 = use range selected by jumpers. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOsetHWDACchanRange | |

#### 6.4.18.16 Configure DAC Hardware Output Range Settings — AIOsetHWDACchanRange

Sets the hardware output ranges for DAC channels in a group. This only has an affect on certain cards.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOsetHWDACchanRange (h, Group, ChMask, Ranges)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to change and 0's for channels to leave alone. |
| | *Ranges* | **unsigned long**: channel output ranges value which depends on the card type.<br><br>For PCI224: Value applies to all channels. *ChMask* is ignored.<br><br>Value Unipolar range Bipolar range<br>0 0 – 1.25V ±1.25V<br>1 0 – 2.5V ±2.5V<br>2 0 – 5V ±5V<br>3 0 – 10V ±10V<br><br>For PCI234: Value applies to all channels. *ChMask* is ignored. 0 = use range selected by jumpers. |
| Returns **short**: | OK | |

| | |
|---|---|
| or | ERRHANDLE<br>ERRCHAN |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups |
| See Also | AIOgetHWDACchanRange |

### 6.4.19 Analogue Input

#### 6.4.19.1 Set ADC Conversion Trigger Source — AIOsetADCconvSource

Sets the ADC conversion trigger source for an ADC channel group. Depending on the card, certain settings may require jumpers to be set, be unsupported or ignored.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOsetADCconvSource (h, Group, Cnv)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *Cnv* | **short**: conversion trigger source:<br><br>CNV_NONE = 0 No trigger<br>CNV_SW = 1 Software triggered<br>CNV_EXT_P = 2 External +ve edge<br>CNV_EXT_N = 3 External –ve edge<br>CNV_CT0 = 4 Timer channel 0 OUT<br>CNV_CT1 = 5 Timer channel 1 OUT<br>CNV_CT2 = 6 Timer channel 2 OUT |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups | |
| See Also | AIOsetADCmultiplexer<br>AIOstartADCconversion<br>AIOgetADCdata | |

#### 6.4.19.2 Set ADC Current Channel in Multiplexer — AIOsetADCmultiplexer

Sets the ADC multiplexer of a specified channel group to select a specified channel.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOsetADCmultiplexer (h, Group, Chan)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the |

| | | |
|---|---|---|
| | | registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *Chan* | **short**: selected channel. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups<br>AIOcountADCchans | |
| See Also | AIOsetADCconvSource<br>AIOstartADCconversion<br>AIOgetADCdata | |

#### 6.4.19.3 Software-trigger ADC Conversion — AIOstartADCconversion

Starts software-triggered A-to-D conversion for the currently multiplexed channel on a specified channel group.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOstartADCconversion (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups<br>AIOsetADCconvSource<br>AIOsetADCmultiplexer | |
| See Also | AIOgetADCdata | |

#### 6.4.19.4 Read ADC Data — AIOgetADCdata

Reads data from an ADC channel group's ADC port and cooks it. The data read is assumed to be from the channel currently multiplexed at the time of the previous software-triggered conversion or assumed time of the previous external-triggered or timer-triggered conversion. (This will be inaccurate if no conversion has been triggered since the last time the multiplexer was changed.) The user can alter the multiplexer in the period between starting a software-triggered interrupt and reading the value back, if desired.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOgetADCdata (h, Group, pData)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pData* | **pointer to long**: pointer to long integer variable which will be set to the cooked ADC data value. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups<br>AIOsetADCconvSource<br>AIOsetADCmultiplexer<br>AIOstartADCconversion | |
| See Also | | |

#### 6.4.19.5 Set ADC Start Acquisition Trigger — AIOsetADCstartAcquisitionTrigger

Sets ADC start acquisition trigger if supported by the driver and the hardware.

A continuous acquisition starts when the start acquisition trigger occurs after the driver is given the first buffer to be filled in by DIO_TC.DLL. Subsequent buffers are filled as normal.

For a non-continuous acquisition, the driver applies the start acquisition trigger to every buffer given to it by DIO_TC.DLL.

If the start acquisition type is START_TRIG, the specified hold-off count must be zero and data acquired before the trigger occurs is discarded.

If the start acquisition type is START_NOW, a non-zero hold-off count may be specified to delay setting the trigger until the number of samples specified by the hold-off count have been acquired. Data acquired before the trigger occurs is not discarded. The function AIOgetADCpretriggerCount may be called to check whether the trigger has occurred and the position of the sample where the trigger occurred.

Start acquisition types other than START_NOW and start acquisition trigger sources other than TRIG_NOW are only supported by the new PCI230+ and PCI260+ cards.

If the start acquisition trigger uses an analogue channel, this should be one of the channels being acquired.

For the PCI230+ and PCI260+ cards, start acquisition types other than START_NOW and start acquisition trigger sources other than TRIG_NOW are only used when the ADC FIFO is enabled.

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = AIOsetADCstartAcquisitionTrigger (h, Group, Trig, Start, Chan,
Threshold, Hysteresis, HoldOff)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the |

| | | |
|---|---|---|
| | | registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *Trig* | **short**: start acquisition trigger source:<br><br>TRIG_NOW = 0<br>Trigger immediately<br>TRIG_NEVER = 1<br>Never trigger<br>TRIG_EXT_LTOH = 2<br>Trigger on external digital low-high transition<br>TRIG_EXT_HTOL = 3<br>Trigger on external digital high-low transition<br>TRIG_EXT_LOW = 4<br>Trigger when external digital signal is low<br>TRIG_EXT_HIGH = 5<br>Trigger when external digital signal is high<br>TRIG_ANA_LTOH = 6<br>Trigger on analogue low-high transition<br>TRIG_ANA_HTOL = 7<br>Trigger on analogue high-low transition<br>TRIG_ANA_LOW = 8<br>Trigger when analogue value is low<br>TRIG_ANA_HIGH = 9<br>Trigger when analogue value is high |
| | *Start* | **short**: start acquisition start type:<br><br>START_NOW = 0<br>Start as soon as possible (pre-trigger mode)<br>START_TRIG = 1<br>Start when trigger occurs (no pre-trigger) |
| | *Chan* | **short**: analogue channel for start acquisition trigger. |
| | *Threshold* | **long**: analogue threshold for start acquisition trigger. |
| | *Hysteresis* | **long**: analogue hysteresis for low-high and high-low transitions. |
| | *HoldOff* | **unsigned long**: number of samples to acquire before applying the trigger when *Start* is START_NOW. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRRANGE<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx | |

