**ENGINEERING, INC.**

# DAQDRIVE®

## Data Acquisition Software

## Users Manual

INTERFACE CARDS FOR PERSONAL COMPUTERS


OMEGA ENGINEERING, INC.
One Omega Drive
P.O. Box 4047
Stamford, CT 06907-4047

Tel: (203) 359-1660
Fax: (203) 359-7700
Toll free: 1-800-826-6342
E-mail: das@omega.com

http://www.dasieee.com

# WARRANTY/DISCLAIMER

OMEGA ENGINEERING, INC., warrants this unit to be free of defects in materials and workmanship for a period of **13 months** from the date of purchase. OMEGA warranty adds an additional one (1) month grace period to the normal **one (1) year product warranty** to cover shipping and handling time. This ensures that OMEGA's customers receive maximum coverage on each product. If the unit should malfunction, it must be returned to the factory for evaluation. OMEGA's Customer Service Department will issue an Authorized Return (AR) number immediately upon phone or written request. Upon examination by OMEGA, if the unit is found to be defective it will be repaired or replaced at no charge. OMEGA's warranty does not apply to defects resulting from any action of the purchaser, including but not limited to mishandling, improper interfacing, operation outside design limits, improper repair or unauthorized modification. This WARRANTY is VOID if the unit shows evidence of having been tampered with or shows evidence of being damaged as a result of excessive corrosion; or current, heat, moisture or vibration; improper specification; misapplication; misuse or other operating conditions outside of OMEGA's control. Components which wear are not warranted, including but not limited to contact points, fuses and triacs.

**OMEGA is pleased to offer suggestions on the use of its various products. However, OMEGA neither assumes responsibility for any omissions or errors nor assumes liability for any damages that result from the use of its products in accordance with information provided from OMEGA, either verbal or written. OMEGA warrants only that the parts manufactured by it will be as specified and free of defects. OMEGA MAKES NO OTHER WARRANTIES OR REPRESENTATIONS OF ANY KIND WHATSOEVER, EXPRESSED OR IMPLIED, EXCEPT THAT OF TITLE, AND ALL IMPLIED WARRANTIES INCLUDING ANY WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE HEREBY DISCLAIMED. LIMITATION OF LIABILITY: The remedies of purchaser set forth herein are exclusive and the total liability of OMEGA with respect to this order, whether based on contract, warranty, negligence, indemnification, strict liability or otherwise, shall not exceed the purchase price of the component upon which liability is based. In no event shall OMEGA be liable for consequential, incidental or special damages.**

CONDITIONS: Equipment sold by OMEGA is not intended to be used**,** nor shall it be used: (1) as a "Basic Component" under 10 CFR 21 (NRC), used in or with any nuclear installation or activity, medical application or used on humans. Should any Product(s) be used in or with any nuclear installation or activity, medical application, used on humans or misused in any way, OMEGA assumes no responsibility as set forth in our basic WARRANTY/DISCLAIMER language, and additionally, purchaser will indemnify OMEGA and hold OMEGA harmless from any liability or damage whatsoever arising out of the use of the Product(s) in such a manner.

# RETURN REQUESTS/INQUIRIES

Direct all warranty and repair requests/inquiries to the OMEGA Customer Service Department. BEFORE RETURNING ANY PRODUCT(S) TO OMEGA, PURCHASER MUST OBTAIN AN AUTHORIZED RETURN (AR) NUMBER FROM OMEGA'S CUSTOMER SERVICE DEPARTMENT (IN ORDER TO AVOID PROCESSING DELAYS). THE ASSIGNED NUMBER SHOULD THEN BE MARKED ON THE OUTSIDE OF THE RETURN PACKAGE AND ON ANY CORRESPONDENCE. THE PURCHASER IS RESPONSIBLE FOR SHIPPING CHARGES, FREIGHT, INSURANCE AND PROPER PACKAGING TO PREVENT BREAKAGE IN TRANSIT.

FOR **WARRANTY** RETURNS, please have the following information available BEFORE contacting OMEGA:
(1) P.O. Number under which the product was purchased,
(2) Model and serial number of the product under warranty, and
(3) Repair instructions and/or specific problems relative to the product.

FOR **NON-WARRANTY** REPAIRS, consult OMEGA for current repair charges. Have the following information available BEFORE contacting OMEGA:
(1) P.O. Number to cover the cost of the repair,
(2) Model and serial number of the product, and
(3) Repair instructions relative to the product.

OMEGA's policy is to make running changes, not model changes, whenever an improvement is possible. This affords our customers the latest in technology and engineering.

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

## Servicing North America:

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

**Canada**: 976 Bergar E-mail: info@omega.com
Laval (Quebec) H7L 5A1
Tel: (514) 856-6928 FAX: (514) 856-6886

## For immediate technical or application assistance:

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

**Mexico and Latin America**: Tel: (001) 800-826-6342 FAX: (001) 203-359-7807
En Espanol: (001) 203-359-7803 E-mail: espanol@omega.com

## Servicing Europe:

**Benelux**: Postbus 8034, 1180 LA Amstelveen, The Netherlands
Tel: (31) 20 6418405
Toll Free in Benelux: 0800 0993344
E-mail: nl@omega.com

**Czech Republic**: ul.Rude armady 1868, 733 01 Karvina-Hraniee
Tel: 42 (69) 6311899 FAX: 42 (69) 6311114
Toll Free: 0800-1-66342 E-mail: czech@omega.com

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

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

**United Kingdom**:
ISO 9002 Certified

One Omega Drive, River Bend Technology Drive
Northbank, Irlam, Manchester
M44 5EX, England
Tel: 44 (161) 777-6611
FAX: 44 (161) 777-6622
Toll Free in England: 0800-488-488
E-mail: info@omega.co.uk

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

The information contained in this document is believed to be correct but OMEGA Engineering, Inc. accepts no liability for any errors it contains, and reserves the right to alter specifications without notice. **WARNING**: These products are not designed for use in, and should not be used for, patient connected applications.

# Table of Contents

# List of Figures

# 1 Introduction

DAQDRIVE is Omega's universal data acquisition interface for the "DAQ" series of ISA bus and PCMCIA data acquisition adapters. DAQDRIVE goes beyond the drivers normally distributed with data acquisition adapters by isolating the application programmer from the hardware.

DAQDRIVE provides support for application programs written in the following languages:

- Microsoft C/C++
- Borland C/C++
- Visual Basic for DOS
- Quick Basic version 4.5
- Turbo Pascal for DOS version 7.0 and newer
- Most Windows languages supporting Dynamic Link Libraries (DLLs) including Visual C/C++, Borland C/C++, Turbo Pascal for Windows, and Borland Delphi

DAQDRIVE uses a "data defined" rather than a "function defined" interface. What this means is that each data acquisition operation is defined by a series of configuration parameters and requires very few function calls to implement. Because of this approach, DAQDRIVE may seem a little unusual; even intimidating at times. However, after writing a few example programs, we feel the user will discover the power behind this type of interface.

DAQDRIVE supports high speed data I/O by providing support for foreground (CPU software polled) and background (DMA and interrupt driven) operation. For increased flexibility, DAQDRIVE also supports software (internal) and hardware (external) clock and trigger sources.

DAQDRIVE supports multiple data acquisition adapters in a single system. In fact, the number of adapters is limited only by the amount of available system memory. DAQDRIVE also supports multiple tasks from one or more applications operating on one or more hardware devices. This multi-tasking support is accomplished by tracking all system and data acquisition resources and rejecting any request for which all of the necessary resources are not available.

In order to minimize the code size of the application programs, DAQDRIVE is distributed as a two-part driver. The first part contains the application program interface (API) and is also responsible for memory management, file I/O, and other hardware independent functions. Regardless of the number of hardware devices installed, only one copy of the hardware independent driver is required.

The second part of the driver is hardware dependent and is responsible for implementing the requested operations on the target hardware device. These drivers are supplied with the data

acquisition adapter and generally support only one family of hardware devices. Only one hardware dependent driver is required for each family of hardware installed in the system.

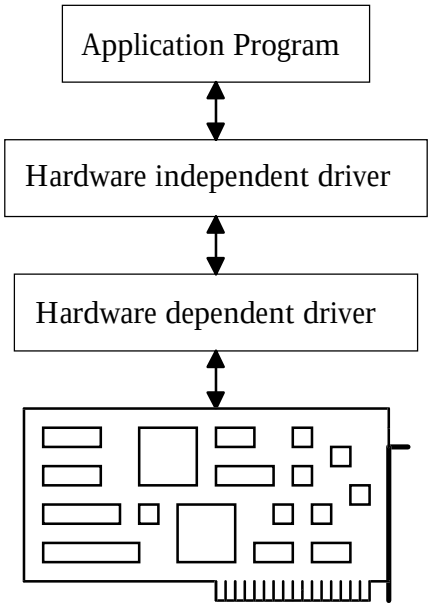



Figure 1. DAQDRIVE interface between an application program and one hardware device.

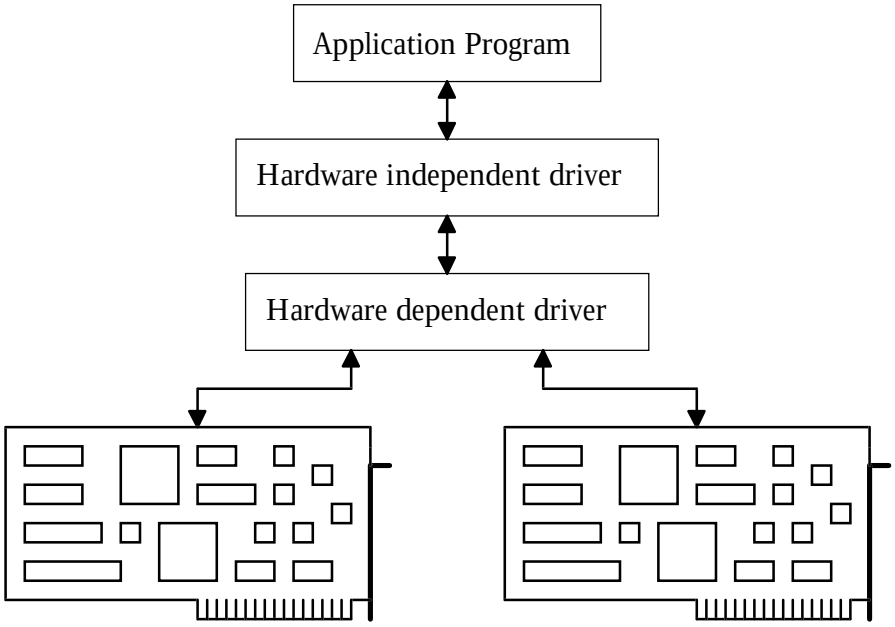



Figure 2. DAQDRIVE interface between an application program and multiple devices of the same family.

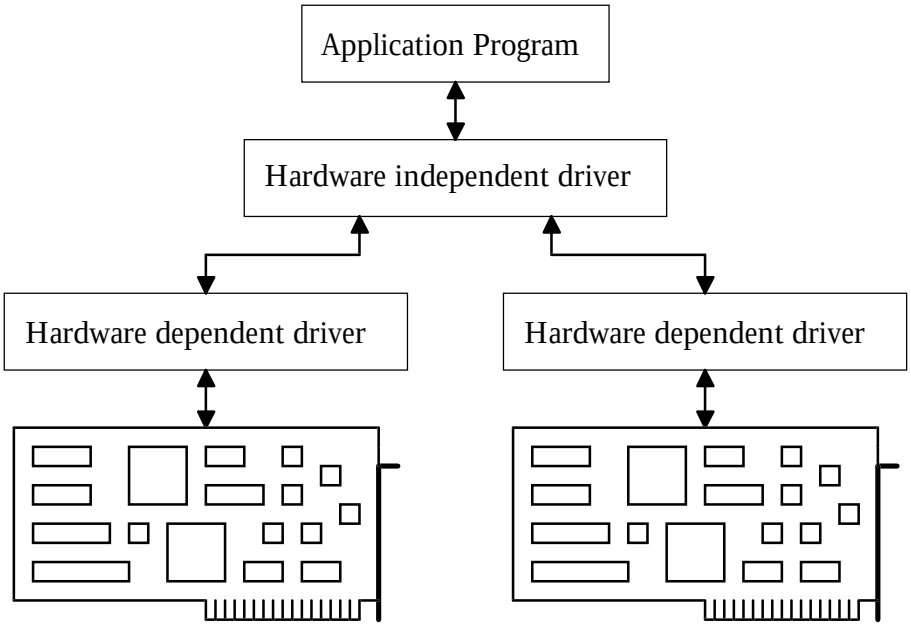



Figure 3. DAQDRIVE interface between an application program and multiple devices of different families.

# 2 Before Beginning

## 2.1 Software Installation

The DAQDRIVE distribution CD contains a Setup program that allows the user to quickly and easily install the necessary DAQDRIVE components onto the host computer. The Setup program is compatible with Windows 3.x and Windows 95/98 and allows the user to install DAQDRIVE for DOS, Windows 3.x, and/or Windows 95/98.

From Windows 3.x:

1. Insert the compact disk into the computer's CD-ROM drive.
2. From the Windows program manager, select **File** then **Run**.
3. Assuming the CD-ROM drive is drive D, enter "**D:\SETUP**" in the command line text box and click OK.

From Windows 95, Windows 98, and Windows NT 4.0:

1. Insert the compact disk into the computer's CD-ROM drive.
2. Click the **Start** button, point to **Settings**, then click **Control Panel**.
3. Double-click on **Add/Remove Programs**.
4. On the **Install/Uninstall** tab, click **Install**.
5. Click **Next**.
6. If the correct Setup program is found, click **Finish**. If not, click **Browse** and select the Setup program in the root directory of the CD.

Follow the on-screen instructions to select the DAQDRIVE components to be installed. When the Setup program is complete, one or more of the following subdirectories will have been created in the target directory:

| | |
|---|---|
| ...\DAQDRIVE\CONFIG | DAQDRIVE Configuration Utility |
| ...\DAQDRIVE\DAQEZ | DaqEZ |
| ...\DAQDRIVE\C_LIBS | C library support for DOS applications |
| ...\DAQDRIVE\TSR | TSR driver support for DOS applications |
| ...\DAQDRIVE\WINDLL | Support for Windows 3.x and 16-bit Windows 95 applications |
| ...\DAQDRIVE\VISDAQLT | Support for 16-bit Visual Basic applications |
| ...\DAQDRIVE\WIN32 | Support for 32-bit Windows 95 applications |

## 2.2 DAQDRIVE Configuration Utilities

Before DAQDRIVE can operate an adapter, a configuration file must be generated to specify the hardware configuration. Three separate Windows based utility programs are provided to generate these configuration files:

1. DAQCFGW.EXE utility to edit DAQDRIVE hardware adapter configuration files
2. EXPBOARD.EXE utility to edit the data base defining available A/D expansion boards and their parameters
3. SIGCON.EXE utility to edit the data base defining available A/D channel signal conditioners and their parameters

**IMPORTANT:**

The DAQDRIVE configuration utilities must be used to edit the DAQDRIVE hardware configuration files. Under no circumstances should the user attempt to create and/or edit DAQDRIVE hardware configuration files directly.

### 2.2.1 Installation

The DAQDRIVE configuration utilities are automatically installed into the ..\DAQDRIVE\CONFIG subdirectory whenever the Setup program is executed. In addition, the Setup program installs sample hardware configuration data files (.DAT), and their associated report files (.RPT). These sample configuration files must be modified to create the user's configuration as DAQCFGW does not allow the creation of new hardware configuration data files, but instead requires all files to be a modified version of an existing file.

DAQCFGW does not allow the creation of new hardware configuration data files, but instead requires all files to be a modified version of an existing data file. The DAQDRIVE installation program installs the necessary sample hardware configuration data files (.DAT), and their associated report files (.RPT), into the ..\DAQDRIVE\CONFIG directory along with the configuration utilities.

**CAUTION:**

Older versions of DAQDRIVE may not be compatible with files generated by the latest configuration utilities.

### 2.2.2 Generating A DAQDRIVE Configuration File

DAQCFGW does not allow the creation of new data files but instead requires all files to be a modified version of an existing data file (*.DAT). For this reason, one or more sample data files are provided on the DAQDRIVE installation diskettes. To view and/or edit a configuration data file:

1. Execute DAQCFGW by double-clicking on the DAQDRIVE configuration utility icon located in the DAQDRIVE program group
2. Select **F**ile, **O**pen
3. Select the drive and directory in the corresponding list boxes.
4. Type the name of an existing configuration data file (.DAT) in the file name text box or select the file from the corresponding list box.
5. Choose **OK**.

Some or all of the following configuration options will appear in the **H**ardware Setup menu:

- General
- A/D Converter
- A/D Expansion Boards
- A/D Signal Conditioners
- D/A converter
- Timer
- Digital I/O
- Configuration Help

To select a subsystem for configuration, select it from the **H**ardware Setup menu or click the associated toolbar icon. If a specific subsystem is not available on the adapter or if there are no user-definable options within that subsystem, the option will be disabled. The hardware specific appendix for the adapter being configured lists the available options.

#### 2.2.2.1 General Configuration

The general configuration window is used to define the interface between the adapter and the host system. All adapters require the general configuration options:

Base Address

The base I/O address of the adapter must be specified using the base address text box. If the adapter is PCMCIA, or PCI compatible, the user may specify a base address of 0. Setting the base address to 0 instructs DAQDRIVE to determine the adapter's base address, interrupt, and DMA settings automatically each time the device is opened.

IRQ Level

The adapter's interrupt level (IRQ) must be selected from the corresponding drop-down list box. If the adapter does not support interrupts or if the base I/O address is set to 0, the interrupt list box is not displayed.

DMA Channel 1
The adapter's primary DMA channel must be selected from the corresponding drop-down list box. If the adapter does not support DMA or if the base I/O address is set to 0, the primary DMA list box is not displayed.

DMA Channel 2
The adapter's secondary DMA channel must be selected from the corresponding drop-down list box. If the adapter does not support two DMA channels, or if the base I/O address is set to 0, the secondary DMA list box is not displayed.

### 2.2.2.2 A/D Converter Configuration

The A/D converter window is used to define the configuration of the adapter's analog input channels. When these parameters define a specific jumper setting on the adapter, it is the user's responsibility to assure the adapter is configured properly. The Configuration **H**elp window provides information regarding hardware modification requirements (see page 29).

The number and type of user-definable options available in this window is dependent on the hardware installed and is discussed in the hardware specific appendix for the adapter being configured.

A/D Converters
Select the analog-to-digital (ADC) device on the A/D adapter to be configured from this list box. Most A/D adapters have only one ADC device (ADC 0).

Channels
Dialog box shows the number of A/D input channels available on the adapter in its current configuration. A multiplexer (mux) feeds multiple analog inputs back into the actual ADC device(s). The number of channels may be affected by the Input Mode.

Input Mode
Select the A/D input mode from the list box.

- Single Ended: A/D converter measures the voltage from one input to ground. All A/D channels normally share common ground.
- Differential: A/D converter measures the voltage difference between two inputs that are isolated from ground.

Signal Type
Select the signal type from the list box.

- Unipolar: A/D converter measures only positive voltages.
- Bipolar: A/D converter measures both positive and negative voltages.

Gain
This list box provides optional signal amplifier settings. Note that this option is only available on devices with hardware selectable gain settings. Devices with software programmable gains are configured at run-time.

2.2.2.3 A/D Converter Expansion Configuration

The A/D converter expansion window is used to define the configuration of any analog input expansion adapters connected to the analog input channels. To assign an expansion board to a main A/D channel click in the **Expansion Board Names** column and a choose from the drop down list box. The first analog input expansion board must always be connected to A/D channel 0, and additional expansion boards then connect to the next lowest channel.

Expansion adapters are defined in a data base using the **EXPBOARD** utility. The expansion board data base may be viewed from the **D**ataBase menu. However, to add or edit the expansion board data base this utility must be run independently (see page 22).

The number and type of user-definable options available in this window is dependent on the hardware installed and the configuration of the expansion board as defined by the **EXPBOARD** utility. When these parameters define a specific jumper setting on the expansion board adapter, it is the user's responsibility to assure the adapter is configured properly.

The parameters in this window refer only to the expansion board adapter and do not effect the A/D converter configuration of the main board. In most cases however, these two sets of parameters must be examined together. For example, a gain of 2 in the A/D converter configuration combined with a gain of 10 on the analog input expansion board results in an overall gain of 20.

Channels

Dialog box shows the number of analog input channels available on the expansion board in its current configuration. The values in this box are defined in the **EXPBOARD** utility. The number of channels may be effected by the Input Mode.

Input Mode

Select the input mode from the list box.

- Single Ended: input signals are measured from one input to ground. All inputs normally share a common ground.
- Differential: input signals are measured as the difference between two inputs that are isolated from ground.

Signal Type

Select the signal type from the list box.

- Unipolar: expansion board accepts only positive voltages.
- Bipolar: expansion board accepts both positive and negative voltages.

Gain

This list box provides optional signal amplifier settings. Note that this option is only available on devices with hardware selectable gain settings. Devices with software programmable gains are configurable at run-time.

2.2.2.4 A/D Signal Conditioners

A signal conditioner may be connected to any A/D main channel, and/or to any A/D expansion channel marked "Signal Conditioner Connectable" in the **EXPBOARD** utility (see page 24). Expansion boards are normally used in conjunction with signal conditioners, but are not required. To assign a signal conditioner to an A/D channel click in the **Signal Conditioner Name** column and a choose from the drop down list box.

Signal conditioners are defined in a data base using the **SIGCON** utility. The signal conditioner data base may be viewed from the **D**ataBase menu. However, to add or edit the signal conditioner data base this utility must be run independently (see page 24).

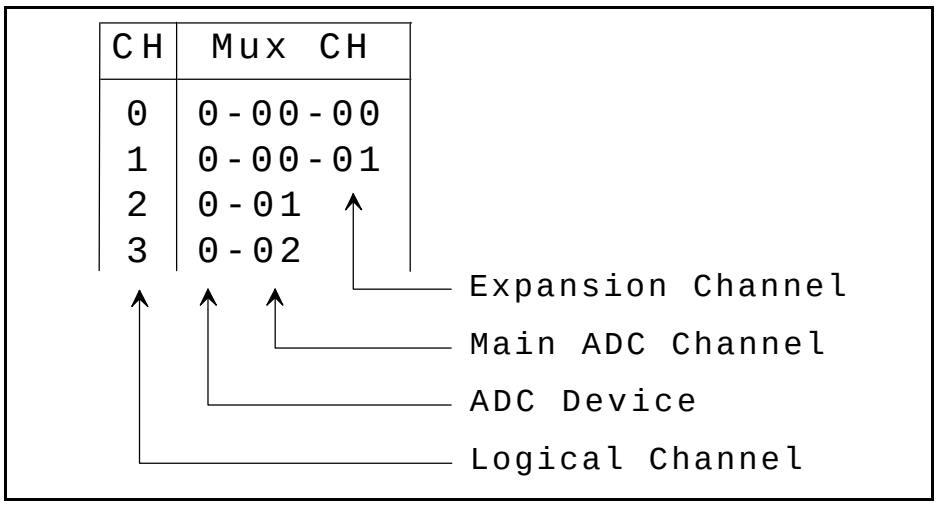



**Figure 1. A/D Channel Numbering**

To help understand the A/D channel numbering system the following terms are defined:

- Logical Channel: The CH column designates the logical number that software should use to access the analog input channel. When using expansion boards you may have up to 256 logical channels.
- ADC Device: The ADC device number. Most A/D adapters have only one ADC device (ADC 0).
- Main ADC Channel: Analog input channel on the A/D adapter board. A multiplexer (mux) on the A/D adapter feeds multiple analog inputs back into the actual ADC device(s).
- Expansion Channel: Analog input channel provided by an expansion board connection to a single main analog channel. Expansion boards use digital I/O to address multiple expansion channels from a single main channel through a multiplexer.

2.2.2.5 D/A Converter Configuration

The D/A converter window is used to define the configuration of the adapter's analog output channels. When these parameters define a specific jumper setting on the adapter, it is the user's responsibility to assure the adapter is configured properly. The Configuration **H**elp window provides information regarding hardware modification requirements (see page 20).

The number and type of user-definable options available in this window is dependent on the hardware installed and is discussed in the hardware specific appendix for the adapter being configured.

D/A Channels

Select the D/A channel to configure from the list. Each D/A channel typically has its own digital-to-analog converter (DAC).

Signal Type

Select the signal type from the list box.

- Unipolar: DAC device outputs only positive voltages.
- Bipolar: DAC device outputs both positive and negative voltages.

Ref. Source

Analog output from DAC is proportional to a reference voltage. Select the voltage source from the list box.

- Internal: Reference voltage generated by adapter board.
- External: Reference voltage supplied by an external source.

Ref. Voltage

The reference voltage is used as scaling multiplier for DAC output. For example, on a 12-bit unipolar operation the analog output can be calculated from the equation:

$$V_{OUT} = V_{REF} * (Digital\_Count / 4096) * Gain$$

Gain

This list box provides optional signal amplifier settings. Note that this option is only available on devices with hardware selectable gain settings. Devices with software programmable gains are configurable at run-time.

2.2.2.6 Digital I/O Configuration

The digital I/O window is used to define the configuration of the adapter's digital input / output channels. When these parameters define a specific jumper setting on the adapter, it is the user's responsibility to assure the adapter is configured properly. The Configuration **H**elp window provides information regarding hardware modification requirements (see page 20).

The number and type of user-definable options available in this window is dependent on the hardware installed and is discussed in the hardware specific appendix for the adapter being configured.

Channel Configuration

Each digital I/O bit on an adapter can be individually accessed though the connector for control/monitoring of external digital devices. The digital I/O bits on each adapter must be configured into logical channels. Digital I/O channels can be set only 1 bit wide to access single I/O lines at the connector, or logical channels that access multiple I/O bits simultaneously are configurable.

Assign a logical channel number to the target digital I/O bit by clicking on the current logical channel number. A drop down channel selection box will appear with the possible channel configurations for this I/O bit (see Figure 2). The rest of the digital I/O bits will be automatically updated with correct channel numbers reflecting any changes. Repeat this step for each digital I/O bit.

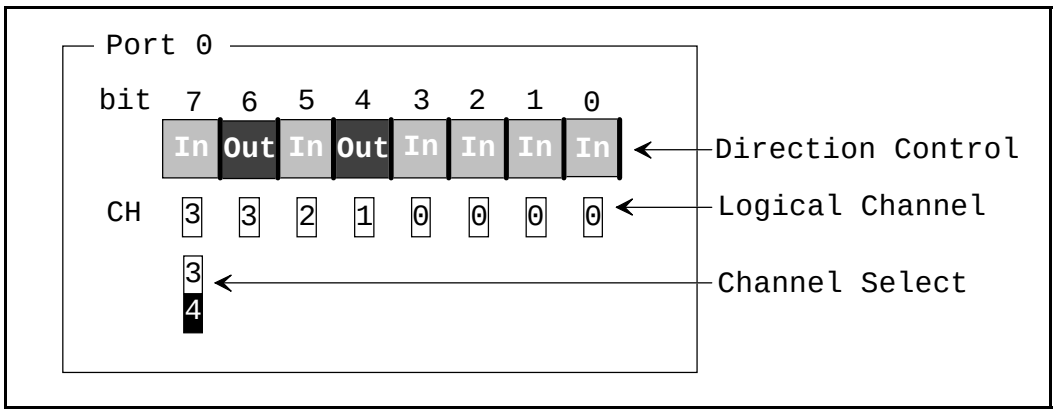



**Figure 2. Digital I/O Configuration Display**

Input/Output Configuration

After all of the logical channels have been defined, they may be configured for input, output, or input/output (I/O) by clicking on the direction control button for each logical channel. All bits defined as a member of that logical channel will toggle between the available settings.

2.2.2.7 Timer Configuration

The timer configuration window is used to define the adapter's onboard counter / timer circuits. Examples of settings found in this section include counter size and input clock frequency. When these parameters define a specific jumper setting on the adapter, it is the user's responsibility to assure the adapter is configured properly. The Configuration **H**elp window provides information regarding hardware modification requirements (see page 20).

The number and type of user-definable options available in this window is dependent on the hardware installed and is discussed in the hardware specific appendix for the adapter being configured.

#### 2.2.2.8 Configuration Help

The hardware configuration of the adapter is the responsibility of the user. Some of these hardware configuration settings may be handled through software, while others may require switches or jumper blocks to be modified. The configuration help window provides the user with the jumper block or switch numbers to modify if required.

It is the responsibility of the user to determine the correct settings for the current hardware configuration. This help window is only a tool to assist the user in determining if and/or where hardware modifications are required. The amount and type of information available in this window is dependent on the hardware installed. No information is provided for configuration options handled through software.

#### 2.2.2.9 Saving The New Configuration

After the adapter configuration is complete, the user may overwrite the current configuration file or a new configuration file can be generated. To overwrite the existing configuration, simply select **F**ile, **S**ave from the menu. To generate a new configuration file:

1. Select **F**ile, Sa**v**e As
2. Select the drive and directory in the corresponding list boxes.
3. Type the name of the new configuration data file in the file name text box.
4. Choose **OK**.

When the user saves an adapter configuration, a corresponding report file is generated using the same file name with the extension .RPT. This report file provides an ASCII description of the hardware configuration and may be viewed using any ASCII text editor.

#### 2.2.2.10 Viewing the Report File

DAQCFGW also provides a utility for viewing the configuration report file (.RPT) generated when the data file was saved.

1. Select **F**ile, View **R**eport
2. Select the drive and directory in the corresponding list boxes.
3. Type the name of a report file (.RPT) in the file name text box or select a report from the corresponding list box.
4. Choose **OK**.
5. Review the adapter's configuration using the Page-Up, Page-Down, and arrow keys as well as the vertical and horizontal scroll bars.
6. When done, close the report viewer utility by selecting **C**lose.

### 2.2.3 A/D Expansion Board Database Utility

DAQCFGW uses a database of analog input expansion boards to assist the user in the configuration of a complete data acquisition system. Omega expansion boards are predefined in the database and may not be modified by the user. New adapters may be added to the database using the **EXPBOARD** utility. Any modification to an expansion board configuration is automatically updated in the database file and made available to the DAQDRIVE configuration utility. To create and/or edit a user-defined analog input expansion board:

1. Execute EXPBOARD by double-clicking on the expansion board utility icon located in the DAQDRIVE program group.
2. Select **ADD NEW**, **EDIT**, or **DELETE** to update the expansion board database.

The parameters for each expansion board in the database refer only to the expansion board adapter and do not effect the A/D converter configuration of the main board. In most cases however, these two sets of parameters must be examined together. For example, a gain of 2 in the A/D converter configuration combined with a gain of 10 on the expansion board results in an overall gain of 20.

When these parameters define a specific jumper setting on the expansion board adapter, it is the user's responsibility to assure the adapter is configured properly. Refer to the expansion board documentation for details and parameter values.

Long Device Name

Each expansion adapter must have a unique **Long Device Name** of 1 - 30 characters. The long device name used only for descriptive purposes.

Device Name

Each expansion adapter must have a unique **Device Name** of 14 characters or less.

Input Mode

Select the A/D input mode from the list box.

- Single Ended: A/D converter measures voltage from one input to ground. All A/D channels normally share common ground.
- Differential: A/D converter measures voltage across two inputs that are isolated from ground.
- DI/SE Selectable

Signal Type

Select the signal type from the list box.

- Unipolar: Expansion device reads only positive voltage.
- Bipolar: Device reads both positive and negative voltages.
- Selectable: Either of the signal types is available.

Num Gains

The **Num Gains** box refers to the number of gains available to the programmer at run-time. Therefore, if the expansion board has 4 software selectable gains this field should be set to 4. But, if the expansion board has 4 hardware (jumper or switch) selectable gains, the **Num Gains** field should be set to 1. In both cases fill in **Gains** list with all available gains.

Differential

Specify the number of differential A/D Expansion channels available on the expansion board. A typical expansion board will connect to one A/D main channel on the A/D adapter and provide up to 8 differential expansion inputs.

Single Ended

Specify the number of Single Ended A/D Expansion channels available on the expansion board. A typical expansion board will connect to one A/D main channel on the A/D adapter and provide up to 16 single ended expansion inputs.

Gains List

List all analog input amplier gains available on expansion board. When the DAQDRIVE Configuration utility is run, the A/D expansion board configuration section will include a **Gains** list box to select the expansion board gain setting if the expansion board has hardware selectable gains. Otherwise, the programmer may select any of the available expansion board gains through software at run-time.

Maximum One Channel Frequency

Maximum sampling rate for a single A/D input.

Maximum Multi-Channel Frequency

Maximum scan rate for multiple A/D inputs. Normally slower than the one channel maximum frequency since the multiplexer must switch between A/D inputs.

Channel Signal Type

Select the channel signal type from the list box. Selection determines whether the DAQDRIVE Configuration utility will allow the use of signal conditioners.

- Direct Analog Signal Connection: Expansion device supports only direct analog signal inputs.
- Signal Conditioner Connectable: Expansion device supports the use of signal conditioners.

### 2.2.4 Signal Conditioner Database Utility

DAQCFGW uses a database of analog input signal conditioners to assist the user in the configuration of a complete data acquisition system. Many standard signal conditioners are predefined in the database and may not be modified by the user. New conditioners may be added to the database using the **SIGCON** utility. Any modification to a signal conditioner's parameters are automatically updated in the database file and made available to the DAQDRIVE configuration utility. To create and/or edit a user-defined analog input signal conditioner:

1. Execute **SIGCON** by double-clicking on the signal conditioner utility icon located in the DAQDRIVE program group.
2. Select **ADD NEW**, **EDIT**, or **DELETE** to update the signal conditioner database.

The parameters for each signal conditioner in the database refer only to the signal conditioner and do not effect the A/D converter configuration or the expansion board configuration. In most cases however, these sets of parameters must be examined together to determine the overall configuration. Refer to the signal conditioner documentation for details and parameter values.

Long Device Name
Each device must have a unique **Long Device Name** of 1 - 30 characters. The long device name used only for descriptive purposes.

Device Name
Each device must have a unique **Device Name** of 14 characters or less.

Device Type
Select the device type from the list box.