| | |
|---|---|
| | AIOcountADCgroups |
| See Also | AIOgetADCpretriggerCount<br>AIOsetADCconvSource |

#### 6.4.19.6 Get ADC Pre-trigger Count — AIOgetADCpretriggerCount

Gets the pre-trigger count for the ADC start acquisition trigger when the start type has been set to START_NOW. This gives the number of samples that were acquired before the trigger occurred, or the number of samples that have been acquired so far if the trigger has not occurred yet. The function return value indicates whether the trigger has occurred.

If the acquisition trigger start type was set to START_TRIG, the pre-trigger count will be 0 (except in non-continuous mode when the buffer length is not an integer multiple of the number of enabled channels). The function return value still indicates whether the trigger has occurred.

Note that in non-continuous mode, the start acquisition trigger is applied at the start of each buffer, so the return value is only valid at certain points in time. In non-continuous mode it is best to call this function from the callback function (if using callbacks) or following a call to the TCdriveNCBufferUserInterrupt function (if not using callbacks).

SUPPORTED IN VERSION 4.42 ONWARDS.

```
i = AIOgetADCpretriggerCount (h, Group, pCount)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: ADC channel group. |
| | *pCount* | **pointer to unsigned long**: pointer to a variable to be set to the pre-trigger count value. |
| Returns **short**: | FALSE (= 0) if the start acquisition trigger has not occurred yet.<br>TRUE (= 1) if the start acquisition trigger has occurred. | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA<br>ERRSUPPORT | |
| Prior Calls | registerBoardEx<br>AIOcountADCgroups<br>AIOsetADCstartAcquisitionTrigger<br>TCsetBufferUserInterruptAIO<br>TCsetNCBufferUserInterruptAIO<br>enableInterrupts | |
| See Also | | |

## 6.4.20 Analogue Output

### 6.4.20.1 Write DAC Data — AIOsetDACchanData

Writes a set of data to a corresponding to a set of DAC channels, all belonging to a specified channel group.

SUPPORTED IN VERSION 4.00 ONWARDS.

```
i = AIOsetDACchanData (h, Group, ChMask, pData)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to write and 0's for channels to leave alone. |
| | *pData* | **pointer to unsigned long**: pointer to first element of array of data to be written to the corresponding DAC channels. Successive elements of the array correspond to successive '1' bits in *ChMask*, from the least significant '1' bit to the most significant '1' bit. Only the least significant 16 bits of each array element are used; this 16-bit value is treated as an unsigned integer for unipolar channels or as a 2's complement signed integer for bipolar channels.<br><br>E.g. if *ChMask* is set to 10 ($1100_2$), the first element of the array contains data for DAC channel 2 and the second element of the array contains data for DAC channel 3.<br><br>The contents of the array are not modified by the function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups<br>AIOcountDACchans | |
| See Also | | |

### 6.4.20.2 Set DAC Conversion Trigger Source — AIOsetDACconvSource

Sets the DAC conversion trigger source for a DAC channel group. This only applies while the DAC is being operated in certain ways. It applies when an interrupt has been set up to drive the DAC in FIFO mode. It also applies when a waveform has been loaded into the DAC hardware's waveform playback buffer. In both cases, it specifies the clock source for

triggering the DAC with the next data, and only applies to card types that have specific support for these operations. Depending on the card, certain settings may be unsupported or ignored.

For PCI224 and PCI234 and PCi230+ hardware version 2, CNV_NONE selects no trigger, CNV_SW selects the software trigger, CNV_EXT_P selects the external positive-going edge trigger, CNV_EXT_N selects the external negative-going edge trigger, CNV_CT0 through CNV_CT2 respectively select Z2 timer channel 0, 1 or 2 trigger.

SUPPORTED IN VERSION 4.20 ONWARDS.

**`i = AIOsetDACconvSource (h, Group, Cnv)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *Cnv* | **short**: conversion trigger source:<br>CNV_NONE = 0 No trigger<br>CNV_SW = 1 Software triggered<br>CNV_EXT_P = 2 External +ve edge<br>CNV_EXT_N = 3 External –ve edge<br>CNV_CT0 = 4 Timer channel 0 OUT<br>CNV_CT1 = 5 Timer channel 1 OUT<br>CNV_CT2 = 6 Timer channel 2 OUT |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups | |
| See Also | AIOstartDACconversion<br>AIOsetDACchanWaveform | |

#### 6.4.20.3 Set DAC Waveform Data — AIOsetDACchanWaveform

Sets up a multi-channel waveform in a DAC channel group's hardware playback buffer. This is only supported on certain cards.

The DAC group's conversion trigger source, that is set using *AIOsetDACconvSource*, is used to clock data for conversion on each tick. On each tick, the next frame of data (one datum for each enabled channel) is converted by the DACs. After the last frame of data, it wraps around to the first frame on the next tick.

The waveform can be cancelled by setting a zero-length waveform, replacing with another waveform, or writing data to the DAC channel group using *AIOsetDACchanData*.