- Linear
- Nonlinear

Minimum Input
Minimum value the signal conditioner is capable of reading. Type of signal is specified by **Input Units**.

Maximum Input
Maximum value the signal conditioner is capable of reading. Type of signal is specified by **Input Units**.

Input Units

Select the measurement units for the signal type from the list box. Below is a sample of the choices.

- V (volts)
- A (amps)
- Degree C (temperature)
- Kg (kilogram)
- Hz (frequency)
- $m/sec^2$ (acceleration)

Minimum Output

Minimum value the signal conditioner returns to the A/D input . Type of signal is specified by **Output Units**.

Maximum Output

Maximum value the signal conditioner returns to the A/D input . Type of signal is specified by **Output Units**.

Output Units

Specifies the measurement units for the signal type returned to the A/D converter. Currently signal is always of type volts.

Bandwidth

Maximum frequency at which the signal conditioner can process data.

Maximum Scan Rate

Maximum frequency at which multiple devices may be scanned. This rate is normally slower than the bandwidth rating due to switching and settling times.

Number of Coefficients

The functional operation of a signal conditioner is defined by a polynomial equation (see Figure 3). Refer to the signal conditioner documentation for the polynomial coefficients defining the polynomial equation. The number of coefficients specified for the equation is manufacturer dependent. Specify the **Number of Coefficients** to be used in the polynomial equation in this text box.

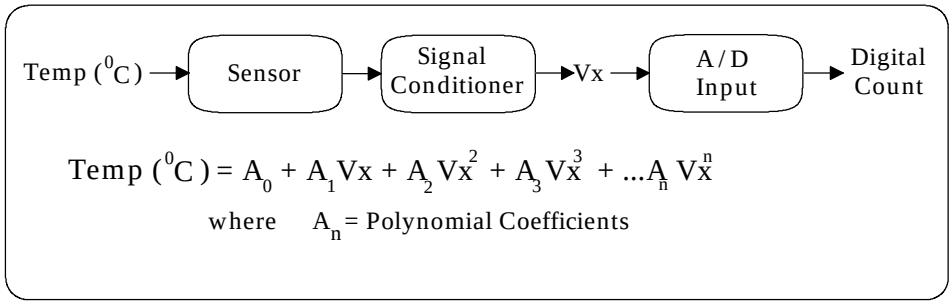



**Figure 3. The Polynomial Equation**

Polynomial Coefficients

Up to 12 polynomial coefficients in the polynomial equation for the signal conditioner may be specified. Fill in the **Number of Coefficients** text box to match the number of coefficient values in the **Polynomial Coefficients** list.

## 2.3 Creating DOS Applications Using The C Libraries

### 2.3.1 Microsoft Visual C/C++

To generate application programs using Microsoft Visual C/C++, the applications must be linked to one of the following DAQDRIVE libraries AND one or more hardware dependent libraries. These libraries MUST match the memory model selected for the application program. The DAQDRIVE installation program installs the following files into the DAQDRIVE\C_LIBS directory:

DAQDRVCS.LIB - small model DAQDRIVE library
DAQDRVCM.LIB - medium model DAQDRIVE library
DAQDRVCC.LIB - compact model DAQDRIVE library
DAQDRVCL.LIB - large model DAQDRIVE library

Three additional files are installed in the DAQDRIVE\C_LIBS directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQDRIVE.H - procedure prototypes
DAQOPENC.H - DaqOpenDevice definition for C
USERDATA.H - data structures and pre-defined constants

#### 2.3.1.1 The hardware dependent include file

The C library version of the DaqOpenDevice procedure is implemented as a macro using the "token-pasting" operator to create a unique open command for each hardware device. Application programs must include the file DAQOPENC.H and the hardware dependent include file defined in the target hardware's appendix. The DAQDRIVE installation program installs these files into the DAQDRIVE\C_LIBS directory.

#### 2.3.1.2 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using byte- aligned data structures.

To select byte aligned structures within the Visual C/C++ environment, first select **O**ptions, **P**roject, **C**ompiler, then set the structure member alignment field to 1 byte. For byte aligned structures from the Visual C/C++ command line, use the '/Zp1' option.

### 2.3.2 Borland C/C++

To generate application programs using Borland C/C++, the applications must be linked to one of the following DAQDRIVE libraries AND one or more hardware dependent libraries. These libraries MUST match the memory model selected for the application program. The DAQDRIVE installation program installs the following files into the DAQDRIVE\C_LIBS directory:

DAQDRVBS.LIB - small model DAQDRIVE library
DAQDRVBM.LIB - medium model DAQDRIVE library
DAQDRVBC.LIB - compact model DAQDRIVE library
DAQDRVBL.LIB - large model DAQDRIVE library

Three additional files are installed in the DAQDRIVE\C_LIBS directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQDRIVE. H - procedure prototypes
DAQOPENC.H - DaqOpenDevice definition for C
USERDATA.H - data structures and pre-defined constants

#### 2.3.2.1 The hardware dependent include file

The C library version of the DaqOpenDevice procedure is implemented as a macro using the "token-pasting" operator to create a unique open command for each hardware device. Application programs must include the file DAQOPENC.H and the hardware dependent include file defined in the target hardware's appendix. The DAQDRIVE installation program installs these files into the DAQDRIVE\C_LIBS directory.

#### 2.3.2.2 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using byte- aligned data structures.

To guarantee structures are byte aligned within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **C**ode Generation, then confirm the **W**ord alignment box is not checked. For byte aligned structures from the Borland C/C++ command line, use the '-a-' option.

2.3.2.3 Program optimization

When selecting the optimization options for the Borland C/C++ compiler, problems may arise if the 'Invariant code motion' option is selected and DAQDRIVE is operated in one of the background modes (IRQ or DMA). To disable the 'Invariant code motion' optimization within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **O**ptimizations, then confirm the '**I**nvariant code motion' box is not checked. From the Borland C/C++ command line, make sure the '-Om' and '-O2' options are not used.

**IMPORTANT:**

It is strongly recommended that the 'Invariant code motion' optimization option be disabled when using the Borland C/C++ compiler.

## 2.4 Creating DOS Applications Using The TSR Drivers

DAQDRIVE provides a TSR (Terminate-and-Stay-Resident) driver for creating DOS applications in any language that supports software interrupt (int) calls. In addition, libraries are provided to interface the following high-level languages to the DAQDRIVE TSR: Visual Basic for DOS, Quick Basic version 4.5, Turbo Pascal version 7.0 and newer, and most C compilers.

Although the interface to each of these languages is similar, the methods for generating application programs varies with the application language. The following sections describe the steps required to load the TSRs into memory and generate an application in each of the supported languages.

### 2.4.1 Loading The TSRs Into Memory

The DAQDRIVE installation program installs the TSR driver into the DAQDRIVE\TSR directory: The first step in creating applications which use the DAQDRIVE TSR is to load the driver into memory using the command line:

DAQDRIVE

In this mode, DAQDRIVE searches software interrupts 60H through 64H for an available interrupt. If an unused interrupt is located, DAQDRIVE takes control of this interrupt and displays a message indicating the installation was successful and which software interrupt is being used. If there are no available interrupts in this range, an error message is displayed and the DAQDRIVE TSR is not installed.

If the user wants to control the software interrupt number, or if all of the software interrupts between 60H and 64H are used, the user may specify a software interrupt with the following command line:

DAQDRIVE [/I=*interrupt*]

where *interrupt* specifies the software interrupt number in hexadecimal format. If the user-specified interrupt is not available, an error message is displayed and the DAQDRIVE TSR is not installed.

**Examples**:

DAQDRIVE /I=63 installs DAQDRIVE on interrupt 63H
DAQDRIVE /I=4F installs DAQDRIVE on interrupt 4FH
DAQDRIVE /I=b9 installs DAQDRIVE on interrupt B9H

After the DAQDRIVE TSR has been loaded, the user must load one or more TSRs for the hardware device(s) to be accessed. The DAQDRIVE installation program installs the TSR driver(s) for the selected hardware device(s) into the DAQDRIVE\TSR directory. For this discussion, we will assume the hardware driver's TSR name is HARDWARE.EXE. To load this TSR simply execute the command:

HARDWARE

The hardware TSR will search for the DAQDRIVE TSR in memory and, if it is located, will install itself using the same software interrupt. If DAQDRIVE was not previously installed, the hardware TSR will respond with an error message and will not be installed.

Multiple TSR drivers may be installed for multiple devices by repeating the above process for each hardware driver.

### 2.4.2 Removing The TSRs From Memory

The DAQDRIVE and hardware device TSR(s) may be removed from memory using the '/R' option to make additional memory available to other applications. The only restriction is that the TSRs must be removed in the reverse order of their installation. Consider an example where the following TSRs have been loaded:

DAQDRIVE - installs the DAQDRIVE TSR
DAQPTSR - installs the DAQP-208 TSR
IOP-241 - installs the IOP-241 TSR

To remove these TSRs from memory, the user simply reverses the installation order and adds the '/R' option to each command line:

IOP-241 /R - removes the IOP-241 TSR
DAQPTSR /R - removes the DAQP-208 TSR
DAQDRIVE /R - removes the DAQDRIVE TSR

### 2.4.3 Microsoft C/C++

To generate application programs using the DAQDRIVE TSR with Microsoft C/C++, the application must be linked with the DAQDRIVE library DAQTSRC.LIB installed in the DAQDRIVE\TSR\C directory by the DAQDRIVE installation program. This library is model independent and should work with most C compilers for DOS.

Three additional files are installed in the DAQDRIVE\TSR\C directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQDRIVE. H - procedure prototypes
DAQOPENT.H - DaqOpenDevice prototype
USERDATA.H - data structures and pre-defined constants

#### 2.4.3.1 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

**IMPORTANT:**
For proper operation, all application programs must be compiled using byte- aligned data structures.

To define byte aligned structures with Microsoft C use the '/Zp1' command line option. Within the Microsoft Visual C/C++ environment, select **O**ptions, **P**roject, **C**ompiler and set the structure member alignment field to 1 byte.

### 2.4.4 Borland C/C++ and Turbo C

To generate application programs using the DAQDRIVE TSR with Borland C/C++ or Turbo C, the application must be linked with the DAQDRIVE library DAQTSRC.LIB installed in the DAQDRIVE\TSR\C directory by the DAQDRIVE installation program. This library is model independent and should work with most C compilers for DOS.

Three additional files are installed in the DAQDRIVE\TSR\C directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQDRIVE. H - procedure prototypes
DAQOPENT.H - DaqOpenDevice prototype
USERDATA.H - data structures and pre-defined constants

#### 2.4.4.1 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using byte- aligned data structures.

To guarantee structures are byte aligned within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **C**ode Generation, then confirm the **W**ord alignment box is not checked. Within the Turbo C environment, select **O**ptions, **C**ompiler, **C**ode Generation, then set the **A**lignment option to byte. For byte aligned structures from the Borland C/C++ or Turbo C command lines, use the '-a-' option.

#### 2.4.4.2 Program optimization

When selecting the optimization options for the Borland C/C++ compiler, problems may arise if the 'Invariant code motion' option is selected and DAQDRIVE is operated in one of the background modes (IRQ or DMA). To disable the 'Invariant code motion' optimization within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **O**ptimizations, then confirm the '**I**nvariant code motion' box is not checked. From the Borland C/C++ command line, make sure the '-Om' and '-O2' options are not used.

> **IMPORTANT:**
> It is strongly recommended that the 'Invariant code motion' optimization option be disabled when using the Borland C/C++ compiler.

### 2.4.5 Quick Basic

To generate application programs using the DAQDRIVE TSR with Quick Basic 4.5, the Quick Library DAQQB45.QLB must be loaded from the Quick Basic command line using the /L option. DAQQB45.QLB is installed into the DAQDRIVE\TSR\QB45 directory by the DAQDRIVE installation program. A standard library, DAQQB45.LIB, is also installed in this directory for creating stand-alone executable programs (.EXE) using Quick Basic.

Two additional files are installed into the DAQDRIVE\TSR\QB45 directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQQB45.INC - procedure declarations
USERDATA.INC - data structures and pre-defined constants

#### 2.4.5.1 Quick Basic's on-line help

Although undocumented, the Quick Basic 4.5 on-line help appears to use software interrupt 60H and may interfere with the DAQDRIVE TSR. If suspicious errors occur while using Quick Basic, users are advised to install DAQDRIVE on software interrupt 61H, 62H, 63H, or 64H.

**IMPORTANT:**
Although undocumented, the Quick Basic 4.5 on-line help appears to use software interrupt 60H and may interfere with DAQDRIVE.

#### 2.4.5.2 Quick Basic and the under-score character

One difference between the Quick Basic version and other versions of DAQDRIVE is that Quick Basic reserves the underscore character ( _ ). The underscore character appears in data declarations and constants throughout this document but has been removed from the Quick Basic version of DAQDRIVE.

#### 2.4.5.3 Adjusting the size of Quick Basic's stack and heap

DAQDRIVE uses the application program's stack for storing local variables and for passing variables between DAQDRIVE procedures. By default, Quick Basic 4.5 only allocates 2K of memory for the application's stack which may be insufficient under come circumstances. It is recommended that the user increase the size of the application's stack by at least 2K using the CLEAR command. Note that the CLEAR command also clears all data memory and should therefore be used at the beginning of the application program.

In addition, application programs written using Quick Basic 4.5 allocate all available DOS memory for use as a local heap. This causes DAQDRIVE to report an error 300 (memory allocation error) when the application attempts to open a device. The application must reduce

the size of the heap using Quick Basic's SETMEM function before executing DaqOpenDevice. As a guide, the application should reduce the heap by 10,000 bytes for each hardware device opened and once the DaqOpenDevice procedure has been executed, the allocated heap space must not be returned to Quick Basic until the DaqCloseDevice procedure has been completed.

```
'*****************************************************************************
'  Increase the size of the Quick Basic stack by 2K
'*****************************************************************************

CLEAR ,,2048

'*****************************************************************************
'  Decrease the size of the heap so DAQDRIVE can allocate required memory
'*****************************************************************************

HeapSize = SETMEM(-10000)

'*****************************************************************************
'  Perform all DAQDRIVE functions
'*****************************************************************************

DaqOpenDevice ....


DaqCloseDevice ....

'*****************************************************************************
'  Optionally restore heap
'*****************************************************************************

HeapSize = SETMEM(+10000)
```

2.4.5.4 The DaqOpenDevice Command

DAQDRIVE's DaqOpenDevice command requires two null-terminated string variables: DeviceType and ConfigFile. Because Quick Basic does not support null-terminated strings, the user must create these strings by appending a null character, CHR$(0), to the end of each string before passing it to DAQDRIVE. For example:

```
A$ = "FILENAME.DAT"                normal Quick Basic string
B$ = "FILENAME.DAT" + CHR$(0)      null-terminated string
```

Furthermore, Quick Basic is unable to pass the address of a string variable as a far pointer. To overcome this problem, the DaqOpenDevice procedure is declared differently for the Quick Basic version of DAQDRIVE:

```
DaqOpenDevice ( BYVAL  TSRNumber          AS Integer,
                SEG    LogicalDevice      AS Integer,
                BYVAL  DeviceTypeSegment  AS Integer,
                BYVAL  DeviceTypeOffset   AS Integer,
                BYVAL  ConfigFileSegment  AS Integer,
                BYVAL  ConfigFileOffset   AS Integer )
```

The string variables normally found in the DaqOpenDevice command have been replaced by integer values which contain the segment and offset address of the string. The application program can obtain these addresses using the VARSEG and SADD functions as shown in the following example.

```
'*************************************************************************
'  Step 1: Open the device
'*************************************************************************

LogicalDevice% = 0
DeviceType$ = "DA8P-12B" + CHR$(0)
ConfigFile$ = "da8p-12b.dat" + CHR$(0)
Status% = DaqOpenDevice(&HF006, LogicalDevice%,
                        VARSEG(DeviceType$), SADD(DeviceType$),
                        VARSEG(ConfigFile$), SADD(ConfigFile$))
```

2.4.5.5 Storing a variable's address in a data structure

Another short-coming of Quick Basic is its inability to easily operate on a variable's address. Because of this limitation, all of the variables declared as 'far pointers' in the DAQDRIVE data structures have been divided into two integer values: a segment address and an offset address. An example of this is the channel array variable in the ADCRequest structure

```
unsigned short far *channel_array_ptr;
```

which becomes

```
ChannelArrayPtrOffset     AS  Integer
ChannelArrayPtrSegment  AS  Integer
```

For an array named Channel, the application fills in the array's address using the VARSEG and VARPTR procedures as follows:

```
DIM Channel[10] AS Integer

ADCRequest.ChannelArrayPtrOffset     = VARPTR(Channel[0])
ADCRequest.ChannelArrayPtrSegment  = VARSEG(Channel[0])
```

2.4.5.6 Dynamic memory allocation

To prevent Quick Basic from dynamically relocating variables, it is good practice to declare all variables before the first instruction of the application program.

### 2.4.6 Visual Basic for DOS

To generate application programs using the DAQDRIVE TSR with Visual Basic for DOS, the Quick Library DAQVBDOS.QLB must be loaded from the Visual Basic command line using the /L option. DAQVBDOS.QLB is installed into the DAQDRIVE\TSR\VBDOS directory by the DAQDRIVE installation program. A standard object library, DAQVBDOS.LIB, is also installed in this directory for creating executable programs (.EXE) using Visual Basic for DOS.

Two additional files are installed into the DAQDRIVE\TSR\VBDOS directory for the programmer's convenience. These files contain the prototypes of all the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

DAQVBDOS.INC - procedure declarations
USERDATA.INC - data structures and pre-defined constants

#### 2.4.6.1 Visual Basic for DOS and the under-score character

One difference between the Visual Basic for DOS version and other versions of DAQDRIVE is that Visual Basic reserves the underscore character ( _ ). The underscore character appears in data declarations and constants throughout this document but has been removed from the Visual Basic for DOS version of DAQDRIVE.

#### 2.4.6.2 Adjusting the size of the Visual Basic's stack and heap

DAQDRIVE uses the application program's stack for storing local variables and for passing variables between DAQDRIVE procedures. By default, Visual Basic for DOS only allocates 2K of memory for the application's stack which may be insufficient under come circumstances. It is recommended that the user increase the size of the application's stack by at least 2K using the CLEAR command. Note that the CLEAR command also clears all data memory and should therefore be used at the beginning of the application program.

In addition, application programs written using Visual Basic for DOS allocate all available DOS memory for use as a local heap. This causes DAQDRIVE to report an error 300 (memory allocation error) when the application attempts to open a device. The application program must reduce the size of the heap using Visual Basic's SETMEM function before executing DaqOpenDevice. As a guide, the application should reduce the heap by 10,000 bytes for each hardware device to be opened and once the DaqOpenDevice procedure has been executed, the allocated heap space must not be returned to Visual Basic until the DaqCloseDevice procedure has been completed.

```
'*************************************************************************
'  Increase the size of the Visual Basic stack by 2K
'*************************************************************************

CLEAR ,,2048

'*************************************************************************
'  Decrease the size of the heap so DAQDRIVE can allocate required memory
'*************************************************************************

HeapSize = SETMEM(-10000)

'*************************************************************************
'  Perform all DAQDRIVE functions
'*************************************************************************

DaqOpenDevice ....


DaqCloseDevice ....

'*************************************************************************
'  Optionally restore heap
'*************************************************************************

HeapSize = SETMEM(+10000)
```

2.4.6.3 The DaqOpenDevice Command

DAQDRIVE's DaqOpenDevice command requires two null-terminated string variables: DeviceType and ConfigFile. Because Visual Basic does not support null-terminated strings, the user must create these strings by appending a null character, CHR$(0), to the end of each string before passing it to DAQDRIVE. For example:

```
A$ = "FILENAME.DAT"                normal Visual Basic string
B$ = "FILENAME.DAT" + CHR$(0)      null-terminated string
```

Furthermore, Visual Basic for DOS is unable to pass the address of a string variable as a far pointer. To overcome this problem, the DaqOpenDevice procedure is declared differently for the Visual Basic for DOS version of DAQDRIVE:

```
DaqOpenDevice ( BYVAL  TSRNumber     AS Integer,
                SEG    LogicalDevice AS Integer,
                BYVAL  DeviceTypePtr AS Long,
                BYVAL  ConfigFilePtr AS Long )
```

The string variables normally found in the DaqOpenDevice command have been replaced by long integer values which contain the string's address. The application program can obtain the string address using the SSEGADD function as shown in the following example.

```
'*************************************************************************
'  Step 1: Open the device
'*************************************************************************

LogicalDevice% = 0
DeviceType$ = "DA8P-12B" + CHR$(0)
ConfigFile$ = "da8p-12b.dat" + CHR$(0)
Status% = DaqOpenDevice(&HF006, LogicalDevice%,
                        SSEGADD(DeviceType$),
                        SSEGADD(ConfigFile$))
```

2.4.6.4 Storing a variable's address in a data structure

Another short-coming of Visual Basic for DOS is its inability to easily operate on a variable's address. Because of this limitation, all of the variables declared as 'far pointers' in the DAQDRIVE data structures have been divided into two integer values: a segment address and an offset address. An example of this is the channel array variable in the ADCRequest structure

```
unsigned short far *channel_array_ptr;
```

which becomes

```
ChannelArrayPtrOffset     AS  Integer
ChannelArrayPtrSegment  AS  Integer
```

For an array named Channel, the application fills in the array's address using the VARSEG and VARPTR procedures as follows:

```
DIM Channel[10] AS Integer

ADCRequest.ChannelArrayPtrOffset     = VARPTR(Channel[0])
ADCRequest.ChannelArrayPtrSegment  = VARSEG(Channel[0])
```

2.4.6.5 Dynamic memory allocation

To prevent Visual Basic for DOS from dynamically relocating variables, it is good practice to declare all variables before the first instruction of the application program.

### 2.4.7 Turbo Pascal

DAQDRIVE supports applications written with Turbo Pascal version 7.0 and newer through the unit files DAQDRIVE.TPU and DAQDATA.TPU installed by the DAQDRIVE installation program into the DAQDRIVE\TSR\PASCAL directory. DAQDRIVE.TPU defines the Turbo Pascal interface to the DAQDRIVE functions while DAQDATA.TPU defines the data structures (Pascal 'records') and constants mentioned throughout this document.

In order to access DAQDRIVE's functions, data structures, and constants, all application programs must include the following statement:

```
USES DAQDRIVE, DAQDATA;
```

#### 2.4.7.1 Turbo Pascal and floating-point math

The Turbo Pascal floating-point emulation library only supports variables of type 'real'. To use the single and double precision variables required by DAQDRIVE, the 8087 floating-point math mode must be enabled by selecting **O**ptions, **C**ompiler, **8**087/80287 or by defining the numeric coprocessor switch {$N+}.

#### 2.4.7.2 Adjusting the size of the Turbo Pascal heap

By default, application programs written using Turbo Pascal allocate all available DOS memory for use as a local heap. This causes DAQDRIVE to report an error 300 (memory allocation error) when the application attempts to open a device. The user must reduce the size of the application's heap by selecting **O**ptions, **M**emory sizes and setting the 'High heap limit' option to a value less than 655,360 (640K). As a guide, reduce the heap by 10,000 bytes for each hardware device to be opened by the application.

> **IMPORTANT:**
> The user must modify the default Turbo Pascal heap settings to prevent the application from allocating all available DOS memory at start-up.

#### 2.4.7.3 Using other Turbo Pascal versions

When using a version of Turbo Pascal other than 7.0, the user must create new unit files (.TPUs) by re-compiling the source files DAQDRIVE.PAS and DAQDATA.PAS. These files, along with the interface library DAQTSR.OBJ, are also installed into the DAQDRIVE\TSR\PASCAL directory by the DAQDRIVE installation program.

## 2.5 Creating 16-bit Windows 3.x/95/98 Applications

DAQDRIVE supports 16-bit Windows application programs written in most languages which support the Windows DLL (Dynamic Link Library) interface. When the Windows application programs are executed, they must be able to dynamically link to DAQDRIVE.DLL and one or more hardware dependent DLLs. Windows searches for any necessary DLLs in the following locations:

1. the current directory
2. the Windows directory (directory containing WIN.COM)
3. the Windows\System directory (directory containing GDI.EXE)
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program installs the DAQDRIVE DLL and the DLLs for any selected hardware device into the Windows\System directory. In addition, an import library, DAQDRIVE.LIB, is installed into the DAQDRIVE\WINDLL directory. This import library can be used by many Windows compilers to simplify the linking of application programs to the APIs available within the DAQDRIVE DLL.

DAQDRIVE has been tested with application programs written in Microsoft Visual C/C++, Borland C/C++, Turbo Pascal for Windows, and Borland Delphi. The following sections provide additional information about producing Windows applications in the languages above.

### 2.5.1 Microsoft Visual C/C++

To generate application programs using the DAQDRIVE DLL with Microsoft Visual C/C++, the application must be linked with the import library, DAQDRIVE.LIB, installed in the DAQDRIVE\WINDLL directory. This library is model independent and should work with most C compilers for Windows.

Three additional files are installed into the DAQDRIVE\WINDLL\C directory for the programmer's convenience. These files contain the prototypes of the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

| | | |
|---|---|---|
| DAQDRIVE. H | - | procedure prototypes |
| DAQOPENW.H | - | DaqOpenDevice definition for Windows |
| USERDATA.H | - | data structures and pre-defined constants |

#### 2.5.1.1 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using byte- aligned data structures.

To select byte aligned structures within the Microsoft Visual C/C++ environment, first select **O**ptions, **P**roject, **C**ompiler, then set the structure member alignment field to 1 byte. For byte aligned structures from the Visual C/C++ command line, use the '/Zp1' option.

### 2.5.2 Borland C/C++

To generate application programs using the DAQDRIVE DLL with Borland C/C++, the application must be linked with the import library DAQDRIVE.LIB installed in the DAQDRIVE\WINDLL directory. This library is model independent and should work with most C compilers for Windows.

Three additional files are installed into the DAQDRIVE\WINDLL\C directory for the programmer's convenience. These files contain the prototypes of the DAQDRIVE procedures, data structure definitions, and constants mentioned throughout this document. These files must be included in all application programs.

| | | |
|---|---|---|
| DAQDRIVE. H | - | procedure prototypes |
| DAQOPENW.H | - | DaqOpenDevice definition for Windows |
| USERDATA.H | - | data structures and pre-defined constants |

#### 2.5.2.1 Creating byte-aligned data structures

Because DAQDRIVE supports multiple languages, all data structures are byte-aligned (packed). The application program must also set structure packing to byte-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using byte- aligned data structures.

Borland C/C++ defines structures as byte aligned by default. To guarantee structures are byte aligned within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **C**ode Generation, then confirm the **W**ord alignment box is not checked. For byte aligned structures from the Borland C/C++ command line, use the '-a-' option.

#### 2.5.2.2 Program optimization

When selecting the optimization options for the Borland C/C++ compiler, problems may arise if the 'Invariant code motion' option is selected and DAQDRIVE is operated in one of the background modes (IRQ or DMA). To disable the 'Invariant code motion' optimization within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **O**ptimizations, then confirm the '**I**nvariant code motion' box is not checked. From the Borland C/C++ command line, make sure the '-Om' and '-O2' options are not used.

> **IMPORTANT:**
> It is strongly recommended that the 'Invariant code motion' optimization option be disabled when using the Borland C/C++ compiler.

### 2.5.3 Visual Basic for Windows

16-bit Visual Basic programming support is provided by the "VisualDAQ" and "VisualDAQ Light" software packages. VisualDAQ is a set of Visual Basic custom controls for Omega's data acquistion hardware. The VisualDAQ controls provide an easy interface to interact with Omega's data acquisition product line.

VisualDAQ Light, which is included free with the purchase of any data aquisition product, provides a simple interface to perform single point data aquisition I/O. On-line documentation is included with VisualDAQ Light.

VisualDAQ provides Visual Basic programmers with custom controls to configure all parameters of the data aquisition board. VisualDAQ is sold seperately and includes a complete programming reference manual.

### 2.5.4 Turbo Pascal for Windows / Borland Delphi

DAQDRIVE supports applications written with Turbo Pascal for Windows version 1.5 and newer through the unit files DAQDRVW.TPU and DAQDATA.TPU while Borland Delphi applications are supported with the unit files DAQDRVW.DCU and DAQDATA.DCU. The unit DAQDRVW defines the interface to the DAQDRIVE DLL functions while DAQDATA defines the data structures (Pascal 'records') and constants mentioned throughout this document. All of these files are installed into the DAQDRIVE\WINDLL\PASCAL directory by the DAQDRIVE installation program.

In order to access DAQDRIVE's functions, data structures, and constants, all Turbo Pascal for Windows and Borland Delphi application programs must include the following statement:

```
USES DAQDRVW, DAQDATA;
```

#### 2.5.4.1 Using other Turbo Pascal for Windows / Delphi versions

When using versions other than Turbo Pascal for Windows 1.5 or Borland Delphi 1.0, the user must create new unit files by re-compiling the source files DAQDRVW.PAS and DAQDATA.PAS. These files are also installed into the DAQDRIVE\WINDLL\PASCAL directory by the DAQDRIVE installation program.

#### 2.5.4.2 Turbo Pascal for Windows and floating-point math

The Turbo Pascal for Windows floating-point emulation library only supports variables of type 'real'. To use the single and double precision variables required by DAQDRIVE, the 8087 floating-point math mode must be enabled by selecting **O**ptions, **C**ompiler, 08x8**7** code or by defining the numeric coprocessor switch {$N+}.

## 2.6 Creating 32-bit Windows 95/98 Applications

DAQDRIVE supports 32-bit Windows 95/98 application programs written in most languages which support the Windows DLL (Dynamic Link Library) interface. When the Windows application programs are executed, they must be able to dynamically link to DDRIVE32.DLL, DDRIVE32.VXD, and one or more hardware dependent DLLs and VxDs. Windows searches for any necessary DLLs and VxDs in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program installs DDRIVE32.DLL, DDRIVE32.VXD and the DLLs and VxDs for any selected hardware devices into the Windows\System directory.

DAQDRIVE has been tested with application programs written in Microsoft Visual C/C++, Borland C/C++, and Microsoft Visual Basic. The following sections provide additional information about producing Windows applications in the languages above.

### 2.6.1 Microsoft Visual C/C++

To generate 32-bit application programs using the DAQDRIVE DLL with Microsoft Visual C/C++, the application must be linked with the import library, DDRIVE32.LIB, installed in the DAQDRIVE\WIN32 directory by the DAQDRIVE installation program.

Three additional files are installed into the DAQDRIVE\WIN32\C directory for the programmer's convenience. These files contain the prototypes of the DAQDRIVE procedures, the DAQDRIVE data structure definitions, and the DAQDRIVE constants mentioned throughout this document. These files must be included in all application programs.

| | | |
|---|---|---|
| DAQDRIVE.H | - | procedure prototypes |
| DAQOPENW.H | - | DaqOpenDevice definition for Windows |
| USERDATA.H | - | data structures and pre-defined constants |

#### 2.6.1.1 Creating dword-aligned data structures

Unlike the 16-bit versions of DAQDRIVE, the 32-bit DAQDRIVE data structures are dword-aligned (4-byte alignment). The application program must also set structure packing to dword-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using dword- aligned (4-byte aligned) data structures.

To select dword-aligned structures within the Microsoft Visual C/C++ environment, first select **O**ptions, **P**roject, **C**ompiler, then set the structure member alignment field to 4 bytes. For byte aligned structures from the Visual C/C++ command line, use the '/Zp4' option.

### 2.6.2 Borland C/C++

To generate 32-bit application programs using the DAQDRIVE DLL with Borland C/C++, the application must be linked with the DAQDRIVE import library DDRV32BC.LIB installed in the DAQDRIVE\WIN32 directory by the DAQDRIVE installation program.

Three additional files are installed into the DAQDRIVE\WIN32\C directory for the programmer's convenience. These files contain the prototypes of the DAQDRIVE procedures, the DAQDRIVE data structure definitions, and the DAQDRIVE constants mentioned throughout this document. These files must be included in all application programs.

| | | |
|---|---|---|
| DAQDRIVE.H | - | procedure prototypes |
| DAQOPENW.H | - | DaqOpenDevice definition for Windows |
| USERDATA.H | - | data structures and pre-defined constants |

#### 2.6.2.1 Creating dword-aligned data structures

Unlike the 16-bit versions of DAQDRIVE, the 32-bit DAQDRIVE data structures are dword-aligned (4-byte alignment). The application program must also set structure packing to dword-aligned for proper operation.

> **IMPORTANT:**
> For proper operation, all application programs must be compiled using dword- aligned (4-byte aligned) data structures.

Borland C/C++ defines structures as byte aligned by default. To guarantee structures are byte aligned within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **C**ode Generation, then confirm the **W**ord alignment box is not checked. For byte aligned structures from the Borland C/C++ command line, use the '-a-' option.

#### 2.6.2.2 Program optimization

When selecting the optimization options for the Borland C/C++ compiler, problems may arise if the 'Invariant code motion' option is selected and DAQDRIVE is operated in one of the background modes (IRQ or DMA). To disable the 'Invariant code motion' optimization within the Borland C/C++ environment, select **O**ptions, **C**ompiler, **O**ptimizations, then confirm the '**I**nvariant code motion' box is not checked. From the Borland C/C++ command line, make sure the '-Om' and '-O2' options are not used.

> **IMPORTANT:**
> It is strongly recommended that the 'Invariant code motion' optimization option be disabled when using the Borland C/C++ compiler.

### 2.6.3 32-bit Visual Basic

Because of Visual Basic's inability to lock memory and represent variables as pointers, a new DLL, DAQDRVVB.DLL, was created to simplify the interface between the application program and the standard DAQDRIVE drivers. This DLL is copied into the Windows\System directory whenever 32-bit Visual Basic support is selected in the DAQDRIVE setup program.

Two additional files are installed into the DAQDRIVE\WIN32\VB directory for the programmer's convenience. These files contain the prototypes of the DAQDRIVE procedures, the DAQDRIVE data structure definitions, and the DAQDRIVE constants mentioned throughout this document. These files must be included in all application programs.

DAQDRVB.BAS - procedure prototypes
USERDATA.BAS - data structures and pre-defined constants

**Differences between Visual Basic and other supported languages**

Although the functionality of the DAQDRIVE APIs has been maintained for Visual Basic, some of the implementations had to be changed. To minimize the impact on Visual Basic programmers, **all** of the DAQDRIVE APIs have been modified to append a 'VB' extension onto the function name. For example, DaqOpenDevice becomes DaqOpenDeviceVB, DaqAllocateRequest becomes DaqAllocateRequestVB, and DaqAnalogInput becomes DaqAnalogInputVB.

**DAQDRIVE data buffers, structures, and locked memory**

Because Visual Basic cannot lock memory, application programs **MUST** use the DaqAllocateRequestVB function to allocate the required request and data structures. Furthermore, since Visual Basic cannot directly access the locked memory allocated by DaqAllocateRequestVB, four additional APIs have been added to the Visual Basic version of DAQDRIVE:

DaqReadBufferVB - Copy data from a locked DAQDRIVE data buffer to a Visual Basic array.
DaqReadBufferFlagVB - Check the current status of a DAQDRIVE data buffer
DaqWriteBufferVB - Copy data from a Visual Basic array to a locked DAQDRIVE data buffer.
DaqWriteBufferFlagVB - Set the current status of a DAQDRIVE data buffer

Each of these functions is discussed in the following sections.

Visual Basic programmer are encouraged to read and understand the use and operation of DAQDRIVE data buffers and data buffer structures as discussed in chapter 9 and the DAQDRIVE event processes as discussed in chapter 11.

2.6.3.1 DaqReadBufferVB

DaqReadBufferVB is a DAQDRIVE Visual Basic utility function used to transfer data from a DAQDRIVE input data buffer to a Visual Basic array. DaqReadBufferVB will copy the data from the DAQDRIVE buffer specified by buffer_number to the Visual Basic array specified by memory_pointer. DaqReadBufferVB will also reset the associated DAQDRIVE_buffer structure's buffer_status field to BUFFER_EMPTY to prepare it for the next acquisition.

```
unsigned short DaqReadBufferVB(unsigned short request_handle,
                               unsigned short buffer_number,
                               void *memory_pointer)
```

request_handle - This unsigned short integer variable is used to specify the request containing the target data buffer. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

buffer_number - This unsigned short integer variable is used to specify which DAQDRIVE data buffer is to be returned to the application program.

memory_pointer - This void pointer defines the address of a Visual Basic data array where the input data will be copied. memory_pointer is declared as type void to allow it to point to data of any type

2.6.3.2 DaqReadBufferFlagVB

DaqReadBufferFlagVB is a DAQDRIVE Visual Basic utility function used to inspect the current status of the DAQDRIVE data buffer flags. DaqReadBufferFlagVB will copy the buffer_status field of the DAQDRIVE_buffer structure specified by buffer_number to the Visual Basic variable specified by buffer_status.

```
unsigned short DaqReadBufferFlagVB(unsigned short  request_handle,
                                   unsigned short  buffer_number,
                                   unsigned short *buffer_status)
```

request_handle - This unsigned short integer variable is used to specify the request containing the target data buffer. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

buffer_number - This unsigned short integer variable is used to specify which DAQDRIVE data buffer's status is to be returned to the application program.

buffer_status - This pointer defines the address of an unsigned short integer value where the DAQDRIVE_buffer structure's buffer_status field will be stored.

```
'*****  Open the DAQP-12(see DaqOpenDevice).  *****

'***** Allocate and lock memory for the Analog Input.  *****

AllocateRequest.request_type         = ADC_TYPE_REQUEST
AllocateRequest.channel_array_length = 1
AllocateRequest.number_of_buffers    = 4
AllocateRequest.buffer_length        = 1000
AllocateRequest.buffer_Attributes    = SEQUENTIAL_BUFFER

gStatus = DaqAllocateRequestVB(iLogicalDevice, AllocateRequest)
If gStatus <> 0 Then GoTo ErrorExit

'*****  Request A/D input (See DaqAnalogInput).  *****

'*****  Arm the request.  *****

gStatus = Call DaqArmRequestVB(iRequestHandle)
If gStatus <> 0 Then Goto ErrorExit

'*****  Trigger the request.  *****

gStatus = Call DaqTriggerRequestVB(iRequestHandle)
If gStatus <> 0 Then Goto ErrorExit

'*****  Wait for completion or error.  *****

lEventMask = CompleteEvent OR RuntimeErrorEvent
while(ADCUserRequest.RequestStatus AND lEventMask) = 0

   '*****  Check for buffer full event.  *****

   If(ADCUserRequest.RequestStatus AND BufferFullEvent) <> 0 Then

      '*****  Loop to find all full buffers.  *****

      For iIndex = 0 to 3
         gStatus = Call DaqReadBufferFlagVB(iRequestHandle, iIndex, iBufferStatus)

      '*****  If buffer full, copy to local array.  *****

         If(gStatus = 0) AND (iBufferStatus = BufferFull) Then
            gStatus = Call DaqReadBufferVB(iRequestHandle, iIndex, iADCData(iIndex))
         End If
      Next iIndex
   End If
Wend

'*****  Check for run-time errors.  *****

If(ADCUserRequest.RequestStatus AND RuntimeErrorEvent) <> 0 Then
   gStatus = Call DaqGetRuntimeErrorVB(iRequestHandle, iRuntimeErrorCode)
   Goto RuntimeErrorExit
End If

'*****  Release the request.  *****

'*****  Close the device.  *****
```

2.6.3.3 DaqWriteBufferVB

DaqWriteBufferVB is a DAQDRIVE Visual Basic utility function used to transfer data from a Visual Basic array to a DAQDRIVE output data buffer. DaqWriteBufferVB copies the data from the Visual Basic array specified by memory_pointer to the DAQDRIVE buffer specified by buffer_number, sets the associated DAQDRIVE_buffer structure's buffer_cycles field to the value specified by buffer_cycles, and initializes the DAQDRIVE_buffer structure's buffer_status field to BUFFER_FULL to prepare it for the pending operation.

```
unsigned short DaqWriteBufferVB(unsigned short request_handle,
                                unsigned short buffer_number,
                                unsigned long  buffer_cycles,
                                void *memory_pointer)
```

request_handle - This unsigned short integer variable is used to specify the request containing the target data buffer. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

buffer_number - This unsigned short integer variable is used to specify which DAQDRIVE data buffer is to be written.

buffer_cycles - This unsigned long integer value is placed in the buffer_cycles field of the DAQDRIVE buffer structure and specifies the number of times the data in this structure is processed before continuing on to the next_structure. See Chapter 9 for further details.

memory_pointer - This void pointer defines the address of a Visual Basic array containing the data to be copied to the DAQDRIVE data buffer. memory_pointer is declared as type void to allow it to point to data of any type.

2.6.3.4 DaqWriteBufferFlagVB

DaqWriteBufferFlagVB is a DAQDRIVE Visual Basic utility function used to set the status of a DAQDRIVE data buffer's flags. DaqWriteBufferFlagVB will set the buffer_status field of the DAQDRIVE_buffer structure specified by buffer_number to the value of the Visual Basic variable specified by buffer_status. Generally, DaqWriteBufferFlagVB is only required to 'clean-up' the buffer_status flags after a run-time error has occurred.

```
unsigned short DaqWriteBufferFlagVB(unsigned short request_handle,
                                    unsigned short buffer_number,
                                    unsigned short buffer_status)
```

request_handle - This unsigned short integer variable is used to specify the request containing the buffer which needs modified. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

buffer_number - This unsigned short integer variable is used to specify which DAQDRIVE data buffer's status is to be modified.

buffer_status - This unsigned short integer variable specifies the new value for the DAQDRIVE_buffer structure's buffer_status field.

```
'*****  Open the DA8P-12(see DaqOpenDevice).  *****

'***** Allocate and lock memory for the Analog Output.  *****

AllocateRequest.request_type         = DAC_TYPE_REQUEST
AllocateRequest.channel_array_length = 1
AllocateRequest.number_of_buffers    = 1
AllocateRequest.buffer_length        = 100
AllocateRequest.buffer_Attributes    = SEQUENTIAL_BUFFER

gStatus = DaqAllocateRequestVB(iLogicalDevice, AllocateRequest)
If gStatus <> 0 Then GoTo ErrorExit

'***** Prepare the D/a request structure.  *****
DacUserRequest.ChannelArrayPtr = DaqGetAddressOfVB(channel)
DacUserRequest.ArrayLength     = 1
DacUserRequest.TriggerSource   = InternalTrigger
DacUserRequest.TriggerMode     = ContinuousTrigger
DacUserRequest.TriggerChannel  = 0
DacUserRequest.TriggerVoltage  = 0
DacUserRequest.IOMode          = BackgroundIRQ
DacUserRequest.ClockSource     = InternalClock
DacUserRequest.SampleRate      = 1000
DacUserRequest.NumberOfScans   = 1
DacUserRequest.ScanEventLevel  = 0
DacUserRequest.Calibration     = NoCalibration
DacUserRequest.TimeoutInterval = 0
DacUserRequest.RequestStatus   = 0

'***** Send request to DAQDRIVE for analog input  *****
gStatus = DaqAnalogOutputVB(iLogicalDevice, DacUserRequest, iRequestHandle)
If gStatus <> 0 Then GoTo ErrorExit

'***** Copy output data to the daqdrive allocated buffer.  *****
gStatus = DaqWriteBufferVB(iRequestHandle, 0, 1, Outputdata(0))
If gStatus <> 0 Then GoTo ErrorExit
```

# 3 Quick Start Procedures

DAQDRIVE's "data defined" interface may be considerably different from "normal" data acquisition drivers and for simple operations may seem to result in more work for the application programmer. For this reason, DAQDRIVE provides a set of procedures to perform simple operations in the "normal" way. These procedures act as DAQDRIVE macros, configuring all of the necessary data structures and executing all of the routines required to complete the pre-defined function. To use any of these functions, the application need only follow the steps listed below.

**Step 1: Define The Hardware Configuration**

DAQDRIVE determines the configuration of a device from the data file specified when the device is opened. These configuration files are created using the DAQDRIVE configuration utility as described in section 2.2.

**Step 2: Open The Hardware Device**

Before the application program can use an adapter, it must first open the device using the DaqOpenDevice command. The application must provide the open command with the adapter type and specify the name of a configuration file (generated in step 1) which describes the target hardware's configuration. If the open command completes successfully, DAQDRIVE assigns a logical device number to be used for all future references to the adapter.

**Step 3: Execute The Quick-Start Procedure(s)**

These procedures are discussed on the following pages.

**Step 4: Close The Hardware Device**

When all operations on the hardware are complete, the device should be closed using DaqCloseDevice to free any resources used by that device. System integrity can not be guaranteed if the application program exits without closing the hardware device.

## 3.1 Analog Input

For analog input, DAQDRIVE provides two special purpose procedures: DaqSingleAnalogInput and DaqSingleAnalogInputScan. The intent of this section is to provide an overview of these procedures. For details on the implementation of the procedures, consult the alphabetical listing of commands in chapter 13.

### 3.1.1 DaqSingleAnalogInput

One of the simplest cases of analog input is to input a single sample from a single A/D channel under CPU control. DAQDRIVE provides a simplified interface for this operation through the DaqSingleAnalogInput procedure. The format of this command is shown below.

```
unsigned short DaqSingleAnalogInput(unsigned short logical_device,
                                    unsigned short channel_number,
                                    float gain_setting,
                                    void far *input_value)
```

DaqSingleAnalogInput sets the gain of the A/D channel specified by channel_number on the adapter specified by logical_device to the value specified by gain_setting. A single sample is then input from this analog channel and stored in the memory location specified by input_value. The following example shows the usage of DaqSingleAnalogInput.

```
/*** Input a single sample from A/D channel 0 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
short input_value;
char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input value from channel 0, gain of 1 ***/

status = DaqSingleAnalogInput(logical_device, 0, 1, &input_value);
if (status != 0)
   printf("\n\nA/D input error.  Status code %d.\n\n", status);
else
   printf("Channel 0:  %d\n\n", input_value);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleAnalogInput is a very basic interface without any allowance for multiple channels, multiple input values, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the analog input configuration procedure (DaqAnalogInput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's analog input interface, the following example program creates the equivalent of the DaqSingleAnalogInput procedure.

```
unsigned short MySingleAnalogInput(unsigned short logical_device,
                                   unsigned short channel_number,
                                   float gain_setting,
                                   void far *input_value)
{
struct ADC_request      my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.gain_array_ptr    = &gain_setting;
my_request.array_length      = 1;
my_request.ADC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = (void huge*)input_value;
my_data.buffer_length  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_EMPTY;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqAnalogInput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

### 3.1.2 DaqSingleAnalogInputScan

Another simple case of analog input is to input one value each from multiple A/D channels under CPU control. This allows multiple analog inputs to be sampled simultaneously (or nearly simultaneously depending on the data acquisition hardware). A simplified interface for this operation is provided through the DaqSingleAnalogInputScan procedure. The format of this command is shown below.

```
unsigned short DaqSingleAnalogInputScan(unsigned short logical_device,
                                        unsigned short far *channel_array,
                                        float far *gain_array,
                                        unsigned short array_length,
                                        void far *input_array)
```

DaqSingleAnalogInputScan inputs a single sample from each of the A/D channels specified by channel_array using the corresponding gain setting in the gain_array. The A/D channels are located on the adapter specified by logical_device and the samples are stored in the array specified by input_array. A one-to-one correspondence is required between the number of analog input channels, the gain settings, and the number of samples. Therefore, array_length specifies the length of channel_array, gain_array, and input_array. The following example shows the usage of DaqSingleAnalogInputScan.

```
/*** Input a single sample from A/D channels 0, 3, and 7 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short channel_array[3] = { 0, 3, 7 };
unsigned short gain_array[3]    = { 1, 1, 2 };
unsigned short status;
short input_array[3];
char far *device_type = "DAQP-208";
char far *config_file = "daqp-208.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input one sample from each channel ***/

status = DaqSingleAnalogInputScan(logical_device, channel_array,
                                  gain_array, 3, input_array);
if (status != 0)
   printf("\n\nA/D input error.  Status code %d.\n\n", status);
else
   {
   printf("Channel 0:  %d\n\n", input_array[0]);
   printf("Channel 3:  %d\n\n", input_array[1]);
   printf("Channel 7:  %d\n\n", input_array[2]);
   }

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleAnalogInputScan is a very basic interface without any allowance for timing information, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the analog input configuration procedure (DaqAnalogInput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's analog input interface, the following example program creates the equivalent of the DaqSingleAnalogInputScan procedure.

```
unsigned short MySingleAnalogInputScan(unsigned short logical_device,
                                       unsigned short far *channel_array,
                                       float far *gain_array,
                                       unsigned short array_length,
                                       void far *input_array)
{
struct ADC_request      my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_array;
my_request.gain_array_ptr    = gain_array;
my_request.array_length      = array_length;
my_request.ADC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = (void huge*)input_array;
my_data.buffer_length = array_length;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_EMPTY;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqAnalogInput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

## 3.2 Analog Output

For analog output, DAQDRIVE provides two special purpose procedures: DaqSingleAnalogOutput and DaqSingleAnalogOutputScan. The intent of this section is to provide an overview of these procedures. For details on the implementation of the procedures, consult the alphabetical listing of commands in chapter 13.

### 3.2.1 DaqSingleAnalogOutput

One of the simplest cases of analog output is to output a single value to a single D/A converter under CPU control. DAQDRIVE provides a simplified interface for this function through the DaqSingleAnalogOutput procedure. The format of this command is shown below.

```
unsigned short DaqSingleAnalogOutput(unsigned short logical_device,
                                     unsigned short channel_number,
                                     void far *output_value)
```

DaqSingleAnalogOutput outputs the value specified by output_value to the D/A converter specified by channel_number on the adapter specified by logical_device. The following example shows the usage of DaqSingleAnalogOutput.

```
/*** Output a single sample to D/A channel 1 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
short output_value;
char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Output the value to channel 1 ***/

output_value = 512;
status = DaqSingleAnalogOutput(logical_device, 1, &output_value);
if (status != 0)
   printf("\n\nD/A output error.  Status code %d.\n\n", status);
else
   printf("\n\nComplete.  No errors.);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleAnalogOutput is a very basic interface without any allowance for multiple channels, multiple output values, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the analog output configuration procedure (DaqAnalogOutput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's analog output interface, the following example program creates the equivalent of the DaqSingleAnalogOutput procedure.

```
unsigned short MySingleAnalogOutput(unsigned short logical_device,
                                    unsigned short channel_number,
                                    void far *output_value)
{
struct DAC_request      my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.DAC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = (void huge*)output_value;
my_data.buffer_length  = 1;
my_data.buffer_cycles  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_FULL;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqAnalogOutput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

### 3.2.2 DaqSingleAnalogOutputScan

Another simple case of analog output is to output one value each to multiple D/A converters under CPU control. This allows multiple analog outputs to be updated simultaneously (or nearly simultaneously depending on the data acquisition hardware). A simplified interface for this operation is provided through the DaqSingleAnalogOutputScan procedure. The format of this command is shown below.

```
unsigned short DaqSingleAnalogOutputScan(unsigned short logical_device,
                                         unsigned short far *channel_array,
                                         unsigned short array_length,
                                         void far *output_array)
```

DaqSingleAnalogOutputScan outputs the values in the array specified by output_array to the D/A converter channels in the array specified by channel_array on the adapter specified by logical_device. A D/A channel may appear in channel_array only once and a one-to-one correspondence is required between the number of D/A converter channels and the number of output values. Therefore, array_length specifies the length of both channel_array and output_array. The following example shows the usage of DaqSingleAnalogOutputScan.

```
/*** Output a single sample to D/A channels 1, 5, and 2 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned short channel_array[3] = {   1,    5,     2 };
short output_array[3]           = { 413, 3781, -1468 };
char far *device_type = "DA8P-12B";
char far *config_file = "da8p-12b.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Output the D/A values ***/

status = DaqSingleAnalogOutputScan(logical_device, channel_array,
                                   3, output_array);
if (status != 0)
   printf("\n\nD/A output error.  Status code %d.\n\n", status);
else
   printf("\n\nComplete.  No errors.);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleAnalogOutputScan is a very basic interface without any allowance for timing information, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the analog output configuration procedure (DaqAnalogOutput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's analog output interface, the following example program creates the equivalent of the DaqSingleAnalogOutputScan procedure.

```
unsigned short MySingleAnalogOutputScan(unsigned short logical_device,
                                        unsigned short far *channel_array,
                                        unsigned short array_length,
                                        void far *output_array)
{
struct DAC_request      my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle, status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_array;
my_request.array_length      = array_length;
my_request.DAC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = (void huge*)output_array;
my_data.buffer_length = array_length;
my_data.buffer_cycles = 1;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_FULL;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqAnalogOutput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

## 3.3 Digital Input

DAQDRIVE provides two special purpose procedures for digital input: DaqSingleDigitalInput and DaqSingleDigitalInputScan. The intent of this section is to provide an overview of these procedures. For details on the implementation of the procedures, consult the alphabetical listing of commands in chapter 13.

### 3.3.1 DaqSingleDigitalInput

One of the simplest cases of digital input is to input a single sample from a single digital I/O channel under CPU control. DAQDRIVE provides a simplified interface for this operation through the DaqSingleDigitalInput procedure. The format of this command is shown below.

```
unsigned short DaqSingleDigitalInput(unsigned short logical_device,
                                     unsigned short channel_number,
                                     void far *input_value)
```

DaqSingleDigitalInput inputs a single sample from the digital I/O specified by channel_number on the adapter specified by logical_device. The sample is stored in the memory location specified by input_value. The following example shows the usage of DaqSingleDigitalInput.

```
/*** Input a single sample from digital I/O channel 3 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned char input_value;
char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input one value from channel 3 ***/

status = DaqSingleDigitalInput(logical_device, 3, &input_value);
if (status != 0)
   printf("\n\nDigital input error.  Status code %d.\n\n", status);
else
   printf("Channel 3:  %d\n\n", (int)input_value);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleDigitalInput is a very basic interface without any allowance for multiple channels, multiple input values, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the digital input configuration procedure (DaqDigitalInput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's digital input interface, the following example program creates the equivalent of the DaqSingleDigitalInput procedure.

```
unsigned short MySingleDigitalInput(unsigned short logical_device,
                                    unsigned short channel_number,
                                    void far *input_value)
{
struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = (void huge*)input_value;
my_data.buffer_length  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_EMPTY;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqDigitalInput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

### 3.3.2 DaqSingleDigitalInputScan

Another simple case of digital input is to input one value each from multiple digital I/O channels under CPU control. This allows multiple digital inputs to be sampled simultaneously (or nearly simultaneously depending on the data acquisition hardware). A simplified interface for this operation is provided through the DaqSingleDigitalInputScan procedure. The format of this command is shown below.

```
unsigned short DaqSingleDigitalInputScan(unsigned short logical_device,
                                         unsigned short far *channel_array,
                                         unsigned short array_length,
                                         void far *input_array)
```

DaqSingleDigitalInputScan inputs a single sample from each of the digital I/O channels specified by channel_array on the adapter specified by logical_device. The samples are stored in the array specified by input_array. A one-to-one correspondence is required between the number of digital input channels and the number of samples. Therefore, array_length specifies the length of channel_array and input_array. The following example shows the usage of DaqSingleDigitalInputScan.

```
/*** Input a single sample from digital I/O channels 3, 2, and 1 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short channel_array[3] = { 3, 2, 1 };
unsigned short status;
unsigned char input_array[3];
char far *device_type = "IOP-241";
char far *config_file = "iop-241.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(IOP241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input one sample from each channel ***/

status = DaqSingleDigitalInputScan(logical_device, channel_array,
                                   3, input_array);
if (status != 0)
   printf("\n\nA/D input error.  Status code %d.\n\n", status);
else
   {
   printf("Channel 3:  %d\n\n", (int)input_array[0]);
   printf("Channel 2:  %d\n\n", (int)input_array[1]);
   printf("Channel 1:  %d\n\n", (int)input_array[2]);
   }

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleDigitalInputScan is a very basic interface without any allowance for timing information, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the digital input configuration procedure (DaqDigitalInput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's digital input interface, the following example program creates the equivalent of the DaqSingleDigitalInputScan procedure.

```
unsigned short MySingleDigitalInputScan(unsigned short logical_device,
                                        unsigned short far *channel_array,
                                        unsigned short array_length,
                                        void far *input_array)
{
struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_array;
my_request.array_length      = array_length;
my_request.ADC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = (void huge*)input_array;
my_data.buffer_length = array_length;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_EMPTY;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqDigitalInput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

## 3.4 Digital Output

DAQDRIVE provides two special purpose procedures for digital output: DaqSingleDigitalOutput and DaqSingleDigitalOutputScan. The intent of this section is to provide an overview of these procedures. For details on the implementation of the procedures, consult the alphabetical listing of commands in chapter 13.

### 3.4.1 DaqSingleDigitalOutput

One of the simplest cases of digital output is to output a single value to a single digital I/O channel under CPU control. DAQDRIVE provides a simplified interface for this function through the DaqSingleDigitalOutput procedure. The format of this command is shown below.

```
unsigned short DaqSingleDigitalOutput(unsigned short logical_device,
                                      unsigned short channel_number,
                                      void far *output_value)
```

DaqSingleDigitalOutput outputs the value specified by output_value to the digital I/O channel specified by channel_number on the adapter specified by logical_device. The following example shows the usage of DaqSingleDigitalOutput.

```
/*** Output a single sample to digital I/O channel 2 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned char output_value;
char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Output the value to channel 2 ***/

output_value = 1;
status = DaqSingleDigitalOutput(logical_device, 2, &output_value);
if (status != 0)
   printf("\n\nDigital I/O output error.  Status code %d.\n\n", status);
else
   printf("\n\nComplete.  No errors.);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleDigitalOutput is a very basic interface without any allowance for multiple channels, multiple output values, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the digital output configuration procedure (DaqDigitalOutput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's digital output interface, the following example program creates the equivalent of the DaqSingleDigitalOutput procedure.

```
unsigned short MySingleDigitalOutput(unsigned short logical_device,
                                     unsigned short channel_number,
                                     void far *output_value)
{
struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = (void huge*)output_value;
my_data.buffer_length  = 1;
my_data.buffer_cycles  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_FULL;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqDigitalOutput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
}
```

### 3.4.2 DaqSingleDigitalOutputScan

Another simple case of digital output is to output one value each to multiple digital I/O channels under CPU control. This allows multiple digital outputs to be updated simultaneously (or nearly simultaneously depending on the data acquisition hardware). A simplified interface for this operation is provided through the DaqSingleDigitalOutputScan procedure. The format of this command is shown below.

```
unsigned short DaqSingleDigitalOutputScan(unsigned short logical_device,
                                          unsigned short far *channel_array,
                                          unsigned short array_length,
                                          void far *output_array)
```

DaqSingleDigitalOutputScan outputs the values in the array specified by output_array to the digital I/O channels in the array specified by channel_array on the adapter specified by logical_device. A digital I/O channel may appear in channel_array only once and a one-to-one correspondence is required between the number of digital output channels and the number of output values. Therefore, array_length specifies the length of both channel_array and output_array. The following example shows the usage of DaqSingleDigitalOutputScan.

```
/*** Output a single sample to digital I/O channels 1, 2, 3, 4, and 5 ***/

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned short channel_array[5] = { 1, 2, 3, 4, 5 };
unsigned char  output_array[5]  = { 3, 0, 0, 1, 3 };
char far *device_type = "DA8P-12B";
char far *config_file = "da8p-12b.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Output the digital I/O values ***/

status = DaqSingleDigitalOutputScan(logical_device, channel_array,
                                    5, output_array);
if (status != 0)
   printf("\n\nDigital I/O output error.  Status code %d.\n\n", status);
else
   printf("\n\nComplete.  No errors.);

/*** Step 3: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

DaqSingleDigitalOutputScan is a very basic interface without any allowance for timing information, trigger sources, etc. It acts as a DAQDRIVE macro defining the necessary data structures, executing the digital output configuration procedure (DaqDigitalOutput), arming the requested configuration (DaqArmRequest), and triggering the operation (DaqTriggerRequest).

For users interested in learning more about DAQDRIVE's digital output interface, the following example program creates the equivalent of the DaqSingleDigitalOutputScan procedure.

```
unsigned short MySingleDigitalOutputScan(unsigned short logical_device,
                                         unsigned short far *channel_array,
                                         unsigned short array_length,
                                         void far *output_array)
{
struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;
unsigned short          request_handle;
unsigned short          status;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_array;
my_request.array_length      = array_length;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = (void huge*)output_array;
my_data.buffer_length = array_length;
my_data.buffer_cycles = 1;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_FULL;

/*****  Execute the request  *****/

request_handle = 0;
status = DaqDigitalOutput(logical_device, &my_request, &request_handle);
if (status != 0)
   return(status);

/*****  If no errors, arm the request  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, software trigger the request  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   return(status);
   }

/*****  If no errors, release the request and return  *****/

status = DaqReleaseRequest(request_handle);
return(status);
```

# 4 Performing An Acquisition

DAQDRIVE uses a "data defined" rather than a "function defined" interface. What this means is that each data acquisition operation is defined by a series of configuration parameters and requires very few function calls to implement. These parameters, which are contained in a data structure, are hereafter referred to as a ***request structure*** or simply a ***request*** and define such parameters as channel numbers, sampling rate, number of scans, trigger source, etc. The key to unlocking the power and flexibility of DAQDRIVE lies in the understanding of these request structures.

Another parameter which the user needs to become familiar with is DAQDRIVE's use of ***request handles*** which are used to identify valid request structures. When a request structure is passed into one of the configuration routines, DAQDRIVE verifies the contents of the structure and confirms that the target hardware can perform the type of operation requested. If the request structure is valid, a request handle is assigned to the structure and all future operations on this request are referenced using its request handle.

The following steps define the sequence required to perform an operation using DAQDRIVE's data defined interface.

**Step 1: Define The Hardware Configuration**

DAQDRIVE determines the configuration of a device from the data file specified when the device is opened. These configuration files are created using the DAQDRIVE configuration utility as described in section 2.2.

**Step 2: Open The Hardware Device**

Before the application program can use an adapter, it must first open the device using the DaqOpenDevice command. The application must provide the open command with the adapter type and specify the name of a configuration file (generated in step 1) which describes the target hardware's configuration. If the open command completes successfully, DAQDRIVE assigns a logical device number to be used for all future references to the adapter.

**Step 3: Define The Request Structure And Data Buffers**

With the device successfully opened, the application must now allocate and define all of the parameters associated with the operation to be performed. These parameters include such variables as the channel number(s), trigger source, and sampling rate. DAQDRIVE's request structures are covered in detail in chapters 5 through 8. In addition to the request structure, the application must define one or more data buffer structures where the request's data is stored. These data buffer structures are discussed in chapter 9.

**Step 4: Request The Operation**

The next step is to request the operation. The configuration procedures serves to validate the contents of the request structure and to determine if the target hardware can support the type of operation requested. If the request is not valid, an error is returned and the application must redefine the request. If the request is valid and the operation is supported by the hardware, a request handle is issued to identify this configuration. Once the request handle is issued, the channel(s) specified in the request structure's channel list are allocated for use by this request. Any other hardware resources required to execute the request (timers, triggers, etc.) remain available until the request is armed. Chapters 5 through 8 contain detailed descriptions of each type of DAQDRIVE request.

**Step 5: Arm The Request**

With a valid configuration requested, the application must now arm the request in order to prepare the hardware for the impending trigger. It is during the arm procedure, DaqArmRequest, that the hardware is programmed and any system resources required for the request (i.e. IRQ levels, DMA channels, timers, etc.) are allocated and assigned to the request. An error will occur during the arm process if any of the required resources are not available.

**Step 6: Trigger The Request**

After the request is armed, the next step is to trigger the request and start the operation. If the request specified an internal (software) trigger, the application must now issue the trigger using DaqTriggerRequest. If the request specified any of the hardware trigger sources, the application may wait for the trigger event to occur or it may continue to step 7.

**Step 7: Wait For Completion**

With the operation in progress, the application must wait for the request to be completed before any further action can be taken on this request. If necessary, the application can terminate the request using the DaqStopRequest procedure. When the operation is completed or otherwise terminated, any system resources allocated by the request are freed for use by other requests. However, the channel(s) specified in the request structure's channel list remain allocated to the request until the request is released by the DaqReleaseRequest procedure.

**Step 8: Release The Configuration**

After the operation is complete (or otherwise terminated), the request may be released using DaqReleaseRequest. Releasing the request frees the channel(s) used by the request making them available for future requests.

**Step 9: Close The Hardware Device**

The final step, after all operations on the hardware are complete, is to close the device using DaqCloseDevice to free any remaining resources used by that device. System integrity can not be guaranteed if the application program exits without closing the hardware device.

# 5 Analog Input Requests

In chapter 3, some special purpose procedures were presented to help the user get familiar with DAQDRIVE's A/D converter interface. The key to understanding and utilizing DAQDRIVE, however, is to understand its request structures. This chapter will present the analog input request structure and provide examples to illustrate how this structure is configured for some common applications.

## 5.1 DaqAnalogInput

DaqAnalogInput is DAQDRIVE's A/D converter interface. Any analog input operation is possible with the proper configuration of the request structure. The format of the command is shown below.

```
unsigned short DaqAnalogInput(unsigned short logical_device,
                              struct ADC_request far *user_request,
                              unsigned short far *request_handle)
```

DaqAnalogInput performs the configuration portion of an analog input request. For a new configuration, the application program sets request_handle to 0 before calling DaqAnalogInput. DaqAnalogInput then analyzes the data structure specified by user_request to determine if all of the parameters are valid and if the requested operation can be performed by the device specified by logical_device. If the requested operation is valid, DaqAnalogInput assigns request_handle a unique non-zero value. This request handle is used to identify this request in all future operations.

If the application program modifies the contents of user_request after executing DaqAnalogInput, the structure must be verified again. To request re-verification of a previously approved request, the application executes DaqAnalogInput with request_handle set to the value returned by DaqAnalogInput when the request was first approved. All parameters except the channel list may be modified after the initial configuration. To modify the channel list, the existing request must be released (using DaqReleaseRequest) and a new configuration requested.