It is supported on PCI224 and PCI234 and PCI230+ hardware version 2. On supported cards, the hardware playback buffer is the FIFO and its size can be determined by calling AIOgetDACgroupFIFOsize. For PCI224 and PCI234 the maximum supported length is 4096. For PCI230+ hardware version 2 the maximum supported length is 1024.

SUPPORTED IN VERSION 4.20 ONWARDS.

**`i = AIOsetDACchanWaveform (h, Group, ChMask, pData, DLen)`**

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| | *ChMask* | **unsigned long**: bit vector with 1's for channels to write and 0's for channels to leave alone. |
| | *pData* | **pointer to unsigned long**: pointer to first element of array of data to be written to the DAC channels enabled by setting bits to '1' in *ChMask*. This may be a two-dimensional array or a suitably formatted one dimensional array. This consists of *m* frames of *n* data values, where *m* is a whole number greater than or equal to 0, and *n* is the number of enabled channels. The *m* frames are indexed by the outer dimension and the *n* data values within each frame are indexed by the inner dimension.<br><br>Successive elements of the array within each frame correspond to successive '1' bits in *ChMask*, from the least significant '1' bit to the most significant '1' bit (exactly *n* bits are set to '1'). Only the least significant 16 bits of each array element are used; this 16-bit value is treated as an unsigned integer for unipolar channels or as a 2's complement signed integer for bipolar channels.<br><br>E.g. if *ChMask* is set to 10 ($1100_2$), *n* is 2; the first element of each frame contains data for DAC channel 2 and the second element of each frame contains data for DAC channel 3.<br><br>The product *m* x *n* must not be larger than the hardware playback buffer.<br><br>The contents of the array are not modified by the function. |
| | *DLen* | **unsigned long**: Total length of data, i.e. *m* x *n*, the product of the number of frames multiplied by the number of enabled channels. This may be zero. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups<br>AIOcountDACchans | |

| | |
|---|---|
| See Also | AIOsetDACconvSource<br>AIOstartDACconversion |

#### 6.4.20.4 Software-trigger DAC Conversion — AIOstartDACconversion

Provides a tick of the software clock for a DAC channel group. This is only effective when the DAC group's conversion trigger source has been set to CNV_SW and the DAC is being operated in certain modes. It applies when an interrupt has been set up to drive the DAC in FIFO mode. It also applies when a waveform has been loaded into the DAC hardware's waveform playback buffer. This only applies to card types that have specific support for these operations.

SUPPORTED IN VERSION 4.20 ONWARDS.

```
i = AIOstartDACconversion (h, Group)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Group* | **short**: DAC channel group. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>AIOcountDACgroups<br>AIOsetDACconvSource | |
| See Also | AIOsetDACchanWaveform | |

### 6.4.21 Support for HP VEE

#### 6.4.21.1 Timer Counter Functions In HP VEE.

As HP Vee does not support passing of float types, there are versions of some of the TC functions that use parameters of type double instead of float. These just call the normal float versions internally. These functions are:

- TCVgetRatio (calls TCgetRatio)
- TCVsetMonoShot (calls TCsetMonoShot)
- TCVsetAstable (calls TCsetAstable)
- TCVgetExtPeriod (calls TCgetExtPeriod)
- TCVgetExtFreq (calls TCgetExtFreq)
- TCVgenerateFreq (calls TCgenerateFreq)
- TCVgeneratePulse (calls TCgeneratePulse)
- TCVgenerateAccFreq (calls TCgenerateAccFreq)
- TCVmultiplyFreq (calls TCmultiplyFreq)
- TCVdivideFreq (calls TCdivideFreq)
- TCVsetDCO (calls TCsetDCO)

### 6.4.22 Legacy Analogue I/O Functions

#### 6.4.22.1 Set PC27 Multiplexer Register — PC27SetMultiplexer

Outputs a byte to the PC27 multiplexer register.

SUPPORTED IN VERSION 2.01 ONWARDS. RETAINED FOR BACKWARD COMPATIBILITY.

```
i = PC27SetMultiplexer (h, Data)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Data* | **short**: data to write. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | PC27StartConversion<br>PC27getData | |

#### 6.4.22.2 Start PC27 ADC Conversion — PC27StartConversion

Writes to the PC27 start conversion register.

SUPPORTED IN VERSION 2.01 ONWARDS. RETAINED FOR BACKWARD COMPATIBILITY.

```
i = PC27StartConversion (h)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx<br>PC27SetMultiplexer | |
| See Also | PC27getData | |

#### 6.4.22.3 Read PC27 ADC Data — PC27getData

Reads a sample from the PC27 ADC port.

SUPPORTED IN VERSION 2.0 ONWARDS. RETAINED FOR BACKWARD COMPATIBILITY.

```
i = PC27getData (h, pData)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the |

| | | |
|---|---|---|
| | | registerBoardEx function. |
| | *pData* | **pointer to short**: pointer to short integer variable which will be set to the raw data value from the PC27 ADC port. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN<br>ERRDATA | |
| Prior Calls | registerBoardEx<br>PC27SetMultiplexer<br>PC27StartConversion | |
| See Also | | |

#### 6.4.22.4 Write PC27 DAC Data — PC24setData

Outputs a sample to a PC24 DAC channel.

SUPPORTED IN VERSION 2.0 ONWARDS. RETAINED FOR BACKWARD COMPATIBILITY.

```
i = PC24setData (h, Chan, Data)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Chan* | **short**: DAC channel number (0, 1, 2, 3). |
| | *Data* | **short**: raw data value to write to DAC channel. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | | |

### 6.4.23 Driver Interface Functions

#### 6.4.23.1 Send IOCTL Instruction — DIO_TC_IOCTL

Pass an IOCTL instruction directly to the driver. These are documented in ADIOCTL.RTF. Examples of use are the DIO_TC.DLL source code itself.