## 5.2 The Analog Input Request Structure

The power of DaqAnalogInput lies in the application's ability to modify a single data structure and execute a single procedure to perform multiple analog input operations. The elements of the analog input request structure are discussed on the following pages.

```
struct ADC_request
   {
   unsigned short far *channel_array_ptr;
   float far *gain_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *ADC_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel,
   double trigger_voltage;
   unsigned long trigger_value,
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short calibration;
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

## IMPORTANT:

1. If the application program modifies the contents of the request structure after executing DaqAnalogInput, the updated structure must be re-verified by DaqAnalogInput before the request is armed.

2. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.

3. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

### 5.2.1 Reserved Fields

The fields reserved1 through reserved5 are provided for expansion of the analog input request structure in future releases of DAQDRIVE. To maintain maximum compatibility, the application program should initialize all reserved fields to 0.

### 5.2.2 Channel Selections / Gain Settings

The analog input request structure begins with a list of one or more analog input channels to be operated on by this request. The application provides the memory address of the first channel in the list using the channel_array_ptr field. In addition to the channel list, the application must provide a gain setting for each channel in the channel list. The application provides the memory address of the first gain setting in the gain list using the gain_array_ptr field. For each channel in the channel list there must be one and only one setting in the gain list. Therefore, the lengths of both lists are specified by the array_length field.

### 5.2.3 Data Buffers

ADC_buffer defines the request's data buffer structure(s) to be used for storing the data input from the specified channel(s). The application program must define these buffers within the guidelines provided in chapter 9.

### 5.2.4 Trigger Selections

The trigger selection determines how the requested operation will be initiated after being armed. Six fields are required to define and configure the trigger for the request: trigger_source, trigger_mode, trigger_slope, trigger_channel, trigger_voltage, and trigger_value. Because the trigger selection is an integral part of the operation and is common to all of DAQDRIVE's request structures. trigger configurations are discussed separately in chapter 10.

### 5.2.5 Data Transfer Modes

The request structure field IO_mode determines the mechanism that will be used to input the data from the hardware device. In general, the foreground modes provide the highest data transfer rates at the expense of requiring 100% of the CPU time. In contrast, background mode operations generally provide lower data transfer rates while allowing the CPU to perform other tasks.

#### 5.2.5.1 Foreground CPU mode

This mode uses the CPU to input the data from the hardware device. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

#### 5.2.5.2 Background IRQ mode

This mode uses interrupts generated by the hardware device to gain control of the CPU to input the data to the hardware device. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

5.2.5.3 Foreground DMA mode

This mode uses the DMA controller to input the data from the hardware device while using the CPU to monitor and control the DMA operation. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

5.2.5.4 Background DMA mode

This mode uses the DMA controller to input the data from the hardware device while using interrupts generated by the hardware device to gain control of the CPU to monitor and control the DMA operation. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

### 5.2.6 Clock Sources

The clock_source field is used to define the source of the timing signal for requests acquiring multiple samples.

5.2.6.1 Internal Clock

When the clock source field is set for an internal clock, the timing for the request is provided by the adapter's on-board timer circuitry. The clock_rate field is unused with the internal clock source and any value provided in the clock_rate field is ignored.

5.2.6.2 External Clock

Setting the clock_source field to external indicates the timing for the request is provided by a signal input to the adapter as defined by the hardware device. The clock_rate field must be used to define the frequency of the external clock signal in Hertz.

### 5.2.7 Sampling Rate

The sampling rate specifies the number of samples / second (Hz) to be input from the hardware device. The application specifies a desired sampling rate in the sample_rate field of the request structure. On most hardware devices, only a finite number of sampling rates are achievable. When DaqAnalogInput configures a request, the closest available sampling rate is selected and the sample_rate field is updated with the actual rate at which the data will be input.

### 5.2.8 Number Of Scans

The number_of_scans field determines the number of times the channel(s) specified in the channel list are processed. For example, to input 100 samples from a single A/D channel, number_of_scans must be set to 100. To input 50 samples each from 6 A/D channels (300 points total), number_of_scans is set to 50.

### 5.2.9 Scan Events

DAQDRIVE generates a scan event each time the number of scans specified by scan_event_level are completed. For example, if scan_event_level is set to 50, a scan event is

generated every time the channel array is processed 50 times. DAQDRIVE events are discussed in detail in chapter 11.

### 5.2.10 Calibration Selections

The calibration field allows the application to specify the type of calibration to be performed (if any) by the hardware device(s) during the requested operation. In general, enabling calibration results in lower throughput rates while providing greater accuracy.

#### 5.2.10.1 Auto-calibration

Enabling auto-calibration instructs the hardware device to perform one or more calibration cycles on the A/D converter(s) specified by this request. The results of auto-calibration vary with different hardware devices. Consult the hardware user's manual and the appendices in the back of this document for details about how auto-calibration operates on the device in use.

#### 5.2.10.2 Auto-zero

Enabling auto-zero instructs the hardware device to perform one or more zero offset adjustment cycles on the A/D converter(s) specified by this request. The results of auto-zeroing vary with different hardware devices. Consult the hardware user's manual and the appendices in the back of this document for details about how auto-zero operates on the device in use.

### 5.2.11 Time-out

The timeout_interval field is used primarily during foreground mode operations to instruct DAQDRIVE when to abandon the processing of a request. When DAQDRIVE has control of the CPU and is waiting for an event to occur (i.e. waiting for a trigger or waiting for the A/D to complete a conversion), DAQDRIVE will wait timeout_interval seconds and if the event has not occurred, the request will be aborted.

### 5.2.12 Request Status

The request_status field provides a mechanism for the application to monitor the state of a request. The request status is an integral part of DAQDRIVE's event mechanisms and is discussed in detail in chapter 11.

## 5.3 Analog Input Examples

### 5.3.1 Example 1 - Single Channel Input

Purpose: Input 1000 samples from a single analog input channel at a 10KHz sampling rate.

```
unsigned short channel_list = 0;
         float gain_list    = 2;
unsigned short input_values[1000];

struct ADC_request      my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_list;
my_request.gain_array+ptr    = &gain_list;
my_request.array_length      = 1;
my_request.ADC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = BACKGROUND_IRQ;
my_request.clock_source      = INTERNAL_CLOCK;
my_request.sample_rate       = 10000;
my_request.number_of_scans   = 1000;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = input_values;
my_data.buffer_length  = 1000;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_EMPTY;
```

Variations on example 1:

1. To change the operating mode from background mode with interrupts to foreground mode using DMA, set
   my_request.IO_mode = FOREGROUND_DMA

2. To change the trigger mode to an analog trigger, set
   my_request.trigger_source = ANALOG_TRIGGER
   my_request.trigger_channel = 0
   my_request.trigger_voltage = 3.0
   my_request.trigger_slope = RISING_EDGE

3. To enable calibration for the request, set
   my_request.calibration = AUTO_CALIBRATE or
   my_request.calibration = AUTO_ZERO or
   my_request.calibration = AUTO_CALIBRATE | AUTO_ZERO

### 5.3.2 Example 2 - Multiple Channel Input

Purpose: Input 1000 sample from each of 4 analog input channels at a rate of 500Hz.

```
unsigned short channel_number[4] = {0, 1, 14,   6};
         float gain_settings[4]  = {1, 1, 10, 100};
unsigned short input_values[4 * 1000];

struct ADC_request      my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_number;
my_request.gain_array_ptr    = gain_settings;
my_request.array_length      = 4;
my_request.ADC_buffer        = &my_data;
my_request.trigger_source    = TTL_TRIGGER;
my_request.trigger_slope     = RISING_EDGE;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.clock_source      = INTERNAL_CLOCK;
my_request.sample_rate       = 500;
my_request.number_of_scans   = 1000;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = input_values;
my_data.buffer_length = 4 * 1000;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_EMPTY;
```

Variations on example 2:

1. To change the number of points from 1000 per channel to 5000 per channel, re-define input_values[ ] and then set
   my_request.number_of_scans = 5000
   my_data.buffer_length = 4 * 5000

2. To notify the application every time 100 scans are complete, set
   my_request.scan_event_level = 100

3. To enable a time-out if no data is available for a period of 3 seconds, set
   my_request.timeout_interval = 3

# 6 Analog Output Requests

In chapter 3, some special purpose procedures were presented to help the user get familiar with DAQDRIVE's D/A converter interface. The key to understanding and utilizing DAQDRIVE, however, is to understand its request structures. This chapter will present the analog output request structure and provide examples to illustrate how this structure is configured for some common applications.

## 6.1 DaqAnalogOutput

DaqAnalogOutput is DAQDRIVE's D/A converter interface. Any analog output operation is possible with the proper configuration of the request structure. The format of the command is shown below.

```
unsigned short DaqAnalogOutput(unsigned short logical_device,
                               struct DAC_request far *user_request,
                               unsigned short far *request_handle)
```

DaqAnalogOutput performs the configuration portion of an analog output request. For a new configuration, the application program sets request_handle to 0 before calling DaqAnalogOutput. DaqAnalogOutput then analyzes the data structure specified by user_request to determine if all of the parameters are valid and if the requested operation can be performed by the device specified by logical_device. If the requested operation is valid, DaqAnalogOutput assigns request_handle a unique non-zero value. This request handle is used to identify this request in all future operations.

If the application program modifies the contents of user_request after executing DaqAnalogOutput, the structure must be verified again. To request re-verification of a previously approved request, the application executes DaqAnalogOutput with request_handle set to the value returned by DaqAnalogOutput when the request was first approved. All parameters except the channel list may be modified after the initial configuration. To modify the channel list, the existing request must be released (using DaqReleaseRequest) and a new configuration requested.

## 6.2 The Analog Output Request Structure

The power of DaqAnalogOutput lies in the application's ability to modify a single data structure and execute a single procedure to perform multiple analog output operations. The elements of the analog output request structure are discussed on the following pages.

```
struct DAC_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *DAC_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel,
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short calibration;
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

## IMPORTANT:

1. If the application program modifies the contents of the request structure after executing DaqAnalogOutput, the updated structure must be re-verified by DaqAnalogOutput before the request is armed.
2. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.
3. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

### 6.2.1 Reserved Fields

The fields reserved1 through reserved5 are provided for expansion of the analog output request structure in future releases of DAQDRIVE. To maintain maximum compatibility, the application program should initialize all reserved fields to 0.

### 6.2.2 Channel Selections

The analog output request structure begins with a list of one or more analog output channels to be operated on by this request. The application provides the memory address of the first channel in the list using the channel_array_ptr field and must specify the length of the list in the array_length field.

### 6.2.3 Data Buffers

DAC_buffer defines the request's data buffer structure(s) containing the data to be output to the specified channel(s). The application program must define these buffers within the guidelines provided in chapter 9.

### 6.2.4 Trigger Selections

The trigger selection determines how the requested operation will be initiated after being armed. Six fields are required to define and configure the trigger for the request: trigger_source, trigger_mode, trigger_slope, trigger_channel, trigger_voltage, and trigger_value. Because the trigger selection is an integral part of the operation and is common to all of DAQDRIVE's request structures. trigger configurations are discussed separately in chapter 10.

### 6.2.5 Data Transfer Modes

The request structure field IO_mode determines the mechanism that will be used to output the data to the hardware device. In general, the foreground modes provide the highest data transfer rates at the expense of requiring 100% of the CPU time. In contrast, background mode operations generally provide lower data transfer rates while allowing the CPU to perform other tasks.

#### 6.2.5.1 Foreground CPU mode

This mode uses the CPU to output the data to the hardware device. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

#### 6.2.5.2 Background IRQ mode

This mode uses interrupts generated by the hardware device to gain control of the CPU to output the data to the hardware device. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

6.2.5.3 Foreground DMA mode

This mode uses the DMA controller to output the data to the hardware device while using the CPU to monitor and control the DMA operation. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

6.2.5.4 Background DMA mode

This mode uses the DMA controller to output the data to the hardware device while using interrupts generated by the hardware device to gain control of the CPU to monitor and control the DMA operation. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

### 6.2.6 Clock Sources

The clock_source field is used to define the source of the timing signal for requests containing multiple data values.

6.2.6.1 Internal Clock

When the clock source field is set for an internal clock, the timing for the request is provided by the adapter's on-board timer circuitry. The clock_rate field is unused with the internal clock source and any value provided in the clock_rate field is ignored.

6.2.6.2 External Clock

Setting the clock_source field to external indicates the timing for the request is provided by a signal input to the adapter as defined by the hardware device. The clock_rate field must be used to define the frequency of the external clock signal in Hertz.

### 6.2.7 Sampling Rate

The sampling rate specifies the number of samples / second (Hz) to be output to the hardware device. The application specifies a desired sampling rate in the sample_rate field of the request structure. On most hardware devices, only a finite number of sampling rates are achievable. When DaqAnalogOutput configures a request, the closest available sampling rate is selected and the sample_rate field is updated with the actual rate at which the data will be output.

### 6.2.8 Number Of Scans

The number_of_scans field determines the number of times the channel(s) specified in the channel list are processed. For example, to output 100 samples to a single D/A channel, number_of_scans must be set to 100. To output 50 samples each to two D/A channels (100 points total), number_of_scans is set to 50.

### 6.2.9 Scan Events

DAQDRIVE generates a scan event each time the number of scans specified by scan_event_level are completed. For example, if scan_event_level is set to 50, a scan event is

generated every time the channel array is processed 50 times. DAQDRIVE events are discussed in detail in chapter 11.

### 6.2.10 Calibration Selections

The calibration field allows the application to specify the type of calibration to be performed (if any) by the hardware device(s) during the requested operation. In general, enabling calibration results in lower throughput rates while providing greater accuracy.

#### 6.2.10.1 Auto-calibration

Enabling auto-calibration instructs the hardware device to perform one or more calibration cycles on the D/A converter(s) specified by this request. The results of auto-calibration vary with different hardware devices. Consult the hardware user's manual and the appendices in the back of this document for details about how auto-calibration operates on the device in use.

#### 6.2.10.2 Auto-zero

Enabling auto-zero instructs the hardware device to perform one or more zero offset adjustment cycles on the D/A converter(s) specified by this request. The results of auto-zeroing vary with different hardware devices. Consult the hardware user's manual and the appendices in the back of this document for details about how auto-zero operates on the device in use.

### 6.2.11 Time-out

The timeout_interval field is used primarily during foreground mode operations to instruct DAQDRIVE when to abandon the processing of a request. When DAQDRIVE has control of the CPU and is waiting for an event to occur (i.e. waiting for a trigger or waiting for the D/A to become ready), DAQDRIVE will wait timeout_interval seconds and if the event has not occurred, the request will be aborted.

### 6.2.12 Request Status

The request_status field provides a mechanism for the application to monitor the state of a request. The request status is an integral part of DAQDRIVE's event mechanisms and is discussed in detail in chapter 11.

## 6.3 Analog Output Examples

### 6.3.1 Example 1 - DC Voltage Level Output

Purpose: Output a single value to each of three analog output channels.

```
unsigned short channel_list[]  = {     4, 0,   1 };
unsigned short output_values[] = { -1024, 0, 512 };

struct DAC_request      my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_list;
my_request.array_length      = 3;
my_request.DAC_buffer        = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = output_values;
my_data.buffer_length  = 3;
my_data.buffer_cycles  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_FULL;
```

Variations on example 1:

1. To change the number of channels from three to five, re-define channel_list[ ] and output_values[ ] then set
   my_request.array_length = 5
   my_data.buffer_length = 5

2. To change the trigger mode to a TTL trigger, set
   my_request.trigger_source = TTL_TRIGGER
   my_request.trigger_slope = RISING_EDGE

3. To enable calibration for the request, set
   my_request.calibration = AUTO_CALIBRATE or
   my_request.calibration = AUTO_ZERO or
   my_request.calibration = AUTO_CALIBRATE | AUTO_ZERO.

### 6.3.2 Example 2 - Simple Waveform Generation

Purpose: Output 300 cycles of a 60 Hz sinewave defined with 180 points per cycle.

```
unsigned short channel_number;
unsigned short sinewave[180];

struct DAC_request      my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Assume data values have been calculated  *****/
/*****  and stored in sinewave[]                 *****/

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.DAC_buffer        = &my_data;
my_request.trigger_source    = TTL_TRIGGER;
my_request.trigger_slope     = RISING_EDGE;
my_request.IO_mode           = FOREGROUND_DMA;
my_request.clock_source      = INTERNAL_CLOCK;
my_request.sample_rate       = 60 * 180;
my_request.number_of_scans   = 300 * 180;
my_request.scan_event_level  = 0;
my_request.calibration       = NO_CALIBRATION;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = sinewave;
my_data.buffer_length = 180;
my_data.buffer_cycles = 300;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_FULL;
```

Variations on example 2:

1. To change the number of points from 180 to 360, re-define sinewave[ ] and then set
   my_request.sample_rate = 60 * 360
   my_request.number_of_scans = 300 * 360
   my_data.buffer_length = 360

2. To change the number of cycles from 300 to 15,000, set
   my_request.number_of_scans = 15000
   my_data.buffer_cycles = 15000

3. To enable a time-out if the trigger does not occur within 15 seconds after the request is armed, set
   my_request.timeout_interval = 15

# 7 Digital Input Requests

In chapter 3, some special purpose procedures were presented to help the user get familiar with DAQDRIVE's digital input interface. The key to understanding and utilizing DAQDRIVE, however, is to understand its request structures. This chapter will present the digital input request structure and provide examples to illustrate how this structure is configured for some common applications.

## 7.1 DaqDigitalInput

DaqDigitalInput is DAQDRIVE's digital input interface. Any digital input operation is possible with the proper configuration of the request structure. The format of the command is shown below.

```
unsigned short DaqDigitalInput(unsigned short logical_device,
                               struct digio_request far *user_request,
                               unsigned short far *request_handle)
```

DaqDigitalInput performs the configuration portion of a digital input request. For a new configuration, the application program sets request_handle to 0 before calling DaqDigitalInput. DaqDigitalInput then analyzes the data structure specified by user_request to determine if all of the parameters are valid and if the requested operation can be performed by the device specified by logical_device. If the requested operation is valid, DaqDigitalInput assigns request_handle a unique non-zero value. This request handle is used to identify this request in all future operations.

If the application program modifies the contents of user_request after executing DaqDigitalInput, the structure must be verified again. To request re-verification of a previously approved request, the application executes DaqDigitalInput with request_handle set to the value returned by DaqDigitalInput when the request was first approved. All parameters except the channel list may be modified after the initial configuration. To modify the channel list, the existing request must be released (using DaqReleaseRequest) and a new configuration requested.

## 7.2 The Digital Input Request Structure

The power of DaqDigitalInput lies in the application's ability to modify a single data structure and execute a single procedure to perform multiple digital input operations. The elements of the digital input request structure are discussed on the following pages.

```
struct digio_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *digio_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel,
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

## IMPORTANT:

1. If the application program modifies the contents of the request structure after executing DaqDigitalInput, the updated structure must be re-verified by DaqDigitalInput before the request is armed.
2. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.
3. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

### 7.2.1 Reserved Fields

The fields reserved1 through reserved5 are provided for expansion of the digital input request structure in future releases of DAQDRIVE. To maintain maximum compatibility, the application program should initialize all reserved fields to 0.

### 7.2.2 Channel Selections

The digital input request structure begins with a list of one or more digital input channels to be operated on by this request. The application provides the memory address of the first channel in the list using the channel_array_ptr field and must specify the length of the list in the array_length field.

### 7.2.3 Data Buffers

digio_buffer defines the request's data buffer structure(s) to be used for storing the data input from the specified channel(s). The application program must define these buffers within the guidelines provided in chapter 9.

### 7.2.4 Trigger Selections

The trigger selection determines how the requested operation will be initiated after being armed. Six fields are required to define and configure the trigger for the request: trigger_source, trigger_mode, trigger_slope, trigger_channel, trigger_voltage, and trigger_value. Because the trigger selection is an integral part of the operation and is common to all of DAQDRIVE's request structures. trigger configurations are discussed separately in chapter 10.

### 7.2.5 Data Transfer Modes

The request structure field IO_mode determines the mechanism that will be used to input the data from the hardware device. In general, the foreground modes provide the highest data transfer rates at the expense of requiring 100% of the CPU time. In contrast, background mode operations generally provide lower data transfer rates while allowing the CPU to perform other tasks.

#### 7.2.5.1 Foreground CPU mode

This mode uses the CPU to input the data from the hardware device. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

#### 7.2.5.2 Background IRQ mode

This mode uses interrupts generated by the hardware device to gain control of the CPU to input the data to the hardware device. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

7.2.5.3 Foreground DMA mode

This mode uses the DMA controller to input the data from the hardware device while using the CPU to monitor and control the DMA operation. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

7.2.5.4 Background DMA mode

This mode uses the DMA controller to input the data from the hardware device while using interrupts generated by the hardware device to gain control of the CPU to monitor and control the DMA operation. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

### 7.2.6 Clock Sources

The clock_source field is used to define the source of the timing signal for requests acquiring multiple samples.

7.2.6.1 Internal Clock

When the clock source field is set for an internal clock, the timing for the request is provided by the adapter's on-board timer circuitry. The clock_rate field is unused with the internal clock source and any value provided in the clock_rate field is ignored.

7.2.6.2 External Clock

Setting the clock_source field to external indicates the timing for the request is provided by a signal input to the adapter as defined by the hardware device. The clock_rate field must be used to define the frequency of the external clock signal in Hertz.

### 7.2.7 Sampling Rate

The sampling rate specifies the number of samples / second (Hz) to be input from the hardware device. The application specifies a desired sampling rate in the sample_rate field of the request structure. On most hardware devices, only a finite number of sampling rates are achievable. When DaqDigitalInput configures a request, the closest available sampling rate is selected and the sample_rate field is updated with the actual rate at which the data will be output.

### 7.2.8 Number Of Scans

The number_of_scans field determines the number of times the channel(s) specified in the channel list are processed. For example, to input 100 samples from a single digital input channel, number_of_scans must be set to 100. To input 50 samples each from four digital input channels (200 points total), number_of_scans is set to 50.

### 7.2.9 Scan Events

DAQDRIVE generates a scan event each time the number of scans specified by scan_event_level are completed. For example, if scan_event_level is set to 50, a scan event is

generated every time the channel array is processed 50 times. DAQDRIVE events are discussed in detail in chapter 11.

### 7.2.10 Time-out

The timeout_interval field is used primarily during foreground mode operations to instruct DAQDRIVE when to abandon the processing of a request. When DAQDRIVE has control of the CPU and is waiting for an event to occur (i.e. waiting for a trigger or waiting for the digital input channel to become ready), DAQDRIVE will wait timeout_interval seconds and if the event has not occurred, the request will be aborted.

### 7.2.11 Request Status

The request_status field provides a mechanism for the application to monitor the state of a request. The request status is an integral part of DAQDRIVE's event mechanisms and is discussed in detail in chapter 11.

## 7.3 Digital Input Examples

### 7.3.1 Example 1 - Single Value Input

Purpose: Input a single value from each of three digital input channels.

```
unsigned short channel_list[]  = { 0, 1, 2 };
unsigned char  input_values[3];

struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_list;
my_request.array_length      = 3;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = input_values;
my_data.buffer_length  = 3;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_EMPTY;
```

Variations on example 1:

1. To change the number of channels from three to eight, re-define channel_list[ ] and input_values[ ] then set
   my_request.array_length = 8
   my_data.buffer_length = 8

2. To change the trigger mode to a TTL trigger, set
   my_request.trigger_source = TTL_TRIGGER
   my_request.trigger_slope = RISING_EDGE

3. To enable a time-out if no data is available after 5 seconds, set
   my_request.timeout_interval = 5

### 7.3.2 Example 2 - Multiple Value Input

Purpose: Input 500 points from a single digital input channel at 1 second intervals.

```
unsigned short channel_number;
unsigned char  input_values[500];

struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = TTL_TRIGGER;
my_request.trigger_slope     = RISING_EDGE;
my_request.IO_mode           = BACKGROUND_IRQ;
my_request.clock_source      = INTERNAL_CLOCK;
my_request.sample_rate       = 1;
my_request.number_of_scans   = 500;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = input_values;
my_data.buffer_length = 500;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_EMPTY;
```

Variations on example 2:

1. To change the number of points from 500 to 650, re-define input_values[ ] and then set
   my_request.number_of_scans = 650
   my_data.buffer_length = 650

2. To notify the application every time 100 scans are complete, set
   my_request.scan_event_level = 100

# 8 Digital Output Requests

In chapter 3, some special purpose procedures were presented to help the user get familiar with DAQDRIVE's digital output interface. The key to understanding and utilizing DAQDRIVE, however, is to understand its request structures. This chapter will present the digital output request structure and provide examples to illustrate how this structure is configured for some common applications.

## 8.1 DaqDigitalOutput

DaqDigitalOutput is DAQDRIVE's digital output interface. Any digital output operation is possible with the proper configuration of the request structure. The format of the command is shown below.

```
unsigned short DaqDigitalOutput(unsigned short logical_device,
                                struct digio_request far *user_request,
                                unsigned short far *request_handle)
```

DaqDigitalOutput performs the configuration portion of a digital output request. For a new configuration, the application program sets request_handle to 0 before calling DaqDigitalOutput. DaqDigitalOutput then analyzes the data structure specified by user_request to determine if all of the parameters are valid and if the requested operation can be performed by the device specified by logical_device. If the requested operation is valid, DaqDigitalOutput assigns request_handle a unique non-zero value. This request handle is used to identify this request in all future operations.

If the application program modifies the contents of user_request after executing DaqDigitalOutput, the structure must be verified again. To request re-verification of a previously approved request, the application executes DaqDigitalOutput with request_handle set to the value returned by DaqDigitalOutput when the request was first approved. All parameters except the channel list may be modified after the initial configuration. To modify the channel list, the existing request must be released (using DaqReleaseRequest) and a new configuration requested.

## 8.2 The Digital Output Request Structure

The power of DaqDigitalOutput lies in the application's ability to modify a single data structure and execute a single procedure to perform multiple digital output operations. The elements of the digital output request structure are discussed on the following pages.

```
struct digio_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *digio_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel,
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

## IMPORTANT:

1. If the application program modifies the contents of the request structure after executing DaqDigitalOutput, the updated structure must be re-verified by DaqDigitalOutput before the request is armed.
2. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.
3. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

### 8.2.1 Reserved Fields

The fields reserved1 through reserved5 are provided for expansion of the digital output request structure in future releases of DAQDRIVE. To maintain maximum compatibility, the application program should initialize all reserved fields to 0.

### 8.2.2 Channel Selections

The digital output request structure begins with a list of one or more digital output channels to be operated on by this request. The application provides the memory address of the first channel in the list using the channel_array_ptr field and must specify the length of the list in the array_length field.

### 8.2.3 Data Buffers

digio_buffer defines the request's data buffer structure(s) containing the data to be output to the specified channel(s). The application program must define these buffers within the guidelines provided in chapter 9.

### 8.2.4 Trigger Selections

The trigger selection determines how the requested operation will be initiated after being armed. Six fields are required to define and configure the trigger for the request: trigger_source, trigger_mode, trigger_slope, trigger_channel, trigger_voltage, and trigger_value. Because the trigger selection is an integral part of the operation and is common to all of DAQDRIVE's request structures. trigger configurations are discussed separately in chapter 10.

### 8.2.5 Data Transfer Modes

The request structure field IO_mode determines the mechanism that will be used to output the data to the hardware device. In general, the foreground modes provide the highest data transfer rates at the expense of requiring 100% of the CPU time. In contrast, background mode operations generally provide lower data transfer rates while allowing the CPU to perform other tasks.

#### 8.2.5.1 Foreground CPU mode

This mode uses the CPU to output the data to the hardware device. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

#### 8.2.5.2 Background IRQ mode

This mode uses interrupts generated by the hardware device to gain control of the CPU to output the data to the hardware device. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

8.2.5.3 Foreground DMA mode

This mode uses the DMA controller to output the data to the hardware device while using the CPU to monitor and control the DMA operation. From the moment the request is triggered, DAQDRIVE uses all of the CPU time and will not return control to the application program until the request is completed or otherwise terminated.

8.2.5.4 Background DMA mode

This mode uses the DMA controller to output the data to the hardware device while using interrupts generated by the hardware device to gain control of the CPU to monitor and control the DMA operation. DAQDRIVE does not require all of the CPU time in this mode and returns control of the CPU to the application after the request is triggered.

### 8.2.6 Clock Sources

The clock_source field is used to define the source of the timing signal for requests containing multiple data values.

8.2.6.1 Internal Clock

When the clock source field is set for an internal clock, the timing for the request is provided by the adapter's on-board timer circuitry. The clock_rate field is unused with the internal clock source and any value provided in the clock_rate field is ignored.

8.2.6.2 External Clock

Setting the clock_source field to external indicates the timing for the request is provided by a signal input to the adapter as defined by the hardware device. The clock_rate field must be used to define the frequency of the external clock signal in Hertz.

### 8.2.7 Sampling Rate

The sampling rate specifies the number of samples / second (Hz) to be output to the hardware device. The application specifies a desired sampling rate in the sample_rate field of the request structure. On most hardware devices, only a finite number of sampling rates are achievable. When DaqDigitalOutput configures a request, the closest available sampling rate is selected and the sample_rate field is updated with the actual rate at which the data will be output.

### 8.2.8 Number Of Scans

The number_of_scans field determines the number of times the channel(s) specified in the channel list are processed. For example, to output 100 samples to a single digital output channel, number_of_scans must be set to 100. To output 50 samples each to two digital output channels (100 points total), number_of_scans is set to 50.

### 8.2.9 Scan Events

DAQDRIVE generates a scan event each time the number of scans specified by scan_event_level are completed. For example, if scan_event_level is set to 50, a scan event is

generated every time the channel array is processed 50 times. DAQDRIVE events are discussed in detail in chapter 11.

### 8.2.10 Time-out

The timeout_interval field is used primarily during foreground mode operations to instruct DAQDRIVE when to abandon the processing of a request. When DAQDRIVE has control of the CPU and is waiting for an event to occur (i.e. waiting for a trigger or waiting for the digital output channel to become ready), DAQDRIVE will wait timeout_interval seconds and if the event has not occurred, the request will be aborted.

### 8.2.11 Request Status

The request_status field provides a mechanism for the application to monitor the state of a request. The request status is an integral part of DAQDRIVE's event mechanisms and is discussed in detail in chapter 11.

## 8.3 Digital Output Examples

### 8.3.1 Example 1 - Single Value Output

Purpose: Output a single value to each of two digital output channels.

```
unsigned short channel_list[]  = { 1,   0 };
unsigned char  output_values[] = { 0, 255 };

struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = channel_list;
my_request.array_length      = 2;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = INTERNAL_TRIGGER;
my_request.IO_mode           = FOREGROUND_CPU;
my_request.number_of_scans   = 1;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer    = output_values;
my_data.buffer_length  = 2;
my_data.buffer_cycles  = 1;
my_data.next_structure = NULL;
my_data.buffer_status  = BUFFER_FULL;
```

Variations on example 1:

1. To change the number of channels from two to three, re-define channel_list[ ] and output_values[ ] then set
   my_request.array_length = 3
   my_data.buffer_length = 3

2. To change the trigger mode to a TTL trigger, set
   my_request.trigger_source = TTL_TRIGGER
   my_request.trigger_slope = FALLING_EDGE

### 8.3.2 Example 2 - Simple Pattern Generation

Purpose: Output 100 cycles of a digital pattern defined with 128 points per cycle with 10ms between samples.

```
unsigned short channel_number;
unsigned char  pattern[128];

struct digio_request    my_request;
struct DAQDRIVE_buffer  my_data;

/*****  Assume data values have been calculated  *****/
/*****  and stored in pattern[]                  *****/

/*****  Construct the request structure  *****/

my_request.channel_array_ptr = &channel_number;
my_request.array_length      = 1;
my_request.digio_buffer      = &my_data;
my_request.trigger_source    = EXTERNAL_TRIGGER;
my_request.trigger_slope     = RISING_EDGE;
my_request.IO_mode           = BACKGROUND_IRQ;
my_request.clock_source      = INTERNAL_CLOCK;
my_request.sample_rate       = 100;
my_request.number_of_scans   = 100 * 128;
my_request.scan_event_level  = 0;
my_request.timeout_interval  = 0;
my_request.request_status    = NO_EVENTS;

/*****  Construct the data buffer structure  *****/

my_data.data_buffer   = pattern;
my_data.buffer_length = 128;
my_data.buffer_cycles = 100;
my_data.next_buffer   = NULL;
my_data.buffer_status = BUFFER_FULL;
```

Variations on example 2:

1. To change the number of points from 128 to 64, re-define pattern[ ] and then set
   my_request.number_of_scans = 100 * 64
   my_data.buffer_length = 64

2. To change the number of cycles from 100 to 800, set
   my_request.number_of_scans = 800 * 128
   my_data.buffer_cycles = 800

# 9 Defining Data Buffers

DAQDRIVE's data buffers are defined as structures containing the buffer configuration and a pointer to the data storage area. This allows multiple data buffers to be defined as each buffer is completely self-contained.

```
struct DAQDRIVE_buffer
   {
   unsigned short buffer_status;
   void huge *data_buffer;
   unsigned long  buffer_length;
   unsigned long  buffer_cycles;
   struct DAQDRIVE_buffer far *next_structure;
   };
```

buffer_status - This unsigned short integer value is used to monitor / control the current state of the data buffer. buffer_status is defined for both input operations (see figure 4) and output operations (see figure 5).

data_buffer - This void huge pointer specifies the address of the actual input / output data buffer. data_buffer is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by data_buffer is correct for the request type and the target hardware as listed in the tables below.

| Request type | Resolution | Configuration | data type |
|---|---|---|---|
| A / D or D / A | 1 to 8 bits | unipolar | unsigned char |
| | | bipolar | signed char |
| | 9 to 16 bits | unipolar | unsigned short |
| | | bipolar | signed short |
| | 17 to 32 bits | unipolar | unsigned long |
| | | bipolar | signed long |

| Request type | Channel size (in bits) | data type |
|---|---|---|
| digital input or digital output | 1 to 8 bits | unsigned char |
| | 9 to 16 bits | unsigned short |
| | 17 to 32 bits | unsigned long |

buffer_length - This unsigned long integer value defines the length of data_buffer in units of "number-of-points". Each data buffer must be large enough to hold at least 1 point for every channel in the channel list. Therefore buffer_length must be greater than 0.

buffer_cycles - This unsigned long integer value is used during output operations (D/A, digital output) only to define the number of times the data in this structure is processed before continuing on to the next_structure. Setting buffer_cycles = 0 causes the data in this buffer to be processed continuously (next_structure will never be accessed). buffer_cycles is undefined for input operations and any value in this field will be ignored.

next_structure - This structure pointer is used to connect multiple data buffers for larger acquisition requests. When the data buffer associated with this structure has been filled (or emptied), DAQDRIVE will switch to the structure pointed to by next_structure and continue the operation using the new structure's data buffer. next_buffer is set to NULL if there are no more structures in the chain.

## IMPORTANT:

1. Once the request is armed using DaqArmRequest, the application program must obey the rules defined in figures 4 and 5 for accessing the buffer structures and data buffers at run-time. These rules apply until the operation is completed or otherwise terminated.
2. If the buffer structures or the data buffers are dynamically allocated by the application, they **MUST NOT** be de-allocated until the request is completed or otherwise terminated. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32 or DaqAllocateRequest if dynamically allocated buffer structures or data buffers are required.

| buffer_status - INPUT OPERATIONS | | | |
|---|---|---|---|
| Bit | Value | DAQDRIVE constant | Description |
| 0 | 0x0001 | BUFFER_FULL | BUFFER_FULL indicates the data buffer associated with this structure is full.<br><br>DAQDRIVE sets BUFFER_FULL to 1 after the last value is transferred into the data buffer. Once BUFFER_FULL is set, DAQDRIVE will not operate on this buffer again unless BUFFER_EMPTY is re-set to 1 by the application program. DAQDRIVE will never clear BUFFER_FULL during input operations.<br><br>The application program may use BUFFER_FULL to determine when a data buffer may be safely accessed. After the application sets BUFFER_EMPTY and arms the request, it **MUST NOT** modify the contents of the DAQDRIVE_buffer structure or the associated data buffer until DAQDRIVE sets BUFFER_FULL to 1 or until the operation is halted. The application may clear BUFFER_FULL at any time. |
| 1 | 0x0002 | BUFFER_EMPTY | BUFFER_EMPTY indicates the data buffer associated with this structure is empty.<br><br>The application must set BUFFER_EMPTY to 1 to inform DAQDRIVE that the data buffer is ready for input. After the application sets BUFFER_EMPTY and arms the request, it **MUST NOT** modify the contents of the DAQDRIVE_buffer structure or the associated data buffer until DAQDRIVE sets BUFFER_FULL to 1 or until the operation is halted (this includes modifying BUFFER_EMPTY).<br><br>DAQDRIVE clears BUFFER_EMPTY to 0 when the first data value is transferred into the buffer and will report a buffer over-run error if a data buffer is encountered that does not have the BUFFER_EMPTY bit set. DAQDRIVE will never set BUFFER_EMPTY during input operations. |

Figure 4. buffer_status definition for input operations (A/D and digital input).

| buffer_status - OUTPUT OPERATIONS | | | |
|---|---|---|---|
| Bit | Value | DAQDRIVE constant | Description |
| 0 | 0x0001 | BUFFER_FULL | BUFFER_FULL indicates the data buffer associated with this structure is full.<br><br>The application must set BUFFER_FULL to 1 to inform DAQDRIVE that the data buffer is ready for output. After the application sets BUFFER_FULL and arms the request, it **MUST NOT** modify the contents of the DAQDRIVE_buffer structure or the associated data buffer until DAQDRIVE sets BUFFER_EMPTY to 1 or until the operation is halted (this includes modifying BUFFER_FULL).<br><br>DAQDRIVE clears BUFFER_FULL to 0 when the first data value is removed from the buffer and will report a buffer under-run error if a data buffer is encountered that does not have BUFFER_FULL bit set. DAQDRIVE will never set BUFFER_FULL during output operations. |
| 1 | 0x0002 | BUFFER_EMPTY | BUFFER_EMPTY indicates the data buffer associated with this structure is empty.<br><br>DAQDRIVE sets BUFFER_EMPTY to 1 after the last value is removed from the data buffer. Once BUFFER_EMPTY is set, DAQDRIVE will not operate on this buffer again unless BUFFER_FULL is re-set to 1 by the application program. DAQDRIVE will never clear BUFFER_EMPTY during output operations.<br><br>The application program may use BUFFER_EMPTY to determine when a data buffer may be safely accessed. After the application sets BUFFER_FULL and arms the request, it **MUST NOT** modify the contents of the DAQDRIVE_buffer structure or the associated data buffer until DAQDRIVE sets BUFFER_EMPTY to 1 or until the operation is halted. The application may clear BUFFER_EMPTY at any time. |

Figure 5. buffer_status definition for output operations (D/A and digital output).

## 9.1 Multiple Channel Operations

When defining a data buffer for single channel operations, the data buffer is simply an array of values and is stored in system memory in continuous, increasing memory locations. For example, if an application requests 10 samples from a single analog input channel, the application must declare an array to hold the ten values

```
short array[10];
```

This array appears in system memory as

```
array[0], array[1], array[2], ..., array[9]
```

When DAQDRIVE is acquiring the data, however, it does not view this memory as an array but simply as a data buffer. For this single channel example, DAQDRIVE would place the following information in the data buffer

```
sample1, sample2, sample3, ..., sample10
```

which the application views as

```
array[0] = sample1
array[1] = sample2
array[2] = sample3
      :
      :
array[9] = sample10
```

As mentioned above, DAQDRIVE views the memory as a data buffer and not as an array. If this same buffer was used to acquire 5 samples from each of two A/D channels, DAQDRIVE would place the following data in the buffer

```
chan1, chan2, chan1, chan2, ..., chan1, chan2
```

which the application would view as

```
array[0] = chan1  (sample #1)
array[1] = chan2  (sample #1)
array[2] = chan1  (sample #2)
array[3] = chan2  (sample #2)
      :
      :
array[8] = chan1  (sample #5)
array[9] = chan2  (sample #5)
```

Obviously, as the number of channels increases, it becomes more difficult to determine the correlation between the channel number and the value.

One solution to this problem is to use two-dimensional arrays. Re-defining the array of the previous example to

```
short array[5][2];
```

does not change the size of the data buffer. The array now appears in memory as

```
array[0][0], array[0][1], ..., array[4][0], array[4][1]
```

and after DAQDRIVE loads the data into the buffer, the application views the data as

```
array[0][0] = chan1     (sample #1)
array[0][1] = chan2     (sample #1)
array[1][0] = chan1     (sample #2)
array[1][1] = chan2     (sample #2)
      :
      :
array[4][0] = chan1     (sample #5)
array[4][1] = chan2     (sample #5)
```

In general terms, the array may be defined as

```
array[sample_number][channel_number];
```

Although the previous examples only illustrated the definition of data buffers for input operations, it should be noted that all DAQDRIVE requests implement the same buffer structure and therefore all output data buffers must be defined accordingly.

## 9.2 Input Operation Examples

### 9.2.1 Example 1: Single Channel Analog Input

An example of a simple input operation is to acquire 100 samples from a single A/D channel. To perform this operation, the application must first allocate enough memory to hold 100 samples. Assuming a 12-bit A/D converter operating in bipolar mode, the samples are each size "short". Therefore, the memory allocation may be done as simply as

```
short input_data[100];
```

The next step is to allocate and configure a DAQDRIVE_buffer structure. data_buffer is set to point to the array defined above. Its length is 100 points and there are no other structures so next_structure is set to NULL.

```
struct DAQDRIVE_buffer my_ADC_data;

my_ADC_data.data_buffer    = input_data;
my_ADC_data.buffer_length  = 100;
my_ADC_data.next_structure = NULL;
```

The next step is to allocate and configure an ADC_request structure. The ADC_request structure is beyond the scope of this chapter and will not be discussed here except for the ADC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, ADC_buffer is set to point to our DAQDRIVE_buffer structure and the number_of_scans is set to 100 scans (1 channel / scan).

```
struct ADC_request my_ADC_request;

my_ADC_request.ADC_buffer      = my_ADC_data;
my_ADC_request.number_of_scans = 100;
```

The final step is to set the BUFFER_EMPTY status in the buffer_status field. Once BUFFER_EMPTY is set and the request is armed, the application must not modify my_ADC_data or input_data until BUFFER_FULL is set or until the operation is terminated.

```
my_ADC_data.buffer_status = BUFFER_EMPTY;
```

### 9.2.2 Example 2: Multi-Channel Analog Input

The purpose of this example is to input 500 samples each from three analog input channels. To perform this operation, the application must first allocate enough memory to hold 1500 (3 * 500) samples. Assuming a 16-bit A/D converter operating in unipolar mode, the samples are each size "unsigned short". Therefore, the memory allocation may be done as simply as

```
unsigned short input_data[1500];
```

The next step is to allocate and configure a DAQDRIVE_buffer structure. data_buffer is set to point to the array defined above. Its length is 1500 points and there are no other structures so next_structure is set to NULL.

```
struct DAQDRIVE_buffer my_ADC_data;

my_ADC_data.data_buffer    = input_data;
my_ADC_data.buffer_length  = 1500;
my_ADC_data.next_structure = NULL;
```

The next step is to allocate and configure an ADC_request structure. The ADC_request structure is beyond the scope of this chapter and will not be discussed here except for the ADC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, ADC_buffer is set to point to our DAQDRIVE_buffer structure and the number_of_scans is set to 500 scans (3 channels / scan).

```
struct ADC_request my_ADC_request;

my_ADC_request.ADC_buffer      = my_ADC_data;
my_ADC_request.number_of_scans = 500;
```

The final step is to set the BUFFER_EMPTY status in the buffer_status field. Once BUFFER_EMPTY is set and the request is armed, the application must not modify my_ADC_data or input_data until BUFFER_FULL is set or until the operation is terminated.

```
my_ADC_data.buffer_status = BUFFER_EMPTY;
```

### 9.2.3 Example 3: Using Multiple Data Buffers

The purpose of this example is to use multiple data buffers to input 25,000 samples from a single analog input channel. The application could allocate a single 25,000 sample buffer but for this example will allocate one 10,000 sample buffer and one 15,000 sample buffer. Assuming a 12-bit A/D converter operating in unipolar mode, the samples are each size "unsigned short".

```
unsigned short input_data0[10000];
unsigned short input_data1[15000];
```

The next step is to allocate and configure two DAQDRIVE_buffer structures. The data_buffer fields are set to point to the arrays defined above and the buffer_length fields are set accordingly. The first structure has its next_structure field set to point to the second structure. The second structure has its next_structure field set to NULL.

```
struct DAQDRIVE_buffer my_ADC_data[2];

my_ADC_data[0].data_buffer    = input_data0;
my_ADC_data[0].buffer_length  = 10000;
my_ADC_data[0].next_structure = &my_ADC_data[1];

my_ADC_data[1].data_buffer    = input_data1;
my_ADC_data[1].buffer_length  = 15000;
my_ADC_data[1].next_structure = NULL;
```

The next step is to allocate and configure an ADC_request structure. The ADC_request structure is beyond the scope of this chapter and will not be discussed here except for the ADC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, ADC_buffer is set to point to our first DAQDRIVE_buffer structure and the number_of_scans is set to 25,000 scans (1 channel / scan).

```
struct ADC_request my_ADC_request;

my_ADC_request.ADC_buffer      = my_ADC_data;
my_ADC_request.number_of_scans = 25000;
```

The final step is to set the BUFFER_EMPTY status in the buffer_status fields. Once BUFFER_EMPTY is set and the request is armed, the application must not modify the DAQDRIVE_buffer structures or the associated input data buffers until BUFFER_FULL is set or until the operation is terminated.

```
my_ADC_data[0].buffer_status = BUFFER_EMPTY;
my_ADC_data[1].buffer_status = BUFFER_EMPTY;
```

### 9.2.4 Example 4: Acquiring Large Amounts Of Data

The purpose of example 4 is to illustrate one way to acquire large amounts of data using relatively small amounts of memory. If, for example, an application wants to input 100,000 samples from each of 5 analog input channels using a 12-bit A/D converter operating in unipolar mode, the samples are each size "unsigned short" and 500,000 samples would require 1 Megabyte of memory. This example will acquire the 500,000 samples using only 40K of memory. The first step is to allocate two buffers with 10,000 points each.

```
unsigned short input_data0[10000];
unsigned short input_data1[10000];
```

The next step is to allocate and configure two DAQDRIVE_buffer structures. The data_buffer fields are set to point to the arrays defined above and the buffer_length fields are set accordingly. The first structure has its next_structure field set to point to the second structure. The second structure has its next_structure field set to point to the first structure forming a circular buffer.

```
struct DAQDRIVE_buffer my_ADC_data[2];

my_ADC_data[0].data_buffer    = input_data0;
my_ADC_data[0].buffer_length  = 10000;
my_ADC_data[0].next_structure = &my_ADC_data[1];

my_ADC_data[1].data_buffer    = input_data1;
my_ADC_data[1].buffer_length  = 10000;
my_ADC_data[1].next_structure = &my_ADC_data[0];
```

The next step is to allocate and configure an ADC_request structure. The ADC_request structure is beyond the scope of this chapter and will not be discussed here except for the ADC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, ADC_buffer is set to point to our first DAQDRIVE_buffer structure and the number_of_scans is set to 100,000 scans (5 channel / scan).

```
struct ADC_request my_ADC_request;

my_ADC_request.ADC_buffer      = my_ADC_data;
my_ADC_request.number_of_scans = 100000;
```

The key to acquiring 500,000 samples with only enough buffer space to hold 20,000 samples is in the use of the BUFFER_FULL and BUFFER_EMPTY bits. Before arming the request, the BUFFER_EMPTY bits in the buffer_status fields are set

```
my_ADC_data[0].buffer_status = BUFFER_EMPTY;
my_ADC_data[1].buffer_status = BUFFER_EMPTY;
```

The application may not modify the DAQDRIVE_buffer structures or the data buffers until the operation is halted or until DAQDRIVE sets the BUFFER_FULL bit. If this is a background operation, the application may sit in a loop waiting for BUFFER_FULL and processing each buffer as it becomes available.

```
// wait in dead loop until buffer 0 is full

while((my_ADC_data[0].buffer_status & BUFFER_FULL) == 0);

// buffer 0 is full. process data, clear BUFFER_FULL, and re-set BUFFER_EMPTY

my_ADC_data[0].buffer_status = BUFFER_EMPTY;

// wait in dead loop until buffer 1 is full

while((my_ADC_data[1].buffer_status & BUFFER_FULL) == 0);

// buffer 1 is full. process data, clear BUFFER_FULL, and re-set BUFFER_EMPTY

my_ADC_data[1].buffer_status = BUFFER_EMPTY;

// repeat until 500,000 samples are processed
```

Another option for background mode operations is to monitor the BUFFER_FULL_EVENT bit in the request_status field of the ADC_request structure. The application may assume the BUFFER_FULL bit is set before the BUFFER_FULL_EVENT is generated and that the application may safely access the data buffer.

```
// wait in dead loop until a buffer is full

while((my_ADC_request.request_status & BUFFER_FULL_EVENT)==0);

// a buffer is full. determine which buffer, process the
// data, clear BUFFER_FULL, and re-set BUFFER_EMPTY

if((my_ADC_data[0].buffer_status & BUFFER_FULL) != 0)
   {
   // process buffer 0

   my_ADC_data[0].buffer_status = BUFFER_EMPTY;
   }
else
   {
   // process buffer 1

   my_ADC_data[1].buffer_status = BUFFER_EMPTY;
   }

// repeat until 500,000 samples are processed
```

Another option for background mode operations, and the only option available for foreground mode operations, is to use the event notification procedure DaqNotifyEvent. The idea of event notification is that DAQDRIVE will execute a user-supplied procedure each time an event occurs. This mechanism can be used to process a data buffer on each occurrence of the BUFFER_FULL_EVENT. The details of event notification are beyond the scope of this chapter but are discussed in chapter 11.

The methods shown in example 4 will work only if the application can process the data and re-set BUFFER_EMPTY before DAQDRIVE tries to access that buffer again. If DAQDRIVE

tries to access a buffer in which the BUFFER_EMPTY bit has not been set, a buffer over-run error will occur.

## 9.3 Output Operation Examples

### 9.3.1 Example 1: Single Channel Analog Output

An example of a simple output operation is to write 100 samples to a single D/A channel. To perform this operation, the application must first allocate enough memory to hold 100 samples. Assuming a 12-bit D/A converter operating in bipolar mode, the samples are each size "short". Therefore, the memory allocation may be done as simply as

```
short output_data[100];
```

The next step is to allocate and configure a DAQDRIVE_buffer structure. data_buffer is set to point to the array defined above. Its length is 100 points, the buffer will be processed only once (buffer_cycles = 1), and there are no other structures so next_structure is set to NULL.

```
struct DAQDRIVE_buffer my_DAC_data;

my_DAC_data.data_buffer    = output_data;
my_DAC_data.buffer_length  = 100;
my_DAC_data.buffer_cycles  = 1;
my_DAC_data.next_structure = NULL;
```

The next step is to allocate and configure a DAC_request structure. The DAC_request structure is beyond the scope of this chapter and will not be discussed here except for the DAC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, DAC_buffer is set to point to our DAQDRIVE_buffer structure and the number_of_scans is set to 100 scans (1 channel / scan).

```
struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 100;
```

The final step is to set the BUFFER_FULL status in the buffer_status field. Once BUFFER_FULL is set and the request is armed, the application must not modify my_DAC_data or output_data until BUFFER_EMPTY is set or until the operation is terminated.

```
my_DAC_data.buffer_status = BUFFER_FULL;
```

### 9.3.2 Example 2: Creating Repetitive Signals

In example 1, 100 samples were output to a single D/A channel. If these 100 points represent a sinewave and the desired output is 250 cycles of this sinewave, the application could allocate 25,000 (250 * 100) points, calculate 250 cycles of the sinewave, and output 25,000 points to the D/A. A simpler approach is to change the configuration as shown below (the original values from example 1 are shown in the comments).

```
short output_data[100];

struct DAQDRIVE_buffer my_DAC_data;

my_DAC_data.data_buffer    = output_data;
my_DAC_data.buffer_length  = 100;
my_DAC_data.buffer_cycles  = 250;  /* was = 1 */
my_DAC_data.next_structure = NULL;

struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 25000;  /* was = 100 */

my_DAC_data.buffer_status = BUFFER_FULL;
```

By changing the number_of_scans to 25,000, the application is instructing DAQDRIVE to output 25,000 samples to the D/A channel. In order to generate these 25,000 points, however, the application is also instructing DAQDRIVE to process the data buffer 250 times (buffer_cycles = 250). The result is that the 100 points contained in the data buffer will be output to the D/A converter 250 times producing the equivalent of a 25,000 point data buffer containing 250 cycles of the sinewave.

**NOTE:**
In this example, setting buffer_cycles = 250 effectively created a 25,000 point data buffer. Setting buffer_cycles = 300 would have effectively created a 30,000 point data buffer. Had a 30,000 point data buffer been used with number_of_scans set to 25,000, the result would be the same except the BUFFER_EMPTY bit would not have been set since all 30,000 points were not output to the D/A.

### 9.3.3 Example 3: Multi-Channel Analog Output

The purpose of this example is to output 500 samples each to three analog output channels. To perform this operation, the application must first allocate enough system memory to hold 1500 (3 * 500) samples. Assuming a 12-bit D/A converter operating in unipolar mode, the samples are each size "unsigned short". Therefore, the memory allocation may be done as simply as

```
unsigned short output_data[1500];
```

The next step is to allocate and configure a DAQDRIVE_buffer structure. data_buffer is set to point to the array defined above. Its length is 1500 points, the buffer will be processed only once (buffer_cycles = 1), and there are no other structures so next_structure is set to NULL.

```
struct DAQDRIVE_buffer my_DAC_data;

my_DAC_data.data_buffer    = output_data;
my_DAC_data.buffer_length  = 1500;
my_DAC_data.buffer_cycles  = 1;
my_DAC_data.next_structure = NULL;
```

The next step is to allocate and configure a DAC_request structure. The DAC_request structure is beyond the scope of this chapter and will not be discussed here except for the DAC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, DAC_buffer is set to point to our DAQDRIVE_buffer structure and the number_of_scans is set to 500 scans (3 channels / scan).

```
struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 500;
```

The final step is to set the BUFFER_FULL status in the buffer_status field. Once BUFFER_FULL is set and the request is armed, the application must not modify my_DAC_data or output_data until BUFFER_EMPTY is set or until the operation is terminated.

```
my_DAC_data.buffer_status = BUFFER_FULL;
```

### 9.3.4 Example 4: Using Multiple Data Buffers

The purpose of this example is to use multiple data buffers to output 4,000 points each to two analog output channels. The application could allocate a single 8,000 (2* 4000) sample buffer but for this example will allocate one 5,000 sample buffer and one 3,000 sample buffer. Assuming a 12-bit D/A converter operating in unipolar mode, the samples are each size "unsigned short".

```
unsigned short output_data0[5000];
unsigned short output_data1[3000];
```

The next step is to allocate and configure two DAQDRIVE_buffer structures. The data_buffer fields are set to point to the arrays defined above and the buffer_length fields are set accordingly. The first structure has its next_structure field set to point to the second structure. The second structure has its next_structure field set to NULL.

```
struct DAQDRIVE_buffer my_DAC_data[2];

my_DAC_data[0].data_buffer    = output_data0;
my_DAC_data[0].buffer_length  = 5000;
my_DAC_data[0].buffer_cycles  = 1;
my_DAC_data[0].next_structure = &my_DAC_data[1];

my_DAC_data[1].data_buffer    = output_data1;
my_DAC_data[1].buffer_length  = 3000;
my_DAC_data[1].buffer_cycles  = 1;
my_DAC_data[1].next_structure = NULL;
```

The next step is to allocate and configure a DAC_request structure. The DAC_request structure is beyond the scope of this chapter and will not be discussed here except for the DAC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, DAC_buffer is set to point to our first DAQDRIVE_buffer structure and number_of_scans is set to 4,000 scans (2 channels / scan).

```
struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 4000;
```

The final step is to set the BUFFER_EMPTY status in the buffer_status fields. Once BUFFER_EMPTY is set and the request is armed, the application must not modify the DAQDRIVE_buffer structures or the associated input data buffers until BUFFER_FULL is set or until the operation is terminated.

```
my_DAC_data[0].buffer_status = BUFFER_EMPTY;
my_DAC_data[1].buffer_status = BUFFER_EMPTY;
```

### 9.3.5 Example 5: Creating Complex Output Patterns

Combining the ideas of example 2 and example 4, the application of example 5 wants to output 50 cycles of a sinewave containing 360 samples, 45 cycles of a square wave containing 2 samples, and 75 cycles of a triangle wave containing 30 samples. Assuming a 12-bit D/A converter operating in bipolar mode, the following arrays are defined

```
short sine[360];
short square[2];
short triangle[30];
```

The next step is to allocate and configure three DAQDRIVE_buffer structures. The data_buffer fields are set to point to the arrays defined above and the buffer_length fields are set accordingly. The first structure has its next_structure field set to point to the second structure. The second structure has its next_structure field set to point to the third structure, and the third structure has its next_structure field set to NULL since it is the last structure in the chain.

```
struct DAQDRIVE_buffer my_DAC_data[3];

my_DAC_data[0].data_buffer    = sine;
my_DAC_data[0].buffer_length  = 360;
my_DAC_data[0].buffer_cycles  = 50;
my_DAC_data[0].next_structure = &my_DAC_data[1];

my_DAC_data[1].data_buffer    = square;
my_DAC_data[1].buffer_length  = 2;
my_DAC_data[1].buffer_cycles  = 45;
my_DAC_data[1].next_structure = &my_DAC_data[2];

my_DAC_data[2].data_buffer    = triangle;
my_DAC_data[2].buffer_length  = 30;
my_DAC_data[2].buffer_cycles  = 75;
my_DAC_data[2].next_structure = NULL;
```

The next step is to allocate and configure a DAC_request structure. The DAC_request structure is beyond the scope of this chapter and will not be discussed here except for the DAC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, DAC_buffer is set to point to our first DAQDRIVE_buffer structure and the number_of_scans is defined as follows

```
number_of_scans =  number_of_samples / samples_per_scan
                =  50 cycles * 360 samples / cycle     (sine)
                +  45 cycles * 2 samples / cycle       (square)
                +  75 cycles * 30 samples / cycle      (triangle)
                =  20,340 samples / (1 sample / scan)
                =  20,340 scans
```

```
struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 20340;
```

The final step is to set the BUFFER_EMPTY status in the buffer_status fields. Once BUFFER_EMPTY is set and the request is armed, the application must not modify the DAQDRIVE_buffer structures or the associated input data buffers until BUFFER_FULL is set or until the operation is terminated.

```
my_DAC_data[0].buffer_status = BUFFER_EMPTY;
my_DAC_data[1].buffer_status = BUFFER_EMPTY;
my_DAC_data[2].buffer_status = BUFFER_EMPTY;
```

Variations on example 5

1. To execute only the sinewave portion of the buffers, simply change number_of_scans to 50 * 360

   ```
   my_DAC_request.number_of_scans = 18000;
   ```

2. To change the square wave portion of the output from 45 cycles to 300 cycles, change the corresponding buffer_cycles to 300 and number_of_scans to (50 * 360) + (300 * 2) + (30 * 75)

   ```
   my_DAC_data[1].buffer_cycles   = 300;
   my_DAC_request.number_of_scans = 20850;
   ```

3. If the triangle wave is redefined to have 60 samples, change the corresponding buffer_length to 60 and number_of_scans to (50 * 360) + (45 * 2) + (75 * 60)

   ```
   my_DAC_data[2].buffer_length   = 60;
   my_DAC_request.number_of_scans = 22590;
   ```

### 9.3.6 Example 6: Outputting Large Amounts Of Data

The multiple buffer operation of example 4 can be extended for the application that needs to output large numbers of points. Assume 500,000 points need to be read from a file and output to a 12-bit D/A converter. If all of the samples are input from the file at once, 1 Megabyte of memory would be required to hold the data. An alternative solution may be to allocate two buffers with 25,000 points each.

```
unsigned short output_data0[25000];
unsigned short output_data1[25000];
```

The next step is to allocate and configure two DAQDRIVE_buffer structures. The data_buffer fields are set to point to the arrays defined above and the buffer_length fields are set accordingly. The first structure has its next_structure field set to point to the second structure. The second structure has its next_structure field set to point to the first structure forming a circular buffer.

```
struct DAQDRIVE_buffer my_DAC_data[2];

my_DAC_data[0].data_buffer    = output_data0;
my_DAC_data[0].buffer_length  = 25000;
my_DAC_data[0].buffer_cycles  = 1;
my_DAC_data[0].next_structure = &my_DAC_data[1];

my_DAC_data[1].data_buffer    = output_data1;
my_DAC_data[1].buffer_length  = 25000;
my_DAC_data[1].buffer_cycles  = 1;
my_DAC_data[1].next_structure = &my_DAC_data[0];
```

The next step is to allocate and configure a DAC_request structure. The DAC_request structure is beyond the scope of this chapter and will not be discussed here except for the DAC_buffer and number_of_scans fields which directly relate to the data structure configuration. For this example, DAC_buffer is set to point to our first DAQDRIVE_buffer structure and number_of_scans is set to 500,000 scans (1 channel / scan).

```
struct DAC_request my_DAC_request;

my_DAC_request.DAC_buffer      = my_DAC_data;
my_DAC_request.number_of_scans = 500000;
```

The key to processing 500,000 samples with only enough buffer space to hold 50,000 samples is in the use of the BUFFER_FULL and BUFFER_EMPTY bits. Before arming the request, the BUFFER_FULL bits in the buffer_status fields are set

```
my_DAC_data[0].buffer_status = BUFFER_FULL;
my_DAC_data[1].buffer_status = BUFFER_FULL;
```

The application may not modify the DAQDRIVE_buffer structures or the data buffers until the operation is halted or until DAQDRIVE sets the BUFFER_EMPTY bit. If this is a background operation, the application may sit in a loop waiting for BUFFER_EMPTY and re-filling each buffer as it becomes available.

```
// wait in dead loop until buffer 0 is empty

while((my_DAC_data[0].buffer_status & BUFFER_EMPTY) == 0);

// buffer 0 is empty. re-fill buffer, clear BUFFER_EMPTY and re-set BUFFER_FULL

my_DAC_data[0].buffer_status = BUFFER_FULL;

// wait in dead loop until buffer 1 is empty

while((my_DAC_data[1].buffer_status & BUFFER_EMPTY) == 0);

// buffer 1 is empty. re-fill buffer, clear BUFFER_EMPTY and re-set BUFFER_FULL

my_DAC_data[1].buffer_status = BUFFER_FULL;

// repeat until 500,000 samples are processed
```

Another option for background mode operations is to monitor the BUFFER_EMPTY_EVENT bit in the DAC_request structure's request_status field. The application may assume the BUFFER_EMPTY bit is set before the BUFFER_EMPTY_EVENT is generated and that the application may safely access the data buffer.

```
// wait in dead loop for BUFFER_EMPTY_EVENT

while((my_DAC_request.request_status & BUFFER_EMPTY_EVENT)==0);

// a buffer is empty. determine which buffer, re-fill the
// buffer, clear BUFFER_EMPTY, and re-set BUFFER_FULL

if((my_DAC_data[0].buffer_status & BUFFER_EMPTY) != 0)
   {
   // re-fill buffer 0

   my_DAC_data[0].buffer_status = BUFFER_FULL;
   }
else
   {
   // re-fill buffer 1

   my_DAC_data[1].buffer_status = BUFFER_FULL;
   }

// repeat until 500,000 samples are processed
```

Another option for background mode operations, and the only option available for foreground mode operations, is to use the event notification procedure DaqNotifyEvent. The idea of event notification is that DAQDRIVE will execute a user-supplied procedure each time an event occurs. This mechanism can be used to re-fill the data buffers on each occurrence of the BUFFER_EMPTY_EVENT. The details of event notification are beyond the scope of this chapter but are discussed in chapter 11. The methods shown in example 5 will work only if the application can process the data and re-set BUFFER_FULL before DAQDRIVE tries to access that buffer again. If DAQDRIVE tries to access a buffer in which the BUFFER_FULL bit has not been set, a buffer under-run error will occur.

# 10 Trigger Selections

Once a request has been configured and armed, the trigger determines when the requested operation will begin. A summary of available trigger sources and their required parameters is shown in figure 6 below.

| Source | Slope | Channel | Voltage | Value |
|---|---|---|---|---|
| internal | | | | |
| TTL | x | | | |
| analog | x | x | x | |
| digital | | x | | x |

Figure 6. Summary of DAQDRIVE trigger sources and parameters.

## 10.1 Trigger Sources

When a request is configured, the application program must specify a trigger source in the request structure. Depending on which trigger is specified, additional trigger related fields in the structure may also be required (see figure 6). When these additional settings are not required, any value provided in the field is ignored.

### 10.1.1 Internal Trigger

The simplest trigger source is an internal trigger, also referred to as a software trigger. To generate an internal trigger, the application program must execute the DaqTriggerRequest procedure. The internal trigger source does not require any additional configuration parameters and any values provided in these fields are ignored.

### 10.1.2 TTL Trigger

The TTL trigger is a specific TTL input to the hardware device that is designated by the adapter as a trigger input. When the TTL trigger source is selected, the trigger slope must also be defined as either rising edge, requiring a low-to-high transition of the trigger signal, or falling edge, requiring a high-to-low transition of the trigger signal. The trigger channel, trigger voltage, and trigger value settings are not required and any values provided in these fields is ignored.

### 10.1.3 Analog Trigger

The analog trigger source allows a request to be initiated by an analog input voltage level. When the analog trigger is selected, the application must specify the voltage required to generate the trigger and the analog input channel to be monitored for this trigger voltage. In addition, the trigger slope must be specified as either rising edge, the voltage must transition from below the trigger voltage to above the trigger voltage, or falling edge, the voltage must

transition from above the trigger voltage to below the trigger voltage. The trigger value setting is not required for an analog trigger and any value provided in this field is ignored.

### 10.1.4 Digital Trigger

The digital trigger allows a request to be initiated when a specific value is detected on a digital input channel. When the digital trigger is selected, the application must specify the digital input channel to be monitored and the value that must be received to generate the trigger. The trigger slope and trigger voltage settings are not required for a digital trigger and any value provided in these fields is ignored.

## 10.2 Trigger Modes

DAQDRIVE supports two trigger modes, one-shot and continuous. When the application configures a request, the trigger mode must be specified along with the trigger source.

### 10.2.1 One-shot Trigger Mode

When a request is configured for one-shot trigger mode, a separate occurrence of the trigger is required for each scan of the channel list. For example, if a single digital output channel is configured for an internal trigger in one-shot mode, each call to DaqTriggerRequest will output one sample to the specified digital channel. If a request is configured to input data from six analog inputs with a rising edge TTL trigger in one-shot mode, then each low-to-high transition of the TTL trigger input will cause six samples to be input (one sample from each of the six channels in the channel list).

### 10.2.2 Continuous Trigger Mode

When a request is configured for continuous trigger mode, only one trigger occurrence is required to initiate the request; the remainder of the operation is executed periodically at time intervals specified by the sample rate. For example, if a request is configured to output data to an analog output channel with a falling edge TTL trigger in continuous mode at a sample rate of 1KHz, then a high-to-low transition of the TTL trigger will output the first sample with additional samples following at 1ms intervals (1KHz). If a request is configured to input data from ten digital inputs with a continuous mode internal trigger at a 50Hz sample rate, then ten samples will be input when the DaqTriggerRequest procedure is executed and ten more samples will be input at each 20ms (50Hz) interval thereafter.

# 11 DAQDRIVE Events

DAQDRIVE uses events to keep the application program informed of the progress of a request. The following sections provide descriptions of DAQDRIVE events and methods of monitoring these events from the application program.

## 11.1 Event Descriptions

### 11.1.1 Trigger Event

The trigger event is generated when a valid trigger is received. If the request was configured for continuous trigger mode, only one trigger event will occur when the operation is initiated. If the request was configured for one-shot trigger mode, a trigger event is generated with each occurrence of the trigger.

### 11.1.2 Complete Event

The complete event is generated when a request has completed successfully. If the complete event occurs at the same time as the buffer full event, the buffer empty event, and/or the scan event, the events are reported in the following sequence: scan event, buffer full or buffer empty event, complete event. A request will never report any events after the complete event.

### 11.1.3 Buffer Empty Event

The buffer empty event is generated during output operations each time one of the specified output data buffers has been completely emptied. If a buffer empty event occurs at the same time as the complete event, the buffer full event is reported before the complete event. If a buffer empty event and a scan event occur simultaneously, the scan event is reported before the buffer empty event.

### 11.1.4 Buffer Full Event

The buffer full event is generated during input operations each time one of the specified input data buffers has been completely filled. If a buffer full event occurs at the same time as the complete event, the buffer full event is reported before the complete event. If a buffer full event and a scan event occur simultaneously, the scan event is reported before the buffer full event.

### 11.1.5 Scan Event

The scan event is generated each time the number of scans specified by the scan_event_level have been completed. If a scan event occurs at the same time as the complete event, the scan event is reported before the complete event. If a scan event and a buffer full or buffer empty event occur simultaneously, the scan event is reported before the buffer full or buffer empty event.

### 11.1.6 User Break Event

The user-break event is generated when a request is aborted as a result of the user-break procedure. The user-break procedure is discussed in section 13.35. A request will never report any events after the user-break event.

### 11.1.7 Time-out Event

The time-out event is generated when a request is aborted because the specified time-out interval was exceeded. A request will never report any events after the time-out event.

### 11.1.8 Run-time Error Event

The run-time error event is generated when an error occurs during the processing of the request. The application can determine the source of the error using the DaqGetRuntimeError procedure. A request will never report any events after the time-out event.