```
i = DIO_TC_IOCTL (h, Code, pIOBuffer, SizeOfBuffer)
```

| | | |
|---|---|---|
| where | *h* | **short**: board handle as issued by the registerBoardEx function. |
| | *Code* | **int**: IOCTL code. See ADIOCTL.H and ADIOCTL.RTF |

| | | |
|---|---|---|
| | *pIOBuffer* | **pointer to unsigned long**: pointer to IOCTL data packet cast as pointer to unsigned long. Content depends on IOCTL code used. (See ADIOCTL.RTF for more information.) It may be one of the following types:<br><br>ULONG<br>TDIO_PORTIO<br>TDIO_2PORTIO<br>TDIO_BLKPORTIO<br>TDIO_IRQSETUP<br>Array of ULONG |
| | *SizeOfBuffer* | **unsigned long**: size of IOCTL buffer pointed to by *pIOBuffer* in bytes. |
| Returns **short**: | OK | |
| or | ERRHANDLE<br>ERRCHAN | |
| Prior Calls | registerBoardEx | |
| See Also | | |

## 6.5 Library Error Codes

| Mnemonic | Returned Value | Meaning |
|---|---|---|
| OK | 0 | Operation successful. |
| ERRSUPPORT | –1 | Operation not supported by board, or the maximum boards/buffers are already registered. |
| ERRBASE | –2 | Base address is invalid or in use. |
| ERRIRQ | –3 | Interrupt level is invalid or in use. |
| ERRHANDLE | –4 | Invalid board handle, or board not registered. |
| ERRCHAN | –5 | Invalid channel number |
| ERRDATA | –6 | Invalid data |
| ERRRANGE | –7 | Out of range |
| ERRMEMORY | –8 | Insufficient Memory |
| ERRBUFFER | –9 | Invalid buffer handle – not allocated |
| ERRPC226 | –10 | PC226 board not found (for VCO function) |

# 7 IOCTL INTERFACE

## 7.1 About this Chapter

This chapter gives a brief outline of the low-level interface to the Amplicon DIO driver. See the AMPIOCTL.RTF file (installed in the DIO_CODE subdirectory) for a full description. See the C source code for DIO_TC.DLL for examples of usage.

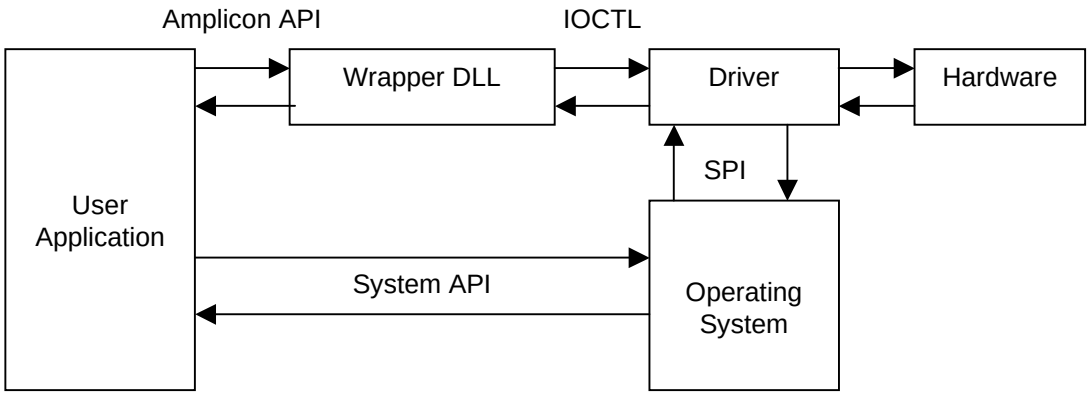
## 7.2 About the Driver

The Amplicon Windows DIO Driver is a kernel mode driver providing and IO control (IOCTL) interface to the hardware. Different drivers are provided for different versions of Windows. A VxD is provided for Windows 95, Windows 98 and Windows ME. A legacy kernel mode driver is provided for Windows NT 4.0. A plug-and-play kernel mode driver is provided for Windows 2000 upwards. All the drivers provide more or less the same ioctl interface, but there are minor differences between the VxD and the drivers for NT-based operating systems.

### 7.2.1 Driver Architecture

Both the Windows 95 and NT device drivers support an IOCTL interface allowing a wrapper DLL or a user application to communicate directly to the hardware. An example wrapper DLL, DIO_TC.DLL has been provided.

**Architecture**




## 7.3 The IOCTL Commands Supported

The IOCTL interface is defined in the C header file AMPIOCTL.H. These IOCTL commands can be used in conjunction with the DIO_IOCTL call function to directly interface to the driver.

| | |
|---|---|
| IOCTL_QUERY_VERSION | Determine driver version. (Version 4.02 onwards.) |
| IOCTL_QUERY_RESOURCE | Determine card type, IRQ and I/O base address. |
| IOCTL_QUERY_PCIPOS | Determine card position on PCI bus. (Version 4.20 onwards.) |
| IOCTL_QUERY_HWVERSION | Determine card's hardware version. (Version |

| | |
|---|---|
| | 4.42 onwards.) |
| IOCTL_QUERY_REALHWVERSION | Determine card's real hardware version. (Version 5.02 onwards.) |
| IOCTL_GET_DEVICE | Windows 95 only — allow support for more than one card. |
| IOCTL_GET_NEXT_DEVICE | Windows 95 only — allow support for more than one card. |
| IOCTL_GET_NTH_DEVICE | Windows 95 only — allow support for more than one card. (Version 4.20 onwards.) |
| IOCTL_SET_CTDATA | Write byte data to counter timer data port. |
| IOCTL_GET_CTDATA | Read byte data from counter timer data port. |
| IOCTL_SET_CTCONTROL | Write byte data to counter timer control port. |
| IOCTL_SET_CTCLK | Set counter timer clock source. |
| IOCTL_SET_CTGATE | Set counter timer gate source. |
| IOCTL_SET_CTDATA16 | Write 16-bit data to counter timer data port. (Version 4.02 onwards.) |
| IOCTL_GET_CTDATA16 | Read 16-bit data from counter timer data port. (Version 4.02 onwards.) |
| IOCTL_SET_CTDATA32 | Write two 16-bit values to two counter timer data ports. (Version 4.02 onwards.) |
| IOCTL_GET_CTDATA32 | Read two 16-bit values from two counter timer data ports. (Version 4.02 onwards) |
| IOCTL_SET_PPIDATA | Write byte data to PPI digital I/O data port. |
| IOCTL_GET_PPIDATA | Read byte data from PPI digital I/O data port. |
| IOCTL_SET_PPICONTROL | Write byte data to PPI digital I/O control port. |
| IOCTL_GET_PPISTATUS | Read data last written to PPI digital I/O control port. |
| IOCTL_SET_PPIABC | Write byte data to PPI ports A, B and C. (Version 4.02 onwards.) |
| IOCTL_SET_PPIXBC | Write byte data to PPI ports B and C. (Version 4.02 onwards.) |
| IOCTL_SET_PPIAXC | Write byte data to PPI ports A and C. (Version 4.02 onwards.) |
| IOCTL_SET_PPIABX | Write byte data to PPI ports A and B. (Version 4.02 onwards.) |
| IOCTL_GET_PPIABC | Read byte data from PPI ports A, B and C. (Version 4.02 onwards.) |

| | |
|---|---|
| IOCTL_GET_IODATA | Read a block of data from any I/O address within card I/O space. (Version 2.00 onwards.) |
| IOCTL_SET_IODATA | Write a block of data to any I/O address within card I/O space. (Version 2.00 onwards.) |
| IOCTL_QUERY_ADCNUMGROUPS | Determine number of ADC channel groups on a card. (Version 4.02 onwards.) |
| IOCTL_QUERY_DACNUMGROUPS | Determine number of DAC channel groups on a card. (Version 4.02 onwards.) |
| IOCTL_QUERY_ADCNUMCHANS | Determine number of channels within an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_QUERY_DACNUMCHANS | Determine number of channels within a DAC channel group. (Version 4.02 onwards.) |