## 11.2 Monitoring Events Using The Request Status

One method of monitoring DAQDRIVE events is through the request_status field in the request structure. When an event occurs during the processing of a request, DAQDRIVE sets the corresponding bit to 1 in the request's request_status field as shown in figure 7. DAQDRIVE does not rely on the information contained in this field nor does it ever clear any of the event bits to 0. Therefore, the application should initialize request_status during the configuration process and may modify its contents at any time.

| DAQDRIVE constant | Value | Description |
|---|---|---|
| NO_EVENTS | 0x00000000 | This constant does not represent an event status. It is provided to the application for convenience. |
| TRIGGER_EVENT | 0x00000001 | When set to 1, this bit indicates the specified trigger has been received. |
| COMPLETE_EVENT | 0x00000002 | When set to 1, this bit indicates the request has completed successfully. |
| BUFFER_EMPTY_EVENT | 0x00000004 | When set to 1, this bit indicates at least one of the output data buffers has been emptied. |
| BUFFER_FULL_EVENT | 0x00000008 | When set to 1, this bit indicates at least one of the input data buffers has been filled. |
| SCAN_EVENT | 0x00000010 | When set to 1, this bit indicates the number of scans specified by scan_event_level have been completed at least once. |
| USER_BREAK_EVENT | 0x20000000 | When set to 1, this bit indicates the request has terminated due to a user-break. |
| TIMEOUT_EVENT | 0x40000000 | When set to 1, this bit indicates the request has terminated because the specified time-out interval was exceeded. |
| RUNTIME_ERROR_EVENT | 0x80000000 | When set to 1, this bit indicates the request has terminated because of an error during processing. The application can determine the source of the error using the DaqGetRuntimeError procedure. |

Figure 7. request_status bit definitions.

```
#include "daqdrive.h"
#include "userdata.h"


unsigned short exit_program;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare a background request.  *****/

my_request.IO_mode        = BACKGROUND_IRQ;
my_request.request_status = NO_EVENTS;

/*****  Request the operation.  *****/

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/

/*****  Define events which will make execution stop.  *****/

exit_program = COMPLETE_EVENT | RUNTIME_ERROR_EVENT | TIMEOUT_EVENT;

/*****  Wait for "exit" event.  *****/

while((my_request.request_status & exit_program) == 0)
   {
   /*****  Wait in dead loop for any event.  *****/

   while(my_request.request_status == NO_EVENTS);

   /*****  Process trigger event.  *****/

   if ((my_request.request_status & TRIGGER_EVENT) != 0)
      {
      printf("Trigger received.\n");
      my_request.request_status &= (~TRIGGER_EVENT);
      }

   /*****  Process scan event.  *****/

   if ((my_request.request_status & SCAN_EVENT) != 0)
      {
      printf("Scan Event.\n");
      my_request.request_status &= (~SCAN_EVENT);
      }
   }

/*****  Indicate time-out error.  *****/

if ((my_request.request_status & TIMEOUT_EVENT) != 0)
   printf("Request aborted.  Time-out error.\n");

/*****  Indicate run-time error.  *****/

if ((my_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   printf("Request aborted.  Run-time error.\n");

/*****  Indicate complete - no errors.  *****/

if ((my_request.request_status & COMPLETE_EVENT) != 0)
   printf("Request completed.\n");

/*****  Release the request (See DaqReleaseRequest).  *****/

/*****  Close the device (See DaqCloseDevice).  *****/
```

## 11.3 Monitoring Events Using Event Notification

Event notification allows the user to define a procedure that DAQDRIVE will execute each time an event occurs. Event notification is especially useful during foreground mode operations when DAQDRIVE has control of the CPU. The event notification procedure is installed using DaqNotifyEvent and should be installed before the request is armed.

```
unsigned short DaqNotifyEvent(unsigned short request_handle,
                              void (far pascal *event_procedure)
                                                (unsigned short,
                                                 unsigned short,
                                                 unsigned short),
                              unsigned long event_mask)
```

The event procedure defined by the application program must be a 'far' pascal compatible procedure of type void. When executed, DAQDRIVE provides the event procedure with the request's request_handle, the type of event which has occurred as shown in figure 8, and an event error code. This error code is set to 0 for all events except the run-time error event where it is used to specify the type of error encountered as defined in chapter 14. Since the request_handle is provided to the event procedure, a single event procedure may service events from multiple requests.

```
void far pascal event_procedure(unsigned short request_handle,
                                unsigned short event_type,
                                unsigned short error_code)
```

The following restrictions apply to the event procedure:

1. only one event procedure may be installed per request.
2. the event procedure can not call any DAQDRIVE procedures.
3. Because the event procedure may be called from within an interrupt service routine (ISR), the event procedure should avoid using BIOS, DOS or Windows system calls.

| DAQDRIVE constant | Value | Description |
|---|---|---|
| EVENT_TYPE_TRIGGER | 0 | This call to the notification procedure is the result of a trigger event. |
| EVENT_TYPE_COMPLETE | 1 | This call to the notification procedure is the result of a complete event. |
| EVENT_TYPE_BUFFER_EMPTY | 2 | This call to the notification procedure is the result of a buffer empty event. |
| EVENT_TYPE_BUFFER_FULL | 3 | This call to the notification procedure is the result of a buffer full event. |
| EVENT_TYPE_SCAN | 4 | This call to the notification procedure is the result of a scan event. |
| EVENT_TYPE_USER_BREAK | 29 | This call to the notification procedure is the result of a user break event. |
| EVENT_TYPE_TIMEOUT | 30 | This call to the notification procedure is the result of a time-out event. |
| EVENT_TYPE_RUNTIME_ERROR | 31 | This call to the notification procedure is the result of a run-time error event. |

Figure 8. event_type definition.

The application may enable or disable the notification of specific events using the bits of the event_mask variable as defined in figure 9. To enable notification of an event, the application need only set the corresponding bit in the event_mask to 1. To disable the notification, the event_mask bit is cleared to 0. Because event_mask is a bit mask, multiple events may be enabled by ORing specific event notification bits.

**IMPORTANT:**
event_mask only controls the notification of events. The request_status field in the request structure is updated regardless of the event_mask settings.

| DAQDRIVE constant | Value | Description |
|---|---|---|
| NO_EVENTS | 0x00000000 | Disable all event notification. |
| TRIGGER_EVENT | 0x00000001 | Enable notification of trigger events. |
| COMPLETE_EVENT | 0x00000002 | Enable notification of complete events. |
| BUFFER_EMPTY_EVENT | 0x00000004 | Enable notification of buffer empty events. |
| BUFFER_FULL_EVENT | 0x00000008 | Enable notification of buffer full events. |
| SCAN_EVENT | 0x00000010 | Enable notification of scan events. |
| USER_BREAK_EVENT | 0x20000000 | Enable notification of user break events. |
| TIMEOUT_EVENT | 0x40000000 | Enable notification of time-out events. |
| RUNTIME_ERROR_EVENT | 0x80000000 | Enable notification of run-time error events. |

Figure 9. event_mask bit definitions.

```
#include "daqdrive.h"
#include "userdata.h"

void far pascal my_event_procedure(unsigned short request_handle,
                                   unsigned short event_type,
                                   unsigned short error_code)
{
switch(event_type)
   {
   case EVENT_TYPE_TRIGGER:
      /*****  process trigger event  *****/
      break;
   case EVENT_TYPE_COMPLETE:
      /*****  process complete event  *****/
      break;
   }
}


void main()
{
unsigned short request_handle;
unsigned short status;
unsigned long  event_mask;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Request an operation. (gets a request_handle)  *****/

/*****  Define events to be notified.  *****/

event_mask = TRIGGER_EVENT | COMPLETE_EVENT;

/*****  Install notification procedure.  *****/

status = DaqNotifyEvent(request_handle, my_event_procedure, event_mask);
if (status != 0)
   printf("Error installing notification.\n");

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/
```

## 11.4 Monitoring Events Using Messages In Windows

DAQDRIVE provides an additional procedure for Windows applications which provides event notification by posting messages to the application window. This procedure, DaqPostMessageEvent, installs a pre-defined messaging procedure using the DaqNotifyEvent mechanism discussed in the previous section. Therefore, DaqPostMessageEvent and DaqNotifyEvent can not both be used on the same request.

```
unsigned short DaqPostMessageEvent(unsigned short request_handle,
                                   unsigned long  event_mask,
                                   unsigned short window_handle)
```

When an event occurs, DAQDRIVE uses the Windows PostMessage procedure to send an event message to the window specified by window_handle. The message number (uMsg) is the sum of the event value specified in figure 8 and the Windows message constant WM_USER. The two message specific arguments, LPARAM and WPARAM, are used to specify the request's request_handle and an event error_code respectively. The error code is set to 0 for all events except the run-time error event where it is used to specify the type of error encountered as defined in chapter 14.

The application may enable or disable the notification of certain events using the bits of the event_mask variable as defined in figure 9. To enable notification of an event, the application need only set the corresponding bit in the event_mask to 1. To disable the notification, the event_mask bit is cleared to 0. Because event_mask is a bit mask, multiple events may be enabled by ORing specific event notification bits.

**IMPORTANT:**
event_mask only controls the notification of events. The request_status field in the request structure is updated regardless of the event_mask settings.

# 12 Common Application Examples

This chapter is dedicated to providing working example programs for some common data acquisition applications. In each of these examples, one data acquisition adapter was selected for the purpose of illustration. All of these examples are written in C using the DOS C-library version of DAQDRIVE. Additional example programs are also provided on the distribution diskette(s) supplied with the data acquisition hardware.

## 12.1 Analog Input (A/D) Examples

### 12.1.1 Example 1

This example inputs a single value to a single A / D channel .

```
// Input a single sample from a single A/D channel

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short channel;
unsigned short status;
short input_value;
float gain;
char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }
do
   {
   /*** Step 2: Get A/D channel and gain ***/

   _clearscreen(_GCLEARSCREEN);
   printf("\n\nEnter a channel number between 0 and 7 or \"99\" to quit:  ");
   scanf("%d", &channel);
   if(channel != 99)
      {
      printf("\n\nEnter a gain of 1, 2, 4, or 8:  ");
      scanf("%f", &gain);

      /*** Step 3: Input value from channel ***/

      status = DaqSingleAnalogInput(logical_device,channel,gain,&input_value);
      if(status != 0)
         printf("\n\nA/D input error.  Status code %d.\n\n", status);
      else
         printf("Channel %d:  %d\n\n",channel, input_value);
      printf("     Press <ESC> to continue.\n");
      while(getch() != 0x1b);
      }
   }
while(channel != 99);

/*** Step 4: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.1.2 Example 2

This example inputs 1000 samples from A/D channel 0 at 100Hz .

```
/*** Input 1000 samples from A/D channel 0 ***/

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daq1200.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct ADC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short i, j;
short input_data[1000];
float gain;
unsigned long event_mask;
char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input data ***/

channel = 0;
gain    = 1;

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = input_data;      /* set pointer to data array     */
my_data.buffer_length  = 1000;            /* number of points in buffer    */
my_data.next_structure = NULL;            /* indicate no more buffers      */
my_data.buffer_status  = BUFFER_EMPTY;    /* indicate buffer empty (ready) */

/*** Prepare the A/D request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels  */
user_request.gain_array_ptr    = &gain;              /* array of gains     */
user_request.array_length      = 1;                  /* number of channels */
user_request.ADC_buffer        = &my_data;           /* pointer to data    */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger   */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* input all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock */
user_request.sample_rate       = 100;                /* 100 Hz input rate  */
user_request.number_of_scans   = 1000;               /* 1000 scans         */
user_request.scan_event_level  = 0;                  /* no scan events     */
user_request.calibration       = NO_CALIBRATION;     /* no calibration     */
```

```
request_handle = 0;                                  /* new request        */
status = DaqAnalogInput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("A/D request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 3: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Trigger the Request ***/

printf("Acquiring data.  This will take 10 seconds.  Please wait.\n");
status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait for event */
if((user_request.request_status & COMPLETE_EVENT) != 0)
   {
   /*** if successful, display data ***/

   for(i = 0; i < 50; i++)
      {
      _clearscreen(_GCLEARSCREEN);
      for(j = 0; j < 20; j++)
         printf("sample #%4d:  value = %6d\n",((i*20)+j),input_data[(i*20)+j]);
      printf("\n  Press <ESC> to continue");
      while(getch() != 0x1b);
      }
   }
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
```

### 12.1.3 Example 3

This example inputs 200 samples each from five A / D channels at 100Hz .

```
/*** Input 200 samples each from A/D channels 0 to 4 ***/

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct ADC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel[5] = {   0,   1,   2,   3,   4 };
float gain[5]             = { 1.0, 1.0, 2.0, 4.0, 1.0 };
unsigned short status;
unsigned short i, j, k;
short input_data[200][5];
unsigned long  event_mask;
char far *device_type = "DAQP-208";
char far *config_file = "daqp-208.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input data ***/

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = input_data;      /* set pointer to data array     */
my_data.buffer_length  = 1000;            /* number of points in buffer    */
my_data.next_structure = NULL;            /* indicate no more buffers      */
my_data.buffer_status  = BUFFER_EMPTY;    /* indicate buffer empty (ready) */

/*** Prepare the A/D request structure ***/

user_request.channel_array_ptr = channel;            /* array of channels  */
user_request.gain_array_ptr    = gain;               /* array of gains     */
user_request.array_length      = 5;                  /* number of channels */
user_request.ADC_buffer        = &my_data;           /* pointer to data    */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger   */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* input all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock */
user_request.sample_rate       = 100;                /* 100 Hz input rate  */
user_request.number_of_scans   = 200;                /* 200 scans          */
user_request.scan_event_level  = 0;                  /* no scan events     */
user_request.calibration       = NO_CALIBRATION;     /* no calibration     */
user_request.timeout_interval  = 0;                  /* disable time-out   */
user_request.request_status    = 0;                  /* initialize status  */

      t_b_dl    0                                    /*           t       */
```

```
status = DaqAnalogInput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("A/D request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }
/*** Step 3: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
/*** Step 4: Trigger the Request ***/

printf("Acquiring data.  This will take 2 seconds.  Please wait.\n");
status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
/*** Step 5: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait for event */
if((user_request.request_status & COMPLETE_EVENT) != 0)
   {
   /*** if successful, display data ***/

   for(i = 0; i < 10; i++)
      {
      _clearscreen(_GCLEARSCREEN);
      for(j = 0; j < 20; j++)
         {
         printf("sample #%4d: ", ((i * 20) + j));
         for(k = 0; k < 5; k++)
            printf("   %6d", input_data[(i * 20) + j][k];
         printf("\n");
         }
      printf("\n  Press <ESC> to continue");
      while(getch() != 0x1b);
      }
   }
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
/*** Step 6: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }
/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.1.4 Example 4

This example simulates a volt meter operation reading 20 samples from A/D channel 0 s at 1Hz using only one data memory location .

```
/*** Input 20 samples from A/D channel 0 ***/

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct ADC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
short current_sample;
float gain;
unsigned long  event_mask;
char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input data ***/

channel = 0;
gain    = 2;

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = &current_sample; /* set pointer to data storage   */
my_data.buffer_length  = 1;               /* number of points in buffer    */
my_data.next_structure = NULL;            /* indicate no more buffers      */
my_data.buffer_status  = BUFFER_EMPTY;    /* indicate buffer empty (ready) */

/*** Prepare the A/D request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels  */
user_request.gain_array_ptr    = &gain;              /* array of gains     */
user_request.array_length      = 1;                  /* number of channels */
user_request.ADC_buffer        = &my_data;           /* pointer to data    */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger   */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* input all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock */
user_request.sample_rate       = 1;                  /* 1 Hz input rate    */
user_request.number_of_scans   = 20;                 /* 20 scans           */
user_request.scan_event_level  = 0;                  /* no scan events     */
user_request.calibration       = NO_CALIBRATION;     /* no calibration     */
```

```
user_request.timeout_interval  = 0;                 /* disable time-out   */
user_request.request_status    = 0;                 /* initialize status  */
request_handle = 0;                                 /* new request        */
status = DaqAnalogInput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("A/D request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 3: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Trigger the Request ***/

status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Wait for completion or error ***/

_clearscreen(_GCLEARSCREEN);
event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
do
   {
   if((user_request.request_status & BUFFER_FULL_EVENT) != 0)
      {
      _settextposition(10,10);
      printf("The current value is %6d", current_sample);
      my_data.buffer_status = BUFFER_EMPTY;         /* buffer empty (ready) */
      user_request.request_status &= (~BUFFER_FULL_EVENT);
      }
   }
while((user_request.request_status & event_mask ) == 0);  /* wait for event */

if((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   DaqGetRuntimeError(request_handle, &status)
   printf("\n\n\nRun-time error. Error code %d. Operation aborted.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
```

### 12.1.5 Example 5

This example inputs 100,000 samples from A/D channel 0 and stores the data in a disk file.

```
/*** Input 100,000 samples from A/D channel 0 and write to disk ***/

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data[4];
struct ADC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short current_buffer;
unsigned short i;
short buffer[4][1000];
float gain;
unsigned long  event_mask;
char far *device_type = "DAQP-208";
char far *config_file = "daqp-208.dat";
FILE *output_file;

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input data ***/

channel = 0;
gain    = 2;

/*** Prepare Buffer Structures ***/

my_data[0].data_buffer    = &buffer[0][0];   /* set pointer to data array   */
my_data[0].buffer_length  = 1000;            /* number of points in buffer  */
my_data[0].next_structure = &my_data[1];     /* point to next buffer        */
my_data[0].buffer_status  = BUFFER_EMPTY;    /* buffer empty (ready)        */

my_data[1].data_buffer    = &buffer[1][0];   /* set pointer to data array   */
my_data[1].buffer_length  = 1000;            /* number of points in buffer  */
my_data[1].next_structure = &my_data[2];     /* point to next buffer        */
my_data[1].buffer_status  = BUFFER_EMPTY;    /* buffer empty (ready)        */

my_data[2].data_buffer    = &buffer[2][0];   /* set pointer to data array   */
my_data[2].buffer_length  = 1000;            /* number of points in buffer  */
my_data[2].next_structure = &my_data[3];     /* point to next buffer        */
my_data[2].buffer_status  = BUFFER_EMPTY;    /* buffer empty (ready)        */
```

```
my_data[3].data_buffer    = &buffer[3][0];   /* set pointer to data array   */
my_data[3].buffer_length  = 1000;            /* number of points in buffer  */
my_data[3].next_structure = &my_data[0];     /* point to next buffer        */
my_data[3].buffer_status  = BUFFER_EMPTY;    /* buffer empty (ready)        */

/*** Prepare the A/D request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels  */
user_request.gain_array_ptr    = &gain;              /* array of gains     */
user_request.array_length      = 1;                  /* number of channels */
user_request.ADC_buffer        = &my_data[0];        /* pointer to data    */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger   */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* input all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock */
user_request.sample_rate       = 1000;               /* 1 KHz input rate   */
user_request.number_of_scans   = 100000;             /* 100000 scans       */
user_request.scan_event_level  = 0;                  /* no scan events     */
user_request.calibration       = NO_CALIBRATION;     /* no calibration     */
user_request.timeout_interval  = 0;                  /* disable time-out   */
user_request.request_status    = 0;                  /* initialize status  */

request_handle = 0;                                  /* new request        */
status = DaqAnalogInput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("A/D request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 3: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Open a data file ***/

output_file = fopen("ADC_DATA.ASC","w");

/*** Step 5: Trigger the Request ***/

printf("Acquiring data.  This will take 100 seconds.  Please wait.\n");
status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   fclose(output_file);
   exit(status);
   }

/*** Step 6: Wait for completion or error ***/

current_buffer = 0;
event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
do
   {
   if((my_data[current_buffer].buffer_status == BUFFER_FULL)
      {
      for(i = 0; i < 1000; i++)
         fprintf(output_file,"%6d\n",buffer[current_buffer][i]);
      my_data[current_buffer].buffer_status = BUFFER_EMPTY;  /* buffer empty */
```

```
      if(current_buffer == 3)
         current_buffer = 0;
      else
         current_buffer++;
      }
   }
while((user_request.request_status & event_mask ) == 0);  /* wait for event */

/*** Exit if error ***/

if((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   DaqGetRuntimeError(request_handle, &status);
   printf("\n\n\nRun-time error. Error code %d. Operation aborted.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   fclose(output_file);
   exit(status);
   }

/*** Write any remaining buffers and close file ***/

do
   {
   if (my_data[current_buffer].buffer_status == BUFFER_FULL)
      {
      for(i = 0; i < 1000; i++)
         fprintf(output_file,"%6d\n",buffer[current_buffer][i]);
      my_data[current_buffer].buffer_status = BUFFER_EMPTY;  /* buffer empty */
      }
   if(current_buffer == 3)
      current_buffer = 0;
   else
      current_buffer++;
   }
while(current_buffer != 0);
fclose(output_file);

/*** Step 6: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

## 12.2 Analog Output (D/A) Examples

### 12.2.1 Example 1

This example outputs a single value to a single D / A channel .

```
/*** Output a single point to a single D/A channel. ***/

#include <conio.h>
#include <graph.h>
#include <stdio.h>
#include <stdlib.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned short channel;
short output_value;
char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

do
   {
   /*** Step 2: Get D/A channel and output value ***/

   _clearscreen(_GCLEARSCREEN);
   printf("\n\nEnter a channel number between 0 and 7 or \"99\" to quit:  ");
   scanf("%d", &channel);
   if(channel != 99)
      {
      printf("\n\nEnter the output value between -2048 and 2047:  ");
      scanf("%d", &output_value);

      /*** Step 3: Output value to channel ***/

      status = DaqSingleAnalogOutput(logical_device, channel, &output_value);
      if(status != 0)
         {
         printf("\n\n     D/A output error.  Status code %d.\n\n", status);
         printf("     Press <ESC> to continue.\n");
         while(getch() != 0x1b);
         }
      }
   }
while(channel != 99);

/*** Step 4: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.2.2 Example 2

This example outputs a single value to a single D / A channel using a TTL trigger.

```
/*** Output a single point to a single D/A channel on an external trigger ***/

#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "da8p-12.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct DAC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;
unsigned short channel;
short output_value;
unsigned long event_mask;
char far *device_type = "DA8P-12B";
char far *config_file = "da8p-12b.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

do
   {
   /*** Step 2: Get D/A channel and output value ***/

   _clearscreen(_GCLEARSCREEN);
   printf("\n\nEnter a channel number between 0 and 7 or \"99\" to quit:  ");
   scanf("%d", &channel);

   if( channel != 99 )
      {
      printf("\n\nEnter the output value between -2048 and 2047:  ");
      scanf("%d", &output_value);

      /*** Step 3: Output data ***/

      /*** Prepare Buffer Structure ***/

      my_data.data_buffer    = &output_value;   /* point to output data   */
      my_data.buffer_length  = 1;               /* number of points       */
      my_data.buffer_cycles  = 1;               /* cycle buffer once      */
      my_data.next_structure = NULL;            /* no more buffers        */
      my_data.buffer_status  = BUFFER_FULL;     /* buffer full (ready)    */

      /*** Prepare the D/A request structure ***/

      user_request.channel_array_ptr = &channel;       /* array of channels  */
      user_request.array_length      = 1;              /* number of channels */
```

```
      user_request.DAC_buffer        = &my_data;       /* pointer to data    */
      user_request.trigger_source    = TTL_TRIGGER;    /* select TTL trigger */
      user_request.trigger_slope     = RISING_EDGE;    /* rising edge trigger*/
      user_request.IO_mode           = BACKGROUND_IRQ; /* background mode    */
      user_request.number_of_scans   = 1;              /* scan channela once */
      user_request.scan_event_level  = 0;              /* no scan events     */
      user_request.calibration       = NO_CALIBRATION; /* no calibration     */
      user_request.timeout_interval  = 0;              /* disable time-out   */
      user_request.request_status    = 0;              /* initialize status  */

      request_handle = 0;                              /* new request        */
      status = DaqAnalogOutput(logical_device,&user_request,&request_handle );
      if(status != 0)
         {
         printf("D/A request error.  Status code %d.\n", status);
         DaqCloseDevice(logical_device);
         exit(status);
         }

      /*** Step 4: Arm the Request ***/

      status = DaqArmRequest(request_handle);
      if(status != 0)
         {
         printf("Arm request error.  Status code %d.\n", status);
         DaqReleaseRequest(request_handle);
         DaqCloseDevice(logical_device);
         exit(status);
         }

      /*** Step 5: Wait for completion or error ***/

      printf("\n\n\n     Waiting for trigger ...");
      event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
      while((user_request.request_status & event_mask ) == 0 );  /* wait for */
                                                                 /* event    */
      if(( user_request.request_status & COMPLETE_EVENT ) != 0 )
         printf("\n\n D/A Output Request complete.\n");
      else
         {
         printf("Run-time error. Operation aborted.\n");
         DaqReleaseRequest(request_handle);
         DaqCloseDevice(logical_device);
         exit(status);
         }

      /*** Step 6: Release the Request ***/

      status = DaqReleaseRequest(request_handle);
      if(status != 0)
         {
         printf("Could not release configuration.  Status code %d.\n", status);
         exit(status);
         }
      }
   }
while(channel != 99);

/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.2.3 Example 3

This example outputs 1000 cycles of a 60 Hz sinewave to D/A channel 0. The sinewave contains 60 points per cycle.

```
/*** Output a sinewave to D/A channel 0 ***/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct DAC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short i;
short sinewave[60];
unsigned long  event_mask;
char far *device_type = "DAQP-208";
char far *config_file = "daqp-208.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Get D/A channel and output values ***/

channel = 0;
for(i = 0; i < 60; i++)
   sinewave[i] = (short)(2047 * sin((2 * 3.1416 * i) / 60));

/*** Step 3: Output data ***/

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = sinewave;        /* set pointer to output data   */
my_data.buffer_length  = 60;              /* number of points in buffer   */
my_data.buffer_cycles  = 1000;            /* 1000 cycles through buffer   */
my_data.next_structure = NULL;            /* indicate no more buffers     */
my_data.buffer_status  = BUFFER_FULL;     /* indicate buffer full (ready) */

/*** Prepare the D/A request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels   */
user_request.array_length      = 1;                  /* number of channels  */
user_request.DAC_buffer        = &my_data;           /* pointer to data     */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger    */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* output all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background  mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock  */
```

```
user_request.number_of_scans   = 60l * 1000l;        /* 60000 scans         */
user_request.scan_event_level  = 0;                 /* no scan events      */
user_request.calibration       = NO_CALIBRATION;    /* no calibration      */
user_request.timeout_interval  = 0;                 /* disable time-out    */
user_request.request_status    = 0;                 /* initialize status   */

request_handle = 0;                                 /* new request         */
status = DaqAnalogOutput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("D/A request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Trigger the Request ***/

status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait for event */
if(( user_request.request_status & COMPLETE_EVENT ) != 0 )
   printf("\n\n D/A Output Request complete.\n");
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 7: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 8: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
       printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.2.4 Example 4

This example outputs 600 cycles of a sinewave, 300 cycles of a ramp, and 18000 cycles of a square wave to D / A channel 0.

```
/*** Output sine, ramp, and square waves to D/A channel 0 ***/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "da8p-12.h"

/*** When defined global or static, structures are initialized to all 0  ***/

struct DAQDRIVE_buffer  my_data[3];
struct DAC_request      user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short i;
short sinewave[60];
short ramp[120];
short square[2];
unsigned long  event_mask;
char far *device_type = "DA8P-12B";
char far *config_file = "da8p-12b.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Define the output channel and data values ***/

channel = 0;
for(i = 0; i < 60; i++)
   sinewave[i] = (short)(2047 * sin((2 * 3.1416 * i) / 60));

for(i = 0; i < 120; i++)
   ramp[i] = (short)((2047.0 * i) / 119);

square[0] = -2048;
square[1] =  2047;

/*** Step 3: Output data ***/
/*** Prepare Buffer Structures ***/

my_data[0].data_buffer    = sinewave;        /* set pointer to output data   */
my_data[0].buffer_length  = 60;              /* number of points in buffer   */
my_data[0].buffer_cycles  = 600;             /* 600 cycles through buffer    */
my_data[0].next_structure = &my_data[1];     /* point to next buffer         */
my_data[0].buffer_status  = BUFFER_FULL;     /* indicate buffer full (ready) */

my_data[1].data_buffer    = ramp;            /* set pointer to output data   */
my_data[1].buffer_length  = 120;             /* number of points in buffer   */
my_data[1].buffer_cycles  = 300;             /* 300 cycles through buffer    */
my_data[1].next_structure = &my_data[2];     /* point to next buffer         */
my_data[1].buffer_status  = BUFFER_FULL;     /* indicate buffer full (ready) */

my_data[2].data_buffer    = square;          /* set pointer to output data   */
my_data[2].buffer_length  = 2;               /* number of points in buffer   */
my_data[2].buffer_cycles  = 18000;           /* 18000 cycles through buffer  */
my_data[2].next_structure = NULL;            /* mo more buffers              */
```

```
/*** Prepare the D/A request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels    */
user_request.array_length      = 1;                  /* number of channels   */
user_request.DAC_buffer        = &my_data[0];        /* pointer to data      */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger     */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* output all points    */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode      */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock   */
user_request.sample_rate       = 3600;               /* 3600 points / second */
user_request.number_of_scans   =  60l *   600l       /* 36000 scans of sine  */
                               + 120l *   300l       /* 36000 scans of ramp  */
                               +   2  * 18000l;      /* 36000 scans of square*/
user_request.scan_event_level  = 0;                  /* no scan events       */
user_request.calibration       = NO_CALIBRATION;     /* no calibration       */
user_request.timeout_interval  = 0;                  /* disable time-out     */
user_request.request_status    = 0;                  /* initialize status    */

request_handle = 0;                                  /* new request          */
status = DaqAnalogOutput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("D/A request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Trigger the Request ***/

status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait or event */
if(( user_request.request_status & COMPLETE_EVENT ) != 0 )
   printf("\n\n D/A Output Request complete.\n");
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
/*** Step 7: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 8: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
```

## 12.3 Digital Input Examples

### 12.3.1 Example 1

This example inputs a single value from a single digital input channel .

```
/*** Input a single point from a single digital input channel ***/

#include <conio.h>
#include <graph.h>
#include <stdio.h>
#include <stdlib.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned short channel;
unsigned char input_value;
char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

do
   {
   /*** Step 2: Get digital input channel ***/

   _clearscreen(_GCLEARSCREEN);
   printf("\n\nEnter a digital input channel number or \"99\" to quit:  ");
   scanf("%d", &channel);

   if(channel != 99)
      {
      /*** Step 3: Input value from channel ***/

      status = DaqSingleDigitalInput(logical_device, channel, &input_value);
      if(status != 0)
         printf("\n\nDigital input error.  Status code %d.\n\n", status);
      else
         printf("Channel %d:  %xH\n\n",channel, (int)input_value);
      printf("     Press <ESC> to continue.\n");
      while(getch() != 0x1b);
      }
   }
while(channel != 99);

/*** Step 4: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.3.2 Example 2

This example inputs 1000 samples from digital I/O channel 0.

```
/*** Input 1000 samples from digital input channel 0 ***/

#include <conio.h>
#include <graph.h>
#include <stdlib.h>
#include <stdio.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "iop241.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct digio_request    user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short i, j;
unsigned char input_data[1000];
unsigned long event_mask;
char far *device_type = "IOP-241";
char far *config_file = "iop-241.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(IOP241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Input data ***/

channel = 0;

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = input_data;      /* set pointer to output data    */
my_data.buffer_length  = 1000;            /* number of points in buffer    */
my_data.next_structure = NULL;            /* indicate no more buffers      */
my_data.buffer_status  = BUFFER_EMPTY;    /* indicate buffer empty (ready) */

/*** Prepare the digital input request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels  */
user_request.array_length      = 1;                  /* number of channels */
user_request.digio_buffer      = &my_data;           /* pointer to data    */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger   */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* input all points   */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode    */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock */
user_request.sample_rate       = 100;                /* 100 Hz input rate  */
user_request.number_of_scans   = 1000;               /* 1000 scans         */
user_request.scan_event_level  = 0;                  /* no scan events     */
user_request.timeout_interval  = 0;                  /* disable time-out   */
user_request.request_status    = 0;                  /* initialize status  */
```

```
request_handle = 0;                                  /* new request        */
status = DaqDigitalInput(logical_device, &user_request, &request_handle);
if(status != 0)
   {
   printf("Digital input request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 3: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Trigger the Request ***/

status = DaqTriggerRequest(request_handle);
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait for event */
if((user_request.request_status & COMPLETE_EVENT) != 0)
   {
   /*** if successful, display data ***/

   for(i = 0; i < 50; i++)
      {
      _clearscreen(_GCLEARSCREEN);
      for(j = 0; j < 20; j++)
         printf("sample #%4d:  value = %2xH\n", ((i*20)+j),
                                                (int)(input_data[(i*20)+j]));
      printf("\n  Press <ESC> to continue");
      while(getch() != 0x1b);
      }
   }
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 7: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
```

## 12.4 Output Examples

### 12.4.1 Example 1

This example outputs a single value to a single digital output channel .

```
/*** Output a single point to a single D/A channel ***/

#include <conio.h>
#include <graph.h>
#include <stdio.h>
#include <stdlib.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;
unsigned short channel;
unsigned char output_value;
char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

do
   {
   /*** Step 2: Get digital output channel and output value ***/

   _clearscreen(_GCLEARSCREEN);
   printf("\n\nEnter a digital output channel number or \"99\" to quit:  ");
   scanf("%d", &channel);

   if(channel != 99)
      {
      printf("\n\nEnter the output value between 0 and 255:  ");
      scanf("%d", &output_value);

      /*** Step 3: Output value to channel ***/

      status = DaqSingleDigitalOutput(logical_device, channel, &output_value);
      if(status != 0)
         {
         printf("\n\n Digital output error.  Status code %d.\n\n", status);
         printf(" Press <ESC> to continue.\n");
         while(getch() != 0x1b);
         }
      }
   }
while(channel != 99);

/*** Step 4: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

### 12.4.2 Example 2

This example outputs a 20 point pattern to digital output channel 0.

```
/*** Output a pattern to digital output channel 0 ***/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "userdata.h"
#include "daqdrive.h"
#include "daqopenc.h"
#include "da8p-12.h"

/*** When defined global or static, structures are ***/
/*** automatically initialized to all 0            ***/

struct DAQDRIVE_buffer  my_data;
struct digio_request    user_request;

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned char pattern[20] = { 0, 1, 1, 0, 1,
                              1, 1, 1, 0, 1,
                              0, 1, 0, 0, 0,
                              1, 1, 0, 1, 1  };
unsigned long event_mask;
char far *device_type = "DA8P-12B";
char far *config_file = "da8p-12b.dat";

/*** Step 1: Initialize Hardware ***/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening device.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 2: Output data ***/

channel = 0;

/*** Prepare Buffer Structure ***/

my_data.data_buffer    = pattern;         /* set pointer to output data   */
my_data.buffer_length  = 20;              /* number of points in buffer   */
my_data.buffer_cycles  = 500;             /* 500 cycles through buffer    */
my_data.next_structure = NULL;            /* indicate no more buffers     */
my_data.buffer_status  = BUFFER_FULL;     /* indicate buffer full (ready) */

/*** Prepare the digital output request structure ***/

user_request.channel_array_ptr = &channel;           /* array of channels    */
user_request.array_length      = 1;                  /* number of channels   */
user_request.digio_buffer      = &my_data;           /* pointer to data      */
user_request.trigger_source    = INTERNAL_TRIGGER;   /* internal trigger     */
user_request.trigger_mode      = CONTINUOUS_TRIGGER; /* output all points    */
user_request.IO_mode           = BACKGROUND_IRQ;     /* background mode      */
user_request.clock_source      = INTERNAL_CLOCK;     /* use on-board clock   */
user_request.sample_rate       = 200;                /* 10 patterns / second */
user_request.number_of_scans   = 20 * 500;           /* 10000 scans          */
user_request.scan_event_level  = 0;                  /* no scan events       */
user_request.timeout_interval  = 0;                  /* disable time-out     */
```

```
request_handle = 0;                                  /* new request       */
status = DaqDigitalOutput( logical_device, &user_request, &request_handle );
if(status != 0)
   {
   printf("Digital output request error.  Status code %d.\n", status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 4: Arm the Request ***/

status = DaqArmRequest(request_handle);
if(status != 0)
   {
   printf("Arm request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 5: Trigger the Request ***/

status = DaqTriggerRequest( request_handle );
if(status != 0)
   {
   printf("Trigger request error.  Status code %d.\n", status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 6: Wait for completion or error ***/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask ) == 0 );  /* wait for event */
if(( user_request.request_status & COMPLETE_EVENT ) != 0 )
   printf("\n\n D/A Output Request complete.\n");
else
   {
   printf("Run-time error. Operation aborted.\n");
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*** Step 7: Release the Request ***/

status = DaqReleaseRequest(request_handle);
if(status != 0)
   {
   printf("Could not release configuration.  Status code %d.\n", status);
   exit(status);
   }

/*** Step 8: Close Hardware Device ***/

status = DaqCloseDevice(logical_device);
if(status != 0)
   printf("Error closing device.  Status code %d.\n", status);
return(status);
}
```

# 13 Command Reference

**Analog Input**
DaqAnalogInput
DaqSingleAnalogInput
DaqSingleAnalogInputScan
DaqSingleSigConInput
DaqSingleSigConInputScan

**Analog Output**
DaqAnalogOutput
DaqSingleAnalogOutput
DaqSingleAnalogOutputScan

**Digital Input**
DaqDigitalInput
DaqSingleDigitalInput
DaqSingleDigitalInputScan

**Digital Output**
DaqDigitalOutput
DaqSingleDigitalOutput
DaqSingleDigitalOutputScan

**Process Control**
DaqArmRequest
DaqCloseDevice
DaqOpenDevice
DaqReleaseRequest
DaqResetDevice
DaqStopRequest
DaqTriggerRequest

**System Configuration**
DaqGetADCfgInfo
DaqGetADGainInfo
DaqGetDACfgInfo
DaqGetDAGainInfo
DaqGetDeviceCfgInfo
DaqGetDigioCfgInfo
DaqGetExpCfgInfo
DaqGetExpGainInfo
DaqGetSigConCfgInfo
DaqGetSigConParamInfo

**Data Processing**
DaqConvertBuffer
DaqConvertPoint
DaqConvertScan

**System Monitoring**
DaqGetRuntimeError
DaqNotifyEvent
DaqPostMessageEvent
DaqUserBreak

**Memory Allocation**
DaqAllocateMemory
DaqAllocateMemory32
DaqAllocateRequest
DaqFreeMemory
DaqFreeMemory32
DaqFreeRequest

**Miscellaneous**
DaqBytesToWords
DaqGetAddressOf
DaqVersionNumber
DaqWordsToBytes

## 13.1 DaqAllocateMemory (16-bit DAQDRIVE only)

DaqAllocateMemory is a DAQDRIVE utility function used to dynamically allocate memory for use by the application program. All memory is allocated from the global (far) heap and should be de-allocated using the DaqFreeMemory procedure before the application terminates. In the 16-bit DLL version of DAQDRIVE, DaqAllocateMemory performs a GlobalLock and a GlobalPageLock on the allocated memory. This allows the memory to be used within DAQDRIVE's interrupt service routines.

**NOTE:**
32-bit application programs must use the DaqAllocateMemory32 procedure.

```
unsigned short DaqAllocateMemory(unsigned long memory_size,
                                 unsigned short far *memory_handle,
                                 void far *(far *memory_pointer))
```

memory_size - This unsigned long integer value is used to specify the amount of memory required by the application.

memory_handle - This unsigned short integer pointer defines the address of a variable where the handle associated with this allocation will be stored. The application must preserve this handle for later use by the DaqFreeMemory procedure.

memory_pointer - This void pointer defines the address of a pointer variable where the starting address of the newly allocated memory block will be stored. The memory is allocated in a manner that makes memory_pointer compatible with all data types including huge. The application must preserve this pointer for later use by the DaqFreeMemory procedure.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[2]   = { 0, 1 };
         float gain_settings[2] = { 1, 8 };
unsigned long memory_size;
unsigned short memory_handle;
void far *memory_pointer;
unsigned short status;

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Allocate memory for the input data.  *****/
/*****  5000 samples/channel * 2 channels * 2 bytes/sample  *****/

memory_size = 5000 * 2 * sizeof(short);
status = DaqAllocateMemory(memory_size, &memory_handle, &memory_pointer);
if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Prepare data structure for analog input.  *****/

data_structure.data_buffer    = memory_pointer;
data_structure.buffer_length  = 10000;

/*****  Prepare the A/D request structure.  *****/

/*****  Request A/D input.  *****/

request_handle = 0;
status = DaqAnalogInput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("A/D request error.  Status code %d.\n",status);
   DaqFreeMemory(memory_handle, memory_pointer);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/

/*****  Wait for complete.  *****/

/*****  Free allocated memory.  *****/

status = DaqFreeMemory(memory_handle, memory_pointer);
if (status != 0)
   {
   printf("Error de-allocating memory.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Close the device.  (See DaqCloseDevice).  *****/
}
```

## 13.2 DaqAllocateMemory32 (32-bit DAQDRIVE only)

DaqAllocateMemory32 is a DAQDRIVE utility function used to dynamically allocate the locked memory required for use by 32-bit application programs. All memory is allocated from the global (far) heap and should be de-allocated using the DaqFreeMemory32 procedure before the application terminates. 32-bit applications that use any of DAQDRIVE's background modes must use this procedure to allocate the request structure, channel and gain arrays, data buffer structures, and data buffers.

**NOTE:**
For 16-bit Windows and all DOS applications, the DaqAllocateMemory function must be used.

```
unsigned short DaqAllocateMemory32(unsigned long memory_size,
                                   unsigned long far *memory_handle,
                                   void far *(far *memory_pointer))
```

memory_size - This unsigned long integer value is used to specify the amount of memory required by the application.

memory_handle - This unsigned long integer pointer defines the address of a variable where the handle associated with this allocation will be stored. The application must preserve this handle for later use by the DaqFreeMemory32 procedure.

memory_pointer - This void pointer defines the address of a pointer variable where the starting address of the newly allocated memory block will be stored. The memory is allocated in a manner that makes memory_pointer compatible with all data types. The application must preserve this pointer for later use by the DaqFreeMemory32 procedure.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned long  memory_handle[3]
unsigned short status;

struct ADC_request     far *user_request;
struct DAQDRIVE_buffer far *data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Allocate memory for the ADC request and data structures.  *****/

status = DaqAllocateMemory32(sizeof(ADC_request),
                             &memory_handle[0],
                             (void far*)&user_request);
if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

status = DaqAllocateMemory32(sizeof(DAQDRIVE_buffer),
                             &memory_handle[1],
                             (void far*)&data_structure);
if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Allocate memory for the input data.  *****/
/*****  5000 samples/channel * 2 channels * 2 bytes/sample  *****/

status = DaqAllocateMemory32(5000 * 2 * sizeof(short),
                             &memory_handle[2],
                             (void far*)&data_structure->data_buffer);

if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Prepare data structure for analog input.  *****/

/*****  Prepare the A/D request structure.  *****/

/*****  Request A/D input.  *****/

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/

/*****  Wait for complete.  *****/

/*****  Free allocated memory.  (See DaqFreeMemory32)  *****/

/*****  Close the device.  (See DaqCloseDevice).  *****/
}
```

## 13.3 DaqAllocateRequest

DaqAllocateRequest is a DAQDRIVE utility function used to dynamically allocate all of the structures and buffers required for a standard request. These include the request structure, channel and gain arrays, data buffer structures, and data buffers. All memory is allocated from the global (far) heap and should be de-allocated using the DaqFreeRequest procedure before the application terminates.

```
unsigned short DaqAllocateRequest(unsigned short logical_device,
                                  struct allocate_request far *user_request)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

user_request - This structure pointer defines the address of a memory allocation request structure containing the desired configuration information for this operation. This structure is discussed in detail on the following pages.

**NOTE:**

In addition to allocating the memory required for the request, DaqAllocateRequest initializes any pointers which reference other fields allocated by the same request. For example, if an analog input request is allocated, the ADC_request structure's channel_array_ptr field will be initialized to point to the allocated channel array, its gain_array_ptr field will be initialized to point to the allocated gain array, and its ADC_buffer field will be initialized to point to the first data buffer structure. Furthermore, if more than one data buffer is requested, each data buffer structure will be initialized such that the next_structure field is pointed to the subsequent data buffer structure to form a 'chain' of data buffers.

```
struct allocate_request
   {
   unsigned long  request_type;
   unsigned short channel_array_length;
   unsigned short number_of_buffers;
   unsigned long  buffer_size;
   unsigned long  buffer_attributes;
   void far      *memory_pointer;
   unsigned long  memory_handle;
   };
```

Figure 10. Analog input request structure.

<table>
<tr><td rowspan="6">request_type</td><td colspan="3">Specifies the type of DAQDRIVE request to allocate memory for.</td></tr>
<tr><td>DAQDRIVE Constant</td><td>Value</td><td>Description</td></tr>
<tr><td>ADC_TYPE_REQUEST</td><td>0</td><td>allocate memory required for an analog input request</td></tr>
<tr><td>DAC_TYPE_REQUEST</td><td>1</td><td>allocate memory required for an analog output request</td></tr>
<tr><td>DIGIN_TYPE_REQUEST</td><td>2</td><td>allocate memory required for a digital input request</td></tr>
<tr><td>DIGOUT_TYPE_REQUEST</td><td>3</td><td>allocate memory required for a digital output request</td></tr>
<tr><td>channel_array_length</td><td colspan="3">This unsigned short integer value defines the required length of the channel and gain arrays (a gain array is only required for analog input requests). The channel array and gain array pointers in the request structure will be initialized to point to the allocated arrays.</td></tr>
<tr><td>number_of_buffers</td><td colspan="3">This unsigned short integer value specifies the number of data buffers and data buffer structures to be allocated for this request. Each buffer structure will have the data_buffer field initialized to point to the actual input/output data buffer and the next_structure field initialized according to the value of buffer_attribute.</td></tr>
<tr><td>buffer_size</td><td colspan="3">This unsigned short integer value specifies the number of bytes in each of the allocated data buffers. All allocated data buffers are the same size.</td></tr>
<tr><td rowspan="4">buffer_attributes</td><td colspan="3">Specifies how the next_structure field in the last DAQDRIVE_buffer structure is initialized.</td></tr>
<tr><td>DAQDRIVE Constant</td><td>Value</td><td>Description</td></tr>
<tr><td>SEQUENTIAL_BUFFER</td><td>0</td><td>The next structure field in the last DAQDRIVE_buffer structure is set to NULL (there are no more structures in the chain).</td></tr>
<tr><td>RING_BUFFER</td><td>1</td><td>The next structure field in the last DAQDRIVE_buffer structure is set to point to the first DAQDRIVE_buffer structure thus creating a ring or circular buffer.</td></tr>
<tr><td>memory_pointer</td><td colspan="3">This void pointer defines the address of a pointer variable where the starting address of the newly allocated memory block will be stored. memory_pointer will specify the address of a request structure of the type specified by the request_type parameter. The application must preserve this pointer for later use by the DaqFreeRequest procedure.</td></tr>
<tr><td>memory_handle</td><td colspan="3">This unsigned long integer variable is used to store the handle associated with this allocation. The application must preserve this handle for later use by the DaqFreeRequest procedure.</td></tr>
</table>

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;

struct ADC_request far *myADCrequest;
struct allocate_request memory_request;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Allocate memory for the ADC request and data structures.  *****/

memory_request.request_type         = ADC_TYPE_REQUEST;
memory_request.channel_array_length = 2;
memory_request.number_of_buffers    = 1;
memory_request.buffer_length        = 2 * 5000 * sizeof(short);
memory_request.buffer_attributes    = SEQUENTIAL_BUFFER;

status = DaqAllocateRequest(logical_device, &memory_request);
if (status != 0)
   {
   printf("Error allocating memory.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

myADCrequest = (ADC_request far *)memory_request.memory_pointer;

myADCrequest->channel_array_pointer[0] = 4;
myADCrequest->channel_array_pointer[1] = 9;
myADCrequest->trigger_source           = INTERNAL_TRIGGER;

/*****  Prepare remainder of A/D request structure.  *****/

/*****  Only status needs initialized in DAQDRIVE buffer structure.  *****/

myADCrequest->ADC_buffer->buffer_status = BUFFER_EMPTY;

/*****  Request A/D input.  *****/
/*****  Arm the request (See DaqArmRequest).  *****/
/*****  Trigger the request (See DaqTriggerRequest).  *****/
/*****  Wait for complete.  *****/
/*****  Free allocated memory.  (See DaqFreeRequest)  *****/

status = DaqFreeRequest(logical_device,
                        memory_request.memory_handle,
                        memory_request.memory_pointer)

if (status != 0)
   printf("Error de-allocating memory.  Status code %d.\n",status);

/*****  Close the device.  (See DaqCloseDevice).  *****/
}
```

## 13.4 DaqAnalogInput

DaqAnalogInput is DAQDRIVE's generic A/D converter interface. It does not configure any hardware but acts simply to confirm that all parameters are valid and that the type of operation requested is supported by the target hardware.

```
unsigned short DaqAnalogInput(unsigned short logical_device,
                              struct ADC_request far *user_request,
                              unsigned short far *request_handle)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

user_request - This structure pointer defines the address of an A/D request structure containing the desired configuration information for this operation. This structure is discussed in detail on the following pages.

request_handle - This unsigned short integer pointer is used to identify this analog input request. For a new configuration, request_handle is set to 0 by the application before calling DaqAnalogInput. If the configuration is successful request_handle will be assigned a unique non-zero value by the DaqAnalogInput procedure. If the application modifies a previously configured request, the application must call DaqAnalogInput using the previously assigned request_handle. All parameters except the channel list may be modified using a reconfiguration request. To modify the channel list, the present request must be released (DaqReleaseRequest) and a new configuration requested.

```
struct ADC_request
   {
   unsigned short far *channel_array_ptr;
   float far *gain_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *ADC_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel;
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double         clock_rate;
   double         sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short calibration;
   unsigned short timeout_interval;
   unsigned long  request_status;
   };
```

Figure 11. Analog input request structure.

## IMPORTANT:

1. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.

2. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

| | | | |
|---|---|---|---|
| channel_array_ptr | This pointer defines the address of an unsigned short integer array specifying the logical analog input channel(s) to be operated on by this request. | | |
| gain_array_ptr | This pointer defines the address of a floating point array specifying the gain for each channel in the array pointed to by channel_array_ptr. There must be a one-to-one correspondence between the values specified by channel_array_ptr and the values specified by gain_array_ptr. | | |
| reserved1[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| array_length | This unsigned short integer value defines the length of the arrays pointed to by channel_array_ptr and gain_array_ptr. The arrays must be of equal length. | | |
| ADC_buffer | This pointer defines the address of the first data buffer structure. Data buffer structures are discussed in chapter 9. | | |
| reserved2[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| trigger_source | This unsigned short integer value specifies the trigger source for this request. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_TRIGGER | 0 | internal (software) trigger |
| | TTL_TRIGGER | 1 | TTL trigger |
| | ANALOG_TRIGGER | 2 | analog trigger |
| | DIGITAL_TRIGGER | 3 | digital value trigger |
| trigger_mode | This unsigned short integer value defines the trigger mode. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | CONTINUOUS_TRIGGER | 0 | only one trigger is required to start the output operation |
| | ONE_SHOT_TRIGGER | 1 | a trigger is required for each input scan |
| trigger_slope | This unsigned short integer value defines the slope for TTL and analog triggers. trigger_slope is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | RISING_EDGE | 0 | for TTL and analog triggers only, specifies a low-to-high transition is required. |
| | FALLING_EDGE | 1 | for TTL and analog triggers only, specifies a high-to-low transition is required. |
| trigger_channel | This unsigned short value specifies the channel to be used as the input for the analog or digital trigger sources. trigger_channel is undefined for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_voltage | This double precision value defines the trigger voltage level for the analog trigger. trigger_voltage is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_value | This unsigned long value defines the value required on the digital input for a digital trigger to be generated. trigger_value is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| reserved3[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| IO_mode | This unsigned short integer value specifies the method of data transfer. | | |
| | DAQDRIVE Constant | Value | Description |
| | FOREGROUND_CPU | 0 | DAQDRIVE takes control of the CPU until the request is complete |
| | BACKGROUND_IRQ | 1 | hardware interrupts are used to gain control of the CPU and input the data |
| | FOREGROUND_DMA | 2 | DMA is used to input the data; the CPU monitors / controls the DMA operation |
| | BACKGROUND_DMA | 3 | DMA is used to input the data; interrupts are used to monitor / control the DMA operation |

Figure 12. Analog input request structure definition.

| | | | |
|---|---|---|---|
| clock_source | This unsigned short value selects the clock source to provide the timing for multiple point input operations. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_CLOCK | 0 | the sampling rate is generated by the on-board clock circuitry |
| | EXTERNAL_CLOCK | 1 | the sampling rate is generated from an external input |
| clock_rate | This double precision value defines the clock frequency of the external clock. clock_rate is ignored for internal clock sources. | | |
| sample_rate | This double precision value specifies the input data rate in samples / second (Hz) for multiple point operations. | | |
| reserved4[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| number_of_scans | This unsigned long integer value defines the number of times the channels specified in channel_array_ptr will be input. Setting number_of_cycles = 0 will cause the channels to be scanned continuously. | | |
| scan_event_level | This unsigned long integer value defines the frequency at which scan events are reported to the application. For example, setting scan_event_level to 100 causes a scan event to be generated each time 100 scans are completed. | | |
| reserved5[8] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| calibration | This unsigned short integer value specifies the type of calibration to be performed for this request. The calibration methods are dependent on the type of hardware installed. Consult the hardware specific appendices for specifics on adapter calibration. | | |
| | DAQDRIVE Constant | Value | Description |
| | NO_CALIBRATION | 0x0000 | No calibration requested. |
| | AUTO_CALIBRATE | 0x0001 | Perform auto-calibration on this request. |
| | AUTO_ZERO | 0x0002 | Perform auto-zero on this request. |
| timeout_interval | This unsigned short integer value defines a time-out interval, in seconds, for foreground mode processes. The input operation will abort if the input can not be read every timeout_interval seconds. Setting timeout_interval = 0 disables the time-out function and causes the routine to wait indefinitely. | | |
| request_status | This unsigned long integer value provides the application with the current status of the request. DAQDRIVE does not rely on the information contained in this field nor does it ever clear any of the event bits to 0. Therefore, the application should initialize request_status during the configuration process and may modify its contents at any time. | | |
| | DAQDRIVE Constant | Value | Description |
| | NO_EVENTS | 0x00000000 | This constant does not represent an event status. It is provided to the application for convenience. |
| | TRIGGER_EVENT | 0x00000001 | When set to 1, this bit indicates the specified trigger has been received. |
| | COMPLETE_EVENT | 0x00000002 | When set to 1, this bit indicates the request has completed successfully. |
| | BUFFER_EMPTY_EVENT | 0x00000004 | When set to 1, this bit indicates at least one of the specified output data buffers has been emptied. |
| | BUFFER_FULL_EVENT | 0x00000008 | When set to 1, this bit indicates at least one of the specified input data buffers has been filled. |
| | SCAN_EVENT | 0x00000010 | When set to 1, this bit indicates the number of scans specified by scan_event_level have been completed at least once. |
| | USER_BREAK_EVENT | 0x20000000 | When set to 1, this bit indicates the request has terminated due to a user-break. |
| | TIMEOUT_EVENT | 0x40000000 | When set to 1, this bit indicates the request has terminated because the specified time-out interval was exceeded. |
| | RUNTIME_ERROR_EVENT | 0x80000000 | When set to 1, this bit indicates the request has terminated because of an error during processing. The application can determine the source of the error using the DaqGetRuntimeError procedure. |

Figure 11 (continued). Analog input request structure definition.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 500 points each from 2 analog input channels.                  */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[2]   = { 0, 1 };
         float gain_settings[2] = { 1, 8 };
unsigned short status;
         short data_array[1000];

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for analog input.  *****/

/*************************************************************************/
/* put data in data_array       data_array is 1000 points long           */
/* next_structure = NULL  (no more structures)                           */
/*************************************************************************/

data_structure.data_buffer    = data_array;
data_structure.buffer_length  = 1000;
data_structure.next_structure = NULL;

/*****  Prepare the A/D request structure.  *****/

/*************************************************************************/
/* channel list is in channel_num      gain list is in gain_settings     */
/* channel & gain array length is 2    use data_structure for data       */
/* trigger source is internal          trigger mode is continuous        */
/* input using IRQs (in background)    use internal clock                */
/* sample at 1 KHz                     scan channel list 500 times       */
/* do not signal buffer scan events    do not implement time-out         */
/*************************************************************************/

user_request.channel_array_ptr = channel_num;
user_request.gain_array_ptr    = gain_settings;
user_request.array_length      = 2;
user_request.ADC_buffer        = data_structure;
user_request.trigger_source    = INTERNAL_TRIGGER
user_request.trigger_mode      = CONTINUOUS_TRIGGER;
user_request.IO_mode           = BACKGROUND_IRQ;
user_request.clock_source      = INTERNAL_CLOCK;
user_request.sample_rate       = 1000;
user_request.number_of_scans   = 500;
user_request.scan_event_level  = 0;
user_request.calibration       = NO_CALIBRATION;
user_request.timeout_interval  = 0;
user_request.request_status    = NO_EVENTS;

/*****  Indicate data buffer ready for input.  *****/

data_structure.buffer_status = BUFFER_EMPTY;

/*****  Request A/D input.  *****/

request_handle = 0;
status = DaqAnalogInput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("A/D request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
```

## 13.5 DaqAnalogOutput

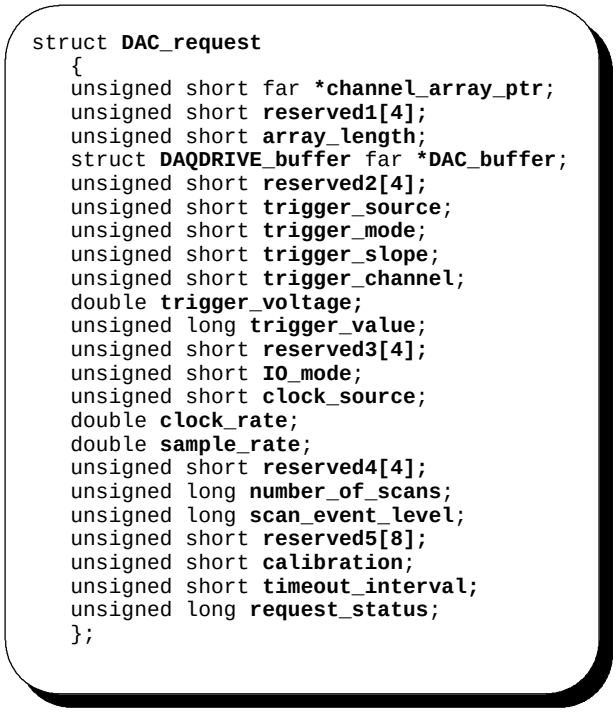
DaqAnalogOutput is DAQDRIVE's generic D/A converter interface. It does not configure any hardware but acts simply to confirm that all parameters are valid and that the type of operation requested is supported by the target hardware.

```
unsigned short DaqAnalogOutput(unsigned short logical_device,
                               struct DAC_request far *user_request,
                               unsigned short far *request_handle)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

user_request - This structure pointer defines the address of a D/A request structure containing the desired configuration information for this operation. This structure is discussed in detail on the following pages.

request_handle - This unsigned short integer pointer is used to identify this analog output request. For a new configuration, request_handle is set to 0 by the application before calling DaqAnalogOutput. If the configuration is successful request_handle will be assigned a unique non-zero value by the DaqAnalogOutput procedure. If the application modifies a previously configured request, the application must call DaqAnalogOutput using the previously assigned request_handle. All parameters except the channel list may be modified using a reconfiguration request. To modify the channel list, the present request must be released (DaqReleaseRequest) and a new configuration requested.



```
struct DAC_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *DAC_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel;
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short calibration;
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

Figure 13. Analog output request structure.

## IMPORTANT:

1. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.

2. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required

| | | | |
|---|---|---|---|
| channel_array_ptr | This pointer defines the address of an unsigned short integer array specifying the logical analog output channel(s) to be operated on by this request. | | |
| reserved1[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| array_length | This unsigned short integer value defines the number of channels contained in the array pointed to by channel_array_ptr. | | |
| DAC_buffer | This pointer defines the address of the first data buffer structure. Data buffer structures are discussed in chapter 9. | | |
| reserved2[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| trigger_source | This unsigned short integer value specifies the trigger source for this request. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_TRIGGER | 0 | internal (software) trigger |
| | TTL_TRIGGER | 1 | TTL trigger |
| | ANALOG_TRIGGER | 2 | analog trigger |
| | DIGITAL_TRIGGER | 3 | digital value trigger |
| trigger_mode | This unsigned short integer value defines the trigger mode. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | CONTINUOUS_TRIGGER | 0 | only one trigger is required to start the output operation |
| | ONE_SHOT_TRIGGER | 1 | a trigger is required for each output scan |
| trigger_slope | This unsigned short integer value defines the slope for TTL and analog triggers. trigger_slope is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | RISING_EDGE | 0 | for TTL and analog triggers only, specifies a low-to-high transition is required. |
| | FALLING_EDGE | 1 | for TTL and analog triggers only, specifies a high-to-low transition is required. |
| trigger_channel | This unsigned short value specifies the channel to be used as the input for the analog or digital trigger sources. trigger_channel is undefined for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_voltage | This double precision value defines the trigger voltage level for the analog trigger. trigger_voltage is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_value | This unsigned long value defines the value required on the digital input for a digital trigger to be generated. trigger_value is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| reserved3[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| IO_mode | This unsigned short integer value specifies the method of data transfer. | | |
| | DAQDRIVE Constant | Value | Description |
| | FOREGROUND_CPU | 0 | DAQDRIVE takes control of the CPU until the request is complete |
| | BACKGROUND_IRQ | 1 | hardware interrupts are used to gain control of the CPU and input the data |
| | FOREGROUND_DMA | 2 | DMA is used to input the data; the CPU monitors / controls the DMA operation |
| | BACKGROUND_DMA | 3 | DMA is used to input the data; interrupts are used to monitor / control the DMA operation |

Figure 14. Analog output request structure definition.

| | | | |
|---|---|---|---|
| clock_source | This unsigned short value selects the clock source to provide the timing for multiple point output operations. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_CLOCK | 0 | the sampling rate is generated by the on-board clock circuitry |
| | EXTERNAL_CLOCK | 1 | the sampling rate is generated from an external input |
| clock_rate | This double precision value defines the clock frequency of the external clock. clock_rate is ignored for internal clock sources. | | |
| sample_rate | This double precision value specifies the output data rate in samples / second (Hz) for multiple point operations. | | |
| reserved4[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| number_of_scans | This unsigned long integer value defines the number of times the channels specified in channel_array_ptr will be written. Setting number_of_cycles = 0 will cause the channels to be scanned continuously. | | |
| scan_event_level | This unsigned long integer value defines the frequency at which scan events are reported to the application. For example, setting scan_event_level to 100 causes a scan event to be generated each time 100 scans are completed. | | |
| reserved5[8] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| calibration | This unsigned short integer value specifies the type of calibration to be performed for this request. The calibration methods are dependent on the type of hardware installed. Consult the hardware specific appendices for specifics on adapter calibration. | | |
| | DAQDRIVE Constant | Value | Description |
| | NO_CALIBRATION | 0x0000 | No calibration requested. |
| | AUTO_CALIBRATE | 0x0001 | Perform auto-calibration on this request. |
| | AUTO_ZERO | 0x0002 | Perform auto-zero on this request. |
| timeout_interval | This unsigned short integer value defines a time-out interval, in seconds, for foreground mode processes. The operation will abort if the analog output can not be updated every timeout_interval seconds. Setting timeout_interval = 0 disables the time-out function and causes the routine to wait indefinitely. | | |
| request_status | This unsigned long integer value provides the application with the current status of the request. DAQDRIVE does not rely on the information contained in this field nor does it ever clear any of the event bits to 0. Therefore, the application should initialize request_status during the configuration process and may modify its contents at any time. | | |
| | DAQDRIVE Constant | Value | Description |
| | NO_EVENTS | 0x00000000 | This constant does not represent an event status. It is provided to the application for convenience. |
| | TRIGGER_EVENT | 0x00000001 | When set to 1, this bit indicates the specified trigger has been received. |
| | COMPLETE_EVENT | 0x00000002 | When set to 1, this bit indicates the request has completed successfully. |
| | BUFFER_EMPTY_EVENT | 0x00000004 | When set to 1, this bit indicates at least one of the specified output data buffers has been emptied. |
| | BUFFER_FULL_EVENT | 0x00000008 | When set to 1, this bit indicates at least one of the specified input data buffers has been filled. |
| | SCAN_EVENT | 0x00000010 | When set to 1, this bit indicates the number of scans specified by scan_event_level have been completed at least once. |
| | USER_BREAK_EVENT | 0x20000000 | When set to 1, this bit indicates the request has terminated due to a user-break. |
| | TIMEOUT_EVENT | 0x40000000 | When set to 1, this bit indicates the request has terminated because the specified time-out interval was exceeded. |
| | RUNTIME_ERROR_EVENT | 0x80000000 | When set to 1, this bit indicates the request has terminated because of an error during processing. The application can determine the source of the error using the DaqGetRuntimeError procedure. |

Figure 13 (continued). Analog output request structure definition.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Output a 20 point waveform to a D/A channel.                         */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num = 0;
unsigned short status;
         short data_array[20];

struct DAC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for analog output.  *****/

/*************************************************************************/
/* data is in data_array     data_array is 20 points long                */
/* output buffer 1 time      next_structure = NULL  (no more structures) */
/*************************************************************************/

data_structure.data_buffer    = data_array;
data_structure.buffer_length  = 20;
data_structure.buffer_cycles  = 1;
data_structure.next_structure = NULL;

/*****  Prepare the D/A request structure.  *****/

/*************************************************************************/
/* channel list is in channel_num      channel_num is 1 channel long     */
/* data is in data_structure           trigger source is internal        */
/* trigger mode is continuous          output using IRQs (in background) */
/* use internal clock                  output 1 point every 10ms (100Hz) */
/* repeat all buffers once             do not signal buffer scan events  */
/* do not implement time-out                                             */
/*************************************************************************/

user_request.channel_array_ptr = &channel_num;
user_request.array_length      = 1;
user_request.DAC_buffer        = data_structure;
user_request.trigger_source    = INTERNAL_TRIGGER
user_request.trigger_mode      = CONTINUOUS_TRIGGER;
user_request.IO_mode           = BACKGROUND_IRQ;
user_request.clock_source      = INTERNAL_CLOCK;
user_request.sample_rate       = 100;
user_request.number_of_scans   = 1;
user_request.scan_event_level  = 0;
user_request.calibration       = NO_CALIBRATION;
user_request.timeout_interval  = 0;
user_request.request_status    = NO_EVENTS;

/*****  Indicate data buffer ready for output.  *****/

data_structure.buffer_status = BUFFER_FULL;

/*****  Request D/A output.  *****/

request_handle = 0;
status = DaqAnalogOutput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("D/A request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

## 13.6 DaqArmRequest

DaqArmRequest is executed after the DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput functions to prepare the specified configuration for execution. During the arming process, any resources required for the request (e.g. IRQs, DMA channels, timers) are allocated for use by this request and all hardware is prepared for the impending trigger.

```
unsigned short DaqArmRequest(unsigned short request_handle)
```

request_handle - This unsigned short integer variable is used to define which request is to be armed. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput or DaqDigitalOutput.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 500 points from an A/D channel.                                */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;
         short data_array[500];

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for analog output.  *****/

/*****  Prepare the A/D request structure.  *****/

/*****  Request A/D input.  *****/

request_handle = 0;
status = DaqAnalogInput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("A/D request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Arm the request.  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   printf("Arm request error.  Status code %d.\n",status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

## 13.7 DaqBytesToWords

DaqBytesToWords reverses the function of DaqWordsToBytes converting an unsigned short integer array of 8-bit "packed" values into an unsigned short integer array of 16-bit "un-packed" values. These function are provided especially for languages that do not support 8-bit variable types.