| IOCTL_QUERY_ADCCHANMODE | Get current software unipolar/bipolar settings for channels within an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_QUERY_DACCHANMODE | Get current software unipolar/bipolar settings for channels within a DAC channel group. (Version 4.02 onwards.) |
| IOCTL_QUERY_HWADCCHANMODE | Get current hardware unipolar/bipolar settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_QUERY_HWDACCHANMODE | Get current hardware unipolar/bipolar settings for channels within a DAC channel group. (Version 4.10 onwards.) |
| IOCTL_QUERY_HWADCDIFFMODE | Get current hardware single-ended/ differential settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_QUERY_HWADCCHANGAIN | Get current hardware gain settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_QUERY_HWDACCHANRANGE | Get current hardware output range settings for channels within a DAC channel group. (Version 4.20 onwards.) |
| IOCTL_SET_ADCCHANMODE | Mark channels within an ADC channel group as unipolar or bipolar. (Version 4.02 onwards.) |
| IOCTL_SET_DACCHANMODE | Mark channels within a DAC channel group as unipolar or bipolar. (Version 4.02 onwards.) |
| IOCTL_SET_HWADCCHANMODE | Set hardware ADC unipolar/bipolar settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_SET_HWDACCHANMODE | Set hardware DAC unipolar/bipolar settings for channels within a DAC channel group. (Version 4.10 onwards.) |

| | |
|---|---|
| IOCTL_SET_HWADCDIFFMODE | Set hardware ADC single-ended/ differential settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_SET_HWADCCHANGAIN | Set hardware gain settings for channels within an ADC channel group. (Version 4.10 onwards.) |
| IOCTL_SET_HWDACCHANRANGE | Set hardware output range settings for channels within a DAC channel group. (Version 4.20 onwards.) |
| IOCTL_SET_ADCCONVSRCE | Set conversion trigger source for an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_SET_ADCMUX | Select multiplexed channel for an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_START_ADCCONV | Provides a software trigger for an A-to-D conversion for an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_GET_ADC | Reads and cooks data from the data port of an ADC channel group. (Version 4.02 onwards.) |
| IOCTL_SET_ADCSTARTACQTRIG | Sets ADC start acquisition trigger for an ADC channel group. (Version 4.42 onwards.) |
| IOCTL_GET_ADCSTARTACQ-TRIGGERED | Determines whether start acquisition trigger has occurred for an ADC channel group, and the count of samples stored before the trigger occurred. (Version 4.42 onwards.) |
| IOCTL_SET_DACCHANS | Writes a set of cooked data values to a corresponding set of channels within a DAC channel group. (Version 4.02 onwards.) |
| IOCTL_SET_DACCONVSRCE | Set conversion trigger source for a DAC channel group in FIFO mode. (Version 4.20 onwards.) |
| IOCTL_SET_DACFIFOWAVE | Set DAC FIFO wrap-around mode waveform. (Version 4.20 onwards.) |
| IOCTL_START_DACCONV | Trigger a DAC conversion in FIFO mode. |
| IOCTL_ENABLE_IRQEVENT | Enable an interrupt source and configure which data to fetch or write during the interrupt service. |
| IOCTL_DISABLE_IRQEVENT | Disable an interrupt source. |
| IOCTL_GET_INTSTATUS | Read interrupt status port. |
| IOCTL_WAIT_INTEVENT | Wait for an interrupt event to occur. |
| IOCTL_WAIT_INTEVENT_0 | Waits for interrupt event 0 to occur. (Version 4.02 onwards.) |
| IOCTL_WAIT_INTEVENT_1 | Waits for interrupt event 1 to occur. (Version 4.02 onwards.) |

| | |
|---|---|
| IOCTL_WAIT_INTEVENT_2 | Waits for interrupt event 2 to occur. (Version 4.02 onwards.) |
| IOCTL_WAIT_INTEVENT_3 | Waits for interrupt event 3 to occur. (Version 4.02 onwards.) |
| IOCTL_WAIT_INTEVENT_4 | Waits for interrupt event 4 to occur. (Version 4.02 onwards.) |
| IOCTL_WAIT_INTEVENT_5 | Waits for interrupt event 5 to occur. (Version 4.02 onwards.) |
| IOCTL_WAIT_INTEVENT_READ_DIRECT | Wait for an interrupt event to occur using “direct I/O” method for buffer transfer. (Version 4.43 onwards.) |
| IOCTL_WAIT_INTEVENT_WRITE_DIRECT | Wait for a “write” interrupt event to occur using “direct I/O” method for buffer transfer. (Version 4.43 onwards.) |
| IOCTL_GET_IRQEVENT_ERROR | Determine if overflow or under-run condition has occurred during interrupt processing. (Version 4.23 onwards.) |
| IOCTL_FLUSH_IRQEVENT | Flushes (discards) interrupt user data in the driver including data in FIFO. (Version 4.35 onwards.) |
| IOCTL_EXPEDITE_READ_IRQEVENT | Causes current or following interrupt data buffer for “read” interrupt event to be completed as soon as possible. (Version 5.02 onwards.) |
| IOCTL_GET_IRQEVENT_AVAILABLE_-TO_READ | Determines how much interrupt data is available for a “read” interrupt event. (Version 5.02 onwards.) |

### 7.3.1 Interrupt Data Transfer Types Supported

The IOCTL_ENABLE_IRQEVENT IOCTL call is used to enable an IRQ event. It allows the user to configure the data to fetch during the interrupt service routine. The following data types are supported:

- ISR_NODATA — reads zeroes
- ISR_READ_16COUNT — reads a timer counter channel
- ISR_READ_16COUNTSTAT — reads a TC channel's status and count (Version 4.40 onwards)
- ISR_READ_32COUNT — reads two chained counters
- ISR_READ_32COUNTSTAT — reads two TC channels' status and counts (Version 4.40 onwards)
- ISR_READ_PPIABC — reads data from all 3 PPI ports
- ISR_READ_PPIC — reads data from PPI port C (Version 2.00 onwards)
- ISR_PC27 — reads PC27 ADC port (Version 2.01 onwards)
- ISR_READ_DATA8 — reads data from 1 DIO port (Version 3.00 onwards)
- ISR_READ_DATA16 — reads data from 2 DIO ports (Version 3.00 onwards)
- ISR_READ_ADCS — reads ADC channels (Version 4.00 onwards)
- ISR_READ_ADCSNOFIFO — reads ADC channels (Version 4.02 onwards)
- ISR_READ_ADCSFIFO — reads ADC channels (Version 4.10 onwards)
- ISR_READ_ADCSASAP — reads ADC channels (Version 4.40 onwards)
- ISR_READ_2PPIABC — reads data from 6 PPI ports on 2 PPI chips (Version 4.35 onwards)
- ISR_READ_3PPIABC — reads data from 9 PPI ports on 3 PPI chips (Version 4.35 onwards)
- ISR_WRITE_DATA8 — writes data to 1 DIO port (Version 3.00 onwards)

- ISR_WRITE_DATA16 — writes data to 2 DIO ports (Version 3.00 onwards)
- ISR_WRITE_PPIABC — writes data to all 3 PPI ports (Version 3.00 onwards)
- ISR_WRITE_DACS — writes DAC channels (Version 4.00 onwards)
- ISR_WRITE_2DACS — writes 2 DAC channels (Version 4.00 onwards)
- ISR_WRITE_DACSNOFIFO — writes DAC channels (Version 4.02 onwards)
- ISR_WRITE_DACSFIFO — writes DAC channels (Version 4.10 onwards)
- ISR_WRITE_16COUNT — writes initial count of a timer channel (Version 4.35 onwards)
- ISR_WRITE_32COUNT — writes initial counts of two timer channels (Version 4.35 onwards)
- ISR_WRITE_2PPIABC — writes data to 6 PPI ports on 2 PPI chips (Version 4.35 onwards)
- ISR_WRITE_3PPIABC — writes data to 9 PPI ports on 3 PPI chips (Version 4.35 onwards)

# APPENDIX A GLOSSARY OF TERMS

The following glossary explains some terms used in this manual and in data acquisition and control applications.

**Active Filter:** An electronic filter that combines active circuit devices with passive circuit elements such as resistors and capacitors. Active filters typically have characteristics that closely match ideal filters.

**ADC (A/D):** Analog to Digital converter. *q.v.*

**Alias Frequency:** A false lower frequency component that appears in analog signal reconstructed from original data acquired at an insufficient sampling rate.

**Algorithm:** A set of rules, with a finite number of steps, for solving a mathematical problem. An algorithm can be used as a basis for a computer program.

**Analog to Digital Converter (ADC):** A device for converting an analog voltage to a parallel digital word where the digital output code represents the magnitude of the input signal. See ‘Successive Approximation’.

**Analog Switch:** An electronic, single pole, two way switch capable of handling the full range of analog signal voltage, and operating under the control of a logic signal.

**Array:** Data arranged in single or multidimensional rows and columns.

**ASCII:** American Standard Code for Information Interchange. A code that is commonly used to represent symbols in computers.

**Assembler:** A program that converts a list of computer instructions written in a specific assembly language format that can be executed by a specific processor.

**Bandpass Filter:** A type of electrical filter that allows a band of signals between two set frequencies to pass, while attenuating all signal frequencies outside the bandpass range.

**Base Address:** A unique address set up on an I/O card to allow reference by the host computer. All registers are located by an offset in relation to the base address.

**BASIC:** The most common computer language. BASIC is an acronym for Beginners All-purpose Symbolic Instruction Code. BASIC is not rigorously structured and relies on English-like instructions which account for its popularity.

**Binary Coded Decimal (BCD):** A system of binary numbering where each decimal digit 0 through 9 is represented by a combination of four bits.

**BIOS:** Basic Input Output System. BIOS resides in ROM on a computer system board and provides device level control for the major I/O devices on the system.

**Bipolar:** A signal being measured is said to be bipolar when the voltage on its 'high' terminal can be either of positive or negative polarity in relation to its 'low' terminal.

**Bit:** Contraction of binary digit. The smallest unit of information. A bit represents the choice between a one or zero value (mark or space in communications technology).

**Buffer:** A storage device used to compensate for a difference in rate of data flow, or time of occurrence of events, when transferring data from one device to another. Also a device without storage that isolates two circuits.

**Bus:** Conductors used to interconnect individual circuitry in a computer. The set of conductors as a whole is called a bus.

**Byte:** A binary element string operated on as a unit and usually shorter than a computer word. Normally eight bits.

**C:** A high level programming language, developed around the concept of structured programming and designed for high operating speeds. Microsoft 'C' and Turbo 'C' are dialects of C.

**Channel:** One of several signal/data paths that may be selected.

**Character:** A letter, figure , number, punctuation or other symbol contained in a message or used in a control function.

**Code:** A set of unambiguous rules specifying the way in which characters may be represented.

**Conversion Time:** The time required for a complete conversion of a value from analog to digital form (ADC) or analog to digital form (DAC). Inverse of Conversion Rate.

**Cold Junction:** See Thermocouple Reference Junction

**Cold Junction Compensation (CJC):** A technique to compensate for thermocouple measurement offset when the reference or cold junction is at a temperature other than 0° C.

**Common Mode Rejection Ratio (CMR):** A measure of the equipment's ability to reject common mode interference. Usually expressed in decibels as the ratio between the common mode voltage and the error in the reading due to this common mode voltage.

**Common Mode Voltage:** In a differential measurement system, the common mode voltage usually represents an interfering signal. The common mode voltage is the average of the voltages on the two input signal lines with respect to ground level of the measuring system.

**Comparator:** An electronic circuit used to compare two values and set an indicator that identifies which value is greater.

**Compiler:** High level language used to pre-process a program in order to convert it to a form that a processor can execute directly.

**Contact Closure:** The closing of a switch, often controlled by an electromagnetic or solid state relay.

**Conversion Time:** The time required, in an analog/digital input/output system, from the instant that a channel is interrogated (such as with a read instruction) to the moment that accurate an accurate representation of the data is available. This could include switching time, settling time, acquisition time , converter processing time etc.

**Counter:** In software, a memory location used by a program for the purpose of counting certain occurrences. In hardware, a circuit that can count pulses.

**Counter/Timer Device:** Converts time-dependent digital signals to a form that can be further processed by the host PC. Typical functions include pulse counting, frequency and pulse width measurement. This can relate to time, number of events, speed etc.

**Crosstalk:** A phenomenon in which a signal in one or more channels interferes with a signal or signals in other channels.

**Current Loop: (**a) Data communications method using presence or absence of current to signal logic ones and zeros.

(b) A method of analog signal transmission where the measured value is represented by a current. The common current loop signal is in the range 4 to 20 mA, but other standards include 1 to 5 mA or 10 to 50 mA.

**DAC (D/A):** Digital to Analog Converter. *q.v.*

**Data Acquisition or Data Collection:** Gathering information from sources such as sensors and transducers in an accurate, timely and organised manner.

**Debouncing:** Either a hardware circuit or software delay to prevent false inputs from a bouncing relay or switch contact.

**Decibel (dB):** A logarithmic representation of the ratio between two signal levels.

**Digital-Analog Multiplier:** Same as Multiplying DAC. *q.v.*

**Digital Signal:** A discrete or discontinuous signal; one whose various states are identified with discrete levels or values.

**Digital to Analog Converter:** A device for converting a parallel digital word to an analog voltage, where the magnitude of the output signal represents the value of the digital input.

**DIP Switch:** A set of switches contained in a dual in line package.

**Drift:** Small variations in a measured parameter over a period of time.

**Drivers:** Part of the software that is used to control a specific hardware device.

**Expansion Slots:** The spaces provided in a computer for expansion boards that enhance the basic operations of the computer.

**FIFO:** First In First Out. A buffer memory that outputs data in the same order that they are input.

**Form A, Form B, Form C Contacts**: Relay contact sets which are normally open, normally closed and changeover respectively.

**Four Quadrant Operation:** In a multiplying DAC, four quadrant operation means that both the reference signal and the number represented by the digital input may both be either positive or negative polarity. The output obeys the rules of multiplication for algebraic sign.

**GAL (Generic Array Logic):** Programmable logic device where the architecture and functionality of each output is defined by the system designer.

**Handshaking:** Exchange of predetermined codes and signals between two data devices to establish and control a connection.

**Hardware:** The visible parts of a computer system such as the circuit boards, chassis, peripherals, cables etc. It does not include data or computer programs.

**Hexadecimal (Hex):** A numbering system to the base 16.

**Input/Output (I/O):** The process of transferring data from or to a computer system including communication channels, operator interface devices or data acquisition and control channels.

**Interface:** A shared boundary defined by common physical interconnection characteristics, signal characteristics and meanings of interchanged signals.

**Interrupt:** A computer signal indicating that the CPU should suspend its current task to service a designated activity.

**I/O Address:** A method that allows the CPU to distinguish between different boards and I/O functions in a system. See Base Address.

**Latch:** A device to store the state of a digital signal until it is changed by another external command signal. The action of storing this signal.

**Least Significant Bit (LSB):** In a system in which a numerical magnitude is represented by a series of digits, the least significant bit (binary digit) is the digit that carries the smallest value or weight.

**Linearity:** Compliance with a straight line law between the input and output of a device.

**Load Voltage Sensing:** A technique for maintaining accuracy of an analog signal at the load by monitoring the voltage and compensating for errors due to cable and source resistance.

**Micro Channel Architecture (MCA):** A unique architecture defined by IBM™ to provide a standard input/output bus for Personal System computers.

**Monotonic:** A DAC is said to be monotonic if the output increases as the digital input increases, with the result that the output is always a single valued function of the input.

**Most Significant Bit (MSB):** In a system in which a numerical magnitude is represented by a series of digits, the most significant bit (binary digit) is the digit that carries the greatest value or weight.

**Multiplexer:** A multiple way analog switch *q.v.*, where a single path through the switch is selected by the value of a digital control word.

**Multiplying DAC:** A Multiplying DAC (or Digital-Analog Multiplier) operates with varying or AC reference signals. The output of a Multiplying DAC is proportional to the product of the analog 'reference' signal and the fractional equivalent of the digital input number.

**Noise:** An undesirable electrical interference to a signal.

**Normal Mode Signal:** Aka Series mode signal. In a differential analog measuring system, the normal mode signal is the required signal and is the difference between the voltages on the two input signal lines with respect to ground level of the measuring system.

**Offset:** (a) A fixed, known voltage added to a signal.
(b) The location of a register above the base address.

**Pascal:** A high level programming language originally developed as a tool for teaching the concepts of structured programming. It has evolved into a powerful general-purpose language popular for writing scientific and business programs. Borland Turbo Pascal is a dialect of Pascal.

**Passive Filter:** A filter circuit using only resistors, capacitors and inductors.

**PC:** Personal Computer (Also printed circuit - see PCB)

**PCB:** Printed Circuit Board

**Port:** An interface on a computer capable of communication with another device.

**Range:** Refers to the maximum allowable full-scale input or output signal for a specified performance.

**Real Time:** Data acted upon immediately instead of being accumulated and processed at a later time.

**Reed Relay:** An electro-mechanical relay where the contacts are enclosed in a hermetically sealed glass tube which is filled with an inert gas.

**Repeatability:** The ability of a measuring system to give the same output or reading under repeated identical conditions.

**Resolution:** A binary converter is said to have a resolution of n-bits when it is able to relate $2^n$ distinct analog values to the set of n-bit binary words.

**Rollover:** Symmetry of the positive and negative values in a bipolar conversion system.

**RTD (Resistive Temperature Device):** An electrical circuit element characterised by a defined coefficient of resistivity.

**Sample/Hold:** A circuit which acquires an analog voltage and stores it for a period of time.

**Sensor:** Device that responds to a physical stimulus (heat, light, sound, pressure, motion etc.) producing a corresponding electrical output.

**Settling Time:** The time taken for the signal appearing at the output of a device to settle to a new value caused by a change of input signal.

**Signal to Noise Ratio:** Ratio of signal level to noise in a circuit. Normally expressed in decibels.

**Simultaneous Sample/Hold:** A data acquisition system in which several sample/hold circuits are used to simultaneously sample a number of analog channels and hold these values for sequential conversion. One sample/hold circuit per analog channel is required.

**Software:** The non-physical parts of a computer system that includes computer programs such as the operating system, high level languages, applications program etc.

**Spike:** A transient disturbance of an electrical circuit.

**Stability:** The ability of an instrument or sensor to maintain a consistent output when a consistent input is applied.

**Successive Approximation:** An analog to digital conversion method that sequentially compares a series of binary weighted values with the analog input signal to produce an output digital word in ‘n’ steps where ‘n’ is the number of bits of the A/D Converter. *q.v.*

**Symbol:** The graphical representation of some idea. Letters and numerals are symbols.

**Syntax:** Syntax is the set of rules used for forming statements in a particular programming language.

**Thermocouple:** A thermocouple is two dissimilar electrical conductors, known as thermo-elements, so joined as to produce a thermal emf when the measuring and reference junctions are at different temperatures.

**Thermocouple Measuring Junction:** The junction of a thermocouple which is subjected to the temperature being measured.

**Thermocouple Reference Junction:** The junction of a thermocouple which is at a known temperature. aka Cold Junction.

**Throughput Rate:** The maximum repetitive rate at which a data conversion system can operate with a specified accuracy. It is determined by summing the various times required for each part of the system and then taking the reciprocal of this time.

**Transducer:** Device that converts length, position, temperature, pressure, level or other physical variable to an equivalent voltage or current accurately representing the original measurement.

**Trigger:** Pulse or signal used to start or stop a particular action. Frequently used to control data acquisition processes.

**Unipolar:** A signal being measured is said to be unipolar when the voltage on its 'high' terminal is always the same polarity (normally positive) in relation to its 'low' terminal.

**Word:** The standard number of bits that can be manipulated at once. Microprocessors typically have word lengths of 8, 16 or 32 bits.

**Wrap, Wraparound:** Connection of a FIFO buffer such that the contents once loaded, are continuously circulated.

# INDEX OF FUNCTIONS