DaqBytesToWords reads the "packed" 8-bit values in array byte_array, converts these values to their "un-packed" 16-bit unsigned short integer format, and stores the results in array word_array. For an array of four values, the packed and un-packed arrays appear as follows:

| | byte | byte | byte | byte | | | | |
|---|---|---|---|---|---|---|---|---|
| "packed" array | 14 | 2E | 6 | F7 | | | | |
| "un-packed" array | 14 | 0 | 2E | 0 | 6 | 0 | F7 | 0 |
| | integer | | integer | | integer | | integer | |

```
void DaqBytesToWords(unsigned short far *byte_array,
                     unsigned short far *word_array,
                     unsigned long array_length)
```

byte_array - This is a pointer to an unsigned short integer array containing the "packed" values to be converted. byte_array must be at least 'array_length ÷ 2' short integers (array_length bytes) in length and may specify the same array as word_array.

word_array - This is a pointer to an unsigned short integer array where the "un-packed" values will be stored. word_array must be at least array_length short integers in length and may specify the same array as byte_array.

array_length - This is an unsigned long integer value defining the number of data points to be converted. byte_array must be at least 'array_length ÷ 2' short integers (array_length bytes) in length while word_array must be at least array_length short integers in length.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 100 points each from 4 digital input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[4] = { 0, 1, 6, 3 };
unsigned short status;
unsigned short data_array[400];
unsigned short array_index;
unsigned short error;

unsigned long  event_mask;

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare the digital input request structure.  *****/

/*****  Request digital input (see DaqDigitalInput).  *****/

/*****  Arm the request (see DaqArmRequest).  *****/

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger request error.  Status code %d.\n",status);
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   status = DaqGetRuntimeError(request_handle, &error);
   exit(error);
   }

/*****  Un-pack the values for display.  *****/

DaqBytesToWords(data_array, data_array, 400);

/*****  Display the input values as integers.  *****/

for (array_index = 0; array_index < 400; array_index++)
   printf("digital input = %4x\n", data_array[array_index]);
```

## 13.8 DaqCloseDevice

DaqCloseDevice informs DAQDRIVE that the specified logical device is no longer required and any resources required by this device may be freed.

```
unsigned short DaqCloseDevice(unsigned short logical_device)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "DAQP-16";
char far *config_file = "daqp-16.dat";

/*****  Open the DAQP-16.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DAQP-16 operations here.  *****/

/*****  Close the DAQP-16.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.9 DaqConvertBuffer

DaqConvertBuffer converts a buffer of raw digital readings returned from an analog input request into a buffer of "real world" floating point values in engineering units. DaqConvertBuffer uses the hardware configuration information stored within DAQDRIVE to convert these numbers based on the signal type, gain settings, and signal conditioner parameters defined for each channel in the system.

```
void DaqConvertBuffer(unsigned short logical_device,
                      struct sigcon_request far *sigcon_request)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

sigcon_request - This structure pointer defines the address of a signal conditioner data conversion request structure specifying the parameters to be used during the conversion process.

```
struct sigcon_request
   {
   unsigned short far *channel_array_ptr;
   float far *gain_array_ptr;
   unsigned short array_length;
   void huge *raw_data_ptr;
   double huge *converted_data_ptr;
   unsigned long number_of_points;
   };
```

| | |
|---|---|
| channel_array_ptr | This pointer defines the address of an unsigned short integer array specifying the logical analog input channel(s) from which the raw data was acquired. |
| gain_array_ptr | This pointer defines the address of a floating point array specifying the gain for each channel in the array pointed to by channel_array_ptr. There must be a one-to-one correspondence between the values specified by channel_array_ptr and the values specified by gain_array_ptr. |
| array_length | This unsigned short integer value defines the length of the arrays pointed to by channel_array_ptr and gain_array_ptr. The arrays must be of equal length. |
| raw_data_ptr | This void huge pointer specifies the address of an array (buffer) containing the raw data to be converted. raw_data_ptr is declared as a void to allow it to point to data of any type.. |
| converted_data_ptr | This pointer defines the address of a double-precision array where the converted data will be stored. |
| number_of_points | This unsigned long integer value defines the length of the arrays pointed to by raw_data_ptr and converted_data_ptr in units of "number-of-samples". The arrays must be of equal length. |

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 100 points each from 4 digital input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[4] = { 0, 1, 6, 3 };
unsigned short status;
unsigned short data_array[400];
unsigned short array_index;
unsigned short error;

unsigned long  event_mask;

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare the digital input request structure.  *****/

/*****  Request digital input (see DaqDigitalInput).  *****/

/*****  Arm the request (see DaqArmRequest).  *****/

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger request error.  Status code %d.\n",status);
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   status = DaqGetRuntimeError(request_handle, &error);
   exit(error);
   }

/*****  Un-pack the values for display.  *****/

DaqBytesToWords(data_array, data_array, 400);

/*****  Display the input values as integers.  *****/

for (array_index = 0; array_index < 400; array_index++)
   printf("digital input = %4x\n", data_array[array_index]);
```

## 13.10 DaqConvertPoint

DaqConvertPoint converts a single raw digital reading returned from an analog input request into a "real world" floating point value in engineering units. DaqConvertPoint uses the hardware configuration information stored within DAQDRIVE to convert the data based on the signal type, gain settings, and signal conditioner parameters defined for each channel in the system.

```
void DaqConvertPoint(unsigned short logical_device,
                     unsigned short ADC_channel,
                     float gain,
                     void far *raw_value,
                     double far *converted_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_channel - This unsigned short integer value is used to define the analog input channel from which the raw data was acquired.

gain - This floating point value is used to define the gain of the analog input channel from which the raw data was acquired.

raw_value - This void pointer specifies the address of the raw data to be converted and is declared as a void to allow it to point to data of any type.

converted_value - This pointer specifies the address of a double-precision floating point value where the converted value is to be stored.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 100 points each from 4 digital input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[4] = { 0, 1, 6, 3 };
unsigned short status;
unsigned short data_array[400];
unsigned short array_index;
unsigned short error;

unsigned long  event_mask;

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare the digital input request structure.  *****/

/*****  Request digital input (see DaqDigitalInput).  *****/

/*****  Arm the request (see DaqArmRequest).  *****/

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger request error.  Status code %d.\n",status);
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   status = DaqGetRuntimeError(request_handle, &error);
   exit(error);
   }

/*****  Un-pack the values for display.  *****/

DaqBytesToWords(data_array, data_array, 400);

/*****  Display the input values as integers.  *****/

for (array_index = 0; array_index < 400; array_index++)
   printf("digital input = %4x\n", data_array[array_index]);
```

## 13.11 DaqConvertScan

DaqConvertScan converts a single scan of raw digital readings returned from an analog input request into "real world" floating point values in engineering units. DaqConvertScan uses the hardware configuration information stored within DAQDRIVE to convert the data based on the signal type, gain settings, and signal conditioner parameters defined for each channel in the system.

```
void DaqConvertScan(unsigned short logical_device,
                    unsigned short far *channel_array_ptr,
                    float far *gain_array_ptr,
                    unsigned short array_length,
                    void far *raw_data_ptr,
                    double far *converted_data_ptr)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array_ptr - This pointer defines the address of an unsigned short integer array specifying the logical analog input channel(s) from which the raw data was acquired.

gain_array_ptr - This pointer defines the address of a floating point array specifying the gain for each channel in the array pointed to by channel_array_ptr. There must be a one-to-one correspondence between the values specified by channel_array_ptr and the values specified by gain_array_ptr.

array_length - This unsigned short integer value defines the length of the arrays pointed to by channel_array_ptr, gain_array_ptr, raw_data_ptr, and converted_data_ptr. The arrays must be of equal length.

raw_data_ptr - This void huge pointer specifies the address of an array (buffer) containing the raw data to be converted. raw_data_ptr is declared as a void to allow it to point to data of any type.

converted_data_ptr - This pointer defines the address of a double-precision array where the converted data will be stored.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 100 points each from 4 digital input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[4] = { 0, 1, 6, 3 };
unsigned short status;
unsigned short data_array[400];
unsigned short array_index;
unsigned short error;

unsigned long  event_mask;

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare the digital input request structure.  *****/

/*****  Request digital input (see DaqDigitalInput).  *****/

/*****  Arm the request (see DaqArmRequest).  *****/

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger request error.  Status code %d.\n",status);
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & RUNTIME_ERROR_EVENT) != 0)
   {
   status = DaqGetRuntimeError(request_handle, &error);
   exit(error);
   }

/*****  Un-pack the values for display.  *****/

DaqBytesToWords(data_array, data_array, 400);

/*****  Display the input values as integers.  *****/

for (array_index = 0; array_index < 400; array_index++)
   printf("digital input = %4x\n", data_array[array_index]);
```

## 13.12 DaqDigitalInput

DaqDigitalInput is DAQDRIVE's generic digital input interface. DaqDigitalInput does not configure any hardware but acts simply to confirm that all parameters are valid and that the type of operation requested is supported by the target hardware.

```
unsigned short DaqDigitalInput(unsigned short logical_device,
                               struct digio_request far *user_request,
                               unsigned short far *request_handle)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

user_request - This structure pointer defines the address of a digital I/O request structure containing the desired configuration information for this operation. This structure is discussed in detail on the following pages.

request_handle - This unsigned short integer pointer is used to identify this digital input request. For a new configuration, request_handle is set to 0 by the application before calling DaqDigitalInput. If the configuration is successful request_handle will be assigned a unique non-zero value by the DaqDigitalInput procedure. If the application modifies a previously configured request, the application must call DaqDigitalInput using the previously assigned request_handle. All parameters except the channel list may be modified using a reconfiguration request. To modify the channel list, the present request must be released (DaqReleaseRequest) and a new configuration requested.

```
struct digio_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *digio_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel;
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

Figure 15. Digital input request structure.

## IMPORTANT:

1. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.

2. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

| | | | |
|---|---|---|---|
| channel_array_ptr | This pointer defines the address of an unsigned short integer array specifying the logical digital input channel(s) to be operated on by this request. | | |
| reserved1[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| array_length | This unsigned short integer value defines the number of channels contained in the array pointed to by channel_array_ptr. | | |
| digio_buffer | This pointer defines the address of the first data buffer structure. Data buffer structures are discussed in chapter 9. | | |
| reserved2[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| trigger_source | This unsigned short integer value specifies the trigger source for this request. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_TRIGGER | 0 | internal (software) trigger |
| | TTL_TRIGGER | 1 | TTL trigger |
| | ANALOG_TRIGGER | 2 | analog trigger |
| | DIGITAL_TRIGGER | 3 | digital value trigger |
| trigger_mode | This unsigned short integer value defines the trigger mode. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | CONTINUOUS_TRIGGER | 0 | only one trigger is required to start the output operation |
| | ONE_SHOT_TRIGGER | 1 | a trigger is required for each output scan |
| trigger_slope | This unsigned short integer value defines the slope for TTL and analog triggers. trigger_slope is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | RISING_EDGE | 0 | for TTL and analog triggers only, specifies a low-to-high transition is required. |
| | FALLING_EDGE | 1 | for TTL and analog triggers only, specifies a high-to-low transition is required. |
| trigger_channel | This unsigned short value specifies the channel to be used as the input for the analog or digital trigger sources. trigger_channel is undefined for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_voltage | This double precision value defines the trigger voltage level for the analog trigger. trigger_voltage is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_value | This unsigned long value defines the value required on the digital input for a digital trigger to be generated. trigger_value is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| reserved3[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| IO_mode | This unsigned short integer value specifies the method of data transfer. | | |
| | DAQDRIVE Constant | Value | Description |
| | FOREGROUND_CPU | 0 | DAQDRIVE takes control of the CPU until the request is complete |
| | BACKGROUND_IRQ | 1 | hardware interrupts are used to gain control of the CPU and input the data |
| | FOREGROUND_DMA | 2 | DMA is used to input the data; the CPU monitors / controls the DMA operation |
| | BACKGROUND_DMA | 3 | DMA is used to input the data; interrupts are used to monitor / control the DMA operation |

Figure 16. Digital input request structure definition.

<table>
<tr><td>clock_source</td><td colspan="3">This unsigned short value selects the clock source to provide the timing for multiple point input operations.<table><tr><th>DAQDRIVE Constant</th><th>Value</th><th>Description</th></tr><tr><td>INTERNAL_CLOCK</td><td>0</td><td>the sampling rate is generated by the on-board clock circuitry</td></tr><tr><td>EXTERNAL_CLOCK</td><td>1</td><td>the sampling rate is generated from an external input</td></tr></table></td></tr>
<tr><td>clock_rate</td><td colspan="3">This double precision value defines the clock frequency of the external clock. clock_rate is ignored for internal clock sources.</td></tr>
<tr><td>sample_rate</td><td colspan="3">This double precision value specifies the input data rate in samples / second (Hz) for multiple point operations.</td></tr>
<tr><td>reserved4[4]</td><td colspan="3">This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0.</td></tr>
<tr><td>number_of_scans</td><td colspan="3">This unsigned long integer value defines the number of times the channels specified in channel_array_ptr will be input. Setting number_of_cycles = 0 will cause the channels to be scanned continuously.</td></tr>
<tr><td>scan_event_level</td><td colspan="3">This unsigned long integer value defines the frequency at which scan events are reported to the application. For example, setting scan_event_level to 100 causes a scan event to be generated each time 100 scans are completed.</td></tr>
<tr><td>reserved5[8]</td><td colspan="3">This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0.</td></tr>
<tr><td>timeout_interval</td><td colspan="3">This unsigned short integer value defines a time-out interval, in seconds, for foreground mode processes. The operation will abort if the digital input can not be read every timeout_interval seconds. Setting timeout_interval = 0 disables the time-out function and causes the routine to wait indefinitely.</td></tr>
<tr><td>request_status</td><td colspan="3">This unsigned long integer value provides the application with the current status of the request. DAQDRIVE does not rely on the information contained in this field nor does it ever clear any of the event bits to 0. Therefore, the application should initialize request_status during the configuration process and may modify its contents at any time.<table><tr><th>DAQDRIVE Constant</th><th>Value</th><th>Description</th></tr><tr><td>NO_EVENTS</td><td>0x00000000</td><td>This constant does not represent an event status. It is provided to the application for convenience.</td></tr><tr><td>TRIGGER_EVENT</td><td>0x00000001</td><td>When set to 1, this bit indicates the specified trigger has been received.</td></tr><tr><td>COMPLETE_EVENT</td><td>0x00000002</td><td>When set to 1, this bit indicates the request has completed successfully.</td></tr><tr><td>BUFFER_EMPTY_EVENT</td><td>0x00000004</td><td>When set to 1, this bit indicates at least one of the specified output data buffers has been emptied.</td></tr><tr><td>BUFFER_FULL_EVENT</td><td>0x00000008</td><td>When set to 1, this bit indicates at least one of the specified input data buffers has been filled.</td></tr><tr><td>SCAN_EVENT</td><td>0x00000010</td><td>When set to 1, this bit indicates the number of scans specified by scan_event_level have been completed at least once.</td></tr><tr><td>USER_BREAK_EVENT</td><td>0x20000000</td><td>When set to 1, this bit indicates the request has terminated due to a user-break.</td></tr><tr><td>TIMEOUT_EVENT</td><td>0x40000000</td><td>When set to 1, this bit indicates the request has terminated because the specified time-out interval was exceeded.</td></tr><tr><td>RUNTIME_ERROR_EVENT</td><td>0x80000000</td><td>When set to 1, this bit indicates the request has terminated because of an error during processing. The application can determine the source of the error using the DaqGetRuntimeError procedure.</td></tr></table></td></tr>
</table>

Figure 15 (continued). Digital input request structure definition.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 100 points each from 4 digital input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[4] = { 0, 1, 6, 3 };
unsigned short status;
unsigned char  data_array[400];

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for digital input.  *****/

/*************************************************************************/
/* put data in data_array       data_array is 400 points long            */
/* next_structure = NULL  (no more structures)                           */
/*************************************************************************/

data_structure.data_buffer    = data_array;
data_structure.buffer_length  = 1000;
data_structure.next_structure = NULL;

/*****  Prepare the digital input request structure.  *****/

/*************************************************************************/
/* channel list is in channel_num      channel list length is 4          */
/* use data_structure for data         trigger source is internal        */
/* trigger mode is continuous          input using CPU (in foreground)   */
/* use internal clock                  sample at 100 Hz                  */
/* scan channels list 100 times        do not signal buffer scan events  */
/* do not implement time-out                                             */
/*************************************************************************/

user_request.channel_array_ptr = channel_num;
user_request.array_length      = 4;
user_request.ADC_buffer        = data_structure;
user_request.trigger_source    = INTERNAL_TRIGGER
user_request.trigger_mode      = CONTINUOUS_TRIGGER;
user_request.IO_mode           = FOREGROUND_CPU;
user_request.clock_source      = INTERNAL_CLOCK;
user_request.sample_rate       = 100;
user_request.number_of_scans   = 100;
user_request.scan_event_level  = 0;
user_request.timeout_interval  = 0;
user_request.request_status    = NO_EVENTS;

/*****  Indicate data buffer ready for input.  *****/

data_structure.buffer_status = BUFFER_EMPTY;

/*****  Request digital input.  *****/

request_handle = 0;
status = DaqDigitalInput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("Digital input request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

## 13.13 DaqDigitalOutput

DaqDigitalOutput is DAQDRIVE's generic digital output interface. DaqDigitalOutput does not configure any hardware but acts simply to confirm that all parameters are valid and that the type of operation requested is supported by the target hardware.

```
unsigned short DaqDigitalOutput(unsigned short logical_device,
                                struct digio_request far *user_request,
                                unsigned short far *request_handle)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

user_request - This structure pointer defines the address of a digital I/O request structure containing the desired configuration information for this operation. This structure is discussed in detail on the following pages.

request_handle - This unsigned short integer pointer is used to identify this digital output request. For a new configuration, request_handle is set to 0 by the application before calling DaqDigitalOutput. If the configuration is successful request_handle will be assigned a unique non-zero value by the DaqDigitalOutput procedure. If the application modifies a previously configured request, the application must call DaqDigitalOutput using the previously assigned request_handle. All parameters except the channel list may be modified using a reconfiguration request. To modify the channel list, the present request must be released (DaqReleaseRequest) and a new configuration requested.

```
struct digio_request
   {
   unsigned short far *channel_array_ptr;
   unsigned short reserved1[4];
   unsigned short array_length;
   struct DAQDRIVE_buffer far *digio_buffer;
   unsigned short reserved2[4];
   unsigned short trigger_source;
   unsigned short trigger_mode;
   unsigned short trigger_slope;
   unsigned short trigger_channel;
   double trigger_voltage;
   unsigned long trigger_value;
   unsigned short reserved3[4];
   unsigned short IO_mode;
   unsigned short clock_source;
   double clock_rate;
   double sample_rate;
   unsigned short reserved4[4];
   unsigned long number_of_scans;
   unsigned long scan_event_level;
   unsigned short reserved5[8];
   unsigned short timeout_interval;
   unsigned long request_status;
   };
```

Figure 17. Digital output request structure.

## IMPORTANT:

1. Once the request is armed using DaqArmRequest, the only field the application can modify is request_status. All other fields in the request structure must remain constant until the operation is completed or otherwise terminated.

2. If the request structure is dynamically allocated by the application, it **MUST NOT** be de-allocated until the request has been released by the DaqReleaseRequest procedure. In addition, applications using the Windows version of DAQDRIVE should use DaqAllocateMemory, DaqAllocateMemory32, or DaqAllocateRequest if dynamically allocated request structures are required.

| | | | |
|---|---|---|---|
| channel_array_ptr | This pointer defines the address of an unsigned short integer array specifying the logical digital output channel(s) to be operated on by this request. | | |
| reserved1[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| array_length | This unsigned short integer value defines the number of channels contained in the array pointed to by channel_array_ptr. | | |
| digio_buffer | This pointer defines the address of the first data buffer structure. Data buffer structures are discussed in chapter 9. | | |
| reserved2[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| trigger_source | This unsigned short integer value specifies the trigger source for this request. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_TRIGGER | 0 | internal (software) trigger |
| | TTL_TRIGGER | 1 | TTL trigger |
| | ANALOG_TRIGGER | 2 | analog trigger |
| | DIGITAL_TRIGGER | 3 | digital value trigger |
| trigger_mode | This unsigned short integer value defines the trigger mode. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | CONTINUOUS_TRIGGER | 0 | only one trigger is required to start the output operation |
| | ONE_SHOT_TRIGGER | 1 | a trigger is required for each output scan |
| trigger_slope | This unsigned short integer value defines the slope for TTL and analog triggers. trigger_slope is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| | DAQDRIVE Constant | Value | Description |
| | RISING_EDGE | 0 | for TTL and analog triggers only, specifies a low-to-high transition is required. |
| | FALLING_EDGE | 1 | for TTL and analog triggers only, specifies a high-to-low transition is required. |
| trigger_channel | This unsigned short value specifies the channel to be used as the input for the analog or digital trigger sources. trigger_channel is undefined for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_voltage | This double precision value defines the trigger voltage level for the analog trigger. trigger_voltage is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| trigger_value | This unsigned long value defines the value required on the digital input for a digital trigger to be generated. trigger_value is ignored for all other trigger sources. Trigger selections are discussed in chapter 10. | | |
| reserved3[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| IO_mode | This unsigned short integer value specifies the method of data transfer. | | |
| | DAQDRIVE Constant | Value | Description |
| | FOREGROUND_CPU | 0 | DAQDRIVE takes control of the CPU until the request is complete |
| | BACKGROUND_IRQ | 1 | hardware interrupts are used to gain control of the CPU and input the data |
| | FOREGROUND_DMA | 2 | DMA is used to input the data; the CPU monitors / controls the DMA operation |
| | BACKGROUND_DMA | 3 | DMA is used to input the data; interrupts are used to monitor / control the DMA operation |

Figure 18. Digital output request structure definition.

| | | | |
|---|---|---|---|
| clock_source | This unsigned short value selects the clock source to provide the timing for multiple point output operations. | | |
| | DAQDRIVE Constant | Value | Description |
| | INTERNAL_CLOCK | 0 | the sampling rate is generated by the on-board clock circuitry |
| | EXTERNAL_CLOCK | 1 | the sampling rate is generated from an external input |
| clock_rate | This double precision value defines the clock frequency of the external clock. clock_rate is ignored for internal clock sources | | |
| sample_rate | This double precision value specifies the output data rate in samples / second (Hz) for multiple point operations. | | |
| reserved4[4] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| number_of_scans | This unsigned long integer value defines the number of times the channels specified in channel_array_ptr will be written. Setting number_of_cycles = 0 will cause the channels to be scanned continuously. | | |
| scan_event_level | This unsigned long integer value defines the frequency at which scan events are reported to the application. For example, setting scan_event_level to 100 causes a scan event to be generated each time 100 scans are completed. | | |
| reserved5[8] | This unsigned short integer array is reserved for the future expansion of DAQDRIVE. For maximum compatibility, the application should initialize all reserved variables to 0. | | |
| timeout_interval | This unsigned short integer value defines a time-out interval, in seconds, for foreground mode processes. The operation will abort if the digital output can not be updated every timeout_interval seconds. Setting timeout_interval = 0 disables the time-out function and causes the routine to wait indefinitely. | | |
| request_status | This unsigned long integer value provides the application with the current status of the request. DAQDRIVE does not rely on the information contained in this field nor does it ever clear any of the event bits to 0. Therefore, the application should initialize request_status during the configuration process and may modify its contents at any time. | | |
| | DAQDRIVE Constant | Value | Description |
| | NO_EVENTS | 0x00000000 | This constant does not represent an event status. It is provided to the application for convenience. |
| | TRIGGER_EVENT | 0x00000001 | When set to 1, this bit indicates the specified trigger has been received. |
| | COMPLETE_EVENT | 0x00000002 | When set to 1, this bit indicates the request has completed successfully. |
| | BUFFER_EMPTY_EVENT | 0x00000004 | When set to 1, this bit indicates at least one of the specified output data buffers has been emptied. |
| | BUFFER_FULL_EVENT | 0x00000008 | When set to 1, this bit indicates at least one of the specified input data buffers has been filled. |
| | SCAN_EVENT | 0x00000010 | When set to 1, this bit indicates the number of scans specified by scan_event_level have been completed at least once. |
| | USER_BREAK_EVENT | 0x20000000 | When set to 1, this bit indicates the request has terminated due to a user-break. |
| | TIMEOUT_EVENT | 0x40000000 | When set to 1, this bit indicates the request has terminated because the specified time-out interval was exceeded. |
| | RUNTIME_ERROR_EVENT | 0x80000000 | When set to 1, this bit indicates the request has terminated because of an error during processing. The application can determine the source of the error using the DaqGetRuntimeError procedure. |

Figure 17 (continued). Digital output request structure definition.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Output three 50 point patterns to three digital output channels.     */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[3] = { 0, 1, 5 };
unsigned short status;
unsigned char  data_array[150];

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for digital output.  *****/

/*************************************************************************/
/* data is in data_array     data_array is 150 points long               */
/* output buffer 10 times    next_structure = NULL  (no more structures) */
/*************************************************************************/

data_structure.data_buffer    = data_array;
data_structure.buffer_length  = 150;
data_structure.buffer_cycles  = 10;
data_structure.next_structure = NULL;

/*****  Prepare the digital output request structure.  *****/

/*************************************************************************/
/* channel list is in channel_num      channel_num is 3 channels long    */
/* data is in data_structure           trigger source is internal        */
/* trigger mode is continuous          output using IRQs (in background) */
/* use internal clock                  output 1 point every 500ms (2Hz)  */
/* repeat all buffers once             do not signal buffer scan events  */
/* do not implement time-out                                             */
/*************************************************************************/

user_request.channel_array_ptr = channel_num;
user_request.array_length      = 3;
user_request.DAC_buffer        = data_structure;
user_request.trigger_source    = INTERNAL_TRIGGER
user_request.trigger_mode      = CONTINUOUS_TRIGGER;
user_request.IO_mode           = BACKGROUND_IRQ;
user_request.clock_source      = INTERNAL_CLOCK;
user_request.sample_rate       = 2;
user_request.number_of_scans   = 1;
user_request.scan_event_level  = 0;
user_request.timeout_interval  = 0;
user_request.request_status    = NO_EVENTS;

/*****  Indicate data buffer ready for output.  *****/

data_structure.buffer_status = BUFFER_FULL;

/*****  Request digital output.  *****/

request_handle = 0;
status = DaqDigitalOutput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("Digital output request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

## 13.14 DaqFreeMemory (16-bit DAQDRIVE only)

DaqFreeMemory is a DAQDRIVE utility function used to free memory previously allocated by the DaqAllocateMemory procedure. All allocated memory should be freed before the application program terminates.

**NOTE:**
32-bit application programs must use the DaqFreeMemory32 procedure.

```
unsigned short DaqFreeMemory(unsigned short memory_handle,
                             void far *memory_pointer)
```

memory_handle - This unsigned short integer specifies the handle of the allocated memory block. This is the value returned by the DaqAllocateMemory procedure.

memory_pointer - This void pointer specifies the starting address of the allocated memory block. This is the value returned by the DaqAllocateMemory procedure.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[2]   = { 0, 1 };
         float gain_settings[2] = { 1, 8 };
unsigned long memory_size;
unsigned short memory_handle;
void far *memory_pointer;
unsigned short status;

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/
/*****  Allocate memory for the input data.  *****/
/*****  5000 samples/channel * 2 channels * 2 bytes/sample  *****/

memory_size = 5000 * 2 * sizeof(short);
status = DaqAllocateMemory(memory_size, &memory_handle, &memory_pointer);
if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Prepare data structure for analog input.  *****/

data_structure.data_buffer    = memory_pointer;
data_structure.buffer_length  = 10000;

/*****  Prepare the A/D request structure.  *****/
/*****  Request A/D input  (See DaqAnalogInput).  *****/
/*****  Arm the request (See DaqArmRequest).  *****/
/*****  Trigger the request (See DaqTriggerRequest).  *****/
/*****  Wait for complete.  *****/

/*****  Free allocated memory.  *****/

status = DaqFreeMemory(memory_handle, memory_pointer);
if (status != 0)
   {
   printf("Error de-allocating memory.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Close the device.  (See DaqCloseDevice).  *****/
```

## 13.15 DaqFreeMemory32 (32-bit DAQDRIVE only)

DaqFreeMemory32 is a DAQDRIVE utility function used to free memory previously allocated by the DaqAllocateMemory32 procedure. All allocated memory should be freed before the application program terminates.

**NOTE:**
16-bit application programs must use the DaqFreeMemory procedure.

```
unsigned short DaqFreeMemory32(unsigned long memory_handle,
                               void far *memory_pointer)
```

memory_handle - This unsigned long integer specifies the handle of the allocated memory block. This is the value returned by the DaqAllocateMemory32 procedure.

memory_pointer - This void pointer specifies the starting address of the allocated memory block. This is the value returned by the DaqAllocateMemory32 procedure.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num[2]   = { 0, 1 };
         float gain_settings[2] = { 1, 8 };
unsigned long  memory_size;
unsigned long  memory_handle;
void far *memory_pointer;
unsigned short status;

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/
/*****  Allocate memory for the input data.  *****/
/*****  5000 samples/channel * 2 channels * 2 bytes/sample  *****/

memory_size = 5000 * 2 * sizeof(short);
status = DaqAllocateMemory32(memory_size, &memory_handle, &memory_pointer);
if (status != 0)
   {
   printf("Error allocating data buffer.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Prepare data structure for analog input.  *****/
/*****  Prepare the A/D request structure.  *****/
/*****  Request A/D input  (See DaqAnalogInput).  *****/
/*****  Arm the request (See DaqArmRequest).  *****/
/*****  Trigger the request (See DaqTriggerRequest).  *****/
/*****  Wait for complete.  *****/

/*****  Free allocated memory.  *****/

status = DaqFreeMemory32(memory_handle, memory_pointer);
if (status != 0)
   {
   printf("Error de-allocating memory.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Close the device.  (See DaqCloseDevice).  *****/
}
```

## 13.16 DaqFreeRequest

DaqFreeRequest is a DAQDRIVE utility function used to free memory previously allocated by the DaqAllocateRequest procedure. All allocated memory should be freed before the application program terminates.

```
unsigned short DaqFreeRequest(unsigned short logical_device,
                              unsigned long memory_handle,
                              void far *memory_pointer)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

memory_handle - This unsigned long integer specifies the handle of the allocated memory block. This is the value returned by the DaqAllocateRequest procedure.

memory_pointer - This void pointer specifies the starting address of the allocated memory block. This is the value returned by the DaqAllocateRequest procedure.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 5000 points each from 2 analog input channels.                 */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;

struct ADC_request far *myADCrequest;
struct allocate_request memory_request;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Allocate memory for the ADC request and data structures.  *****/

memory_request.request_type         = ADC_TYPE_REQUEST;
memory_request.channel_array_length = 2;
memory_request.number_of_buffers    = 1;
memory_request.buffer_length        = 2 * 5000 * sizeof(short);
memory_request.buffer_attributes    = SEQUENTIAL_BUFFER;

status = DaqAllocateRequest(logical_device, &memory_request);
if (status != 0)
   {
   printf("Error allocating memory.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

myADCrequest = (ADC_request far *)memory_request.memory_pointer;

myADCrequest->channel_array_pointer[0] = 4;
myADCrequest->channel_array_pointer[1] = 9;
myADCrequest->trigger_source           = INTERNAL_TRIGGER;

/*****  Prepare remainder of A/D request structure.  *****/

/*****  Only status needs initialized in DAQDRIVE buffer structure.  *****/

myADCrequest->ADC_buffer->buffer_status = BUFFER_EMPTY;

/*****  Request A/D input.  *****/
/*****  Arm the request (See DaqArmRequest).  *****/
/*****  Trigger the request (See DaqTriggerRequest).  *****/
/*****  Wait for complete.  *****/
/*****  Free allocated memory.  (See DaqFreeRequest)  *****/

status = DaqFreeRequest(logical_device,
                        memory_request.memory_handle,
                        memory_request.memory_pointer)

if (status != 0)
   printf("Error de-allocating memory.  Status code %d.\n",status);

/*****  Close the device.  (See DaqCloseDevice).  *****/
}
```

## 13.17 DaqGetADCfgInfo

DaqGetADCfgInfo returns the configuration of the A/D converter specified by ADC_device on the adapter specified by logical_device.

```
unsigned short DaqGetADCfgInfo(unsigned short logical_device,
                               unsigned short ADC_device,
                               struct ADC_configuration far *ADC_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_device - This unsigned short integer value is used to select one of the A/D converters on the target hardware device.

ADC_info - This structure pointer defines the address of an A/D configuration structure where the configuration of the specified A/D converter will be stored.

```
struct ADC_configuration
   {
   unsigned short resolution;
   unsigned short signal_type;
   unsigned short input_mode;
   unsigned short data_coding;
            long  min_digital;
            long  max_digital;
            long  zero_offset;
           float min_analog;
           float max_analog;
           float min_sample_rate;
           float max_sample_rate;
           float max_scan_rate;
   unsigned short num_exp_boards;
   unsigned short total_channels;
   unsigned short max_scan_length;
   unsigned short gain_array_length;
   unsigned short calibration_modes;
   };
```

| | | |
|---|---|---|
| resolution | This unsigned short integer value specifies the resolution of the A/D converter in bits. | |
| signal_type | This unsigned short integer value specifies the A/D input signal type. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the A/D input is bipolar. |
| | 0x0002 | When set to 1, this bit indicates the A/D input is unipolar. |
| input_mode | This unsigned short integer value specifies the A/D input mode. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the A/D input is differential. |
| | 0x0002 | When set to 1, this bit indicates the A/D input is single-ended. |
| data_coding | This unsigned short integer value specifies the A/D data coding format. | |
| | Value | Description |
| | 0 | Indicates data is in two's complement format. |
| | 1 | Indicates data is in binary format. |
| min_digital | This long integer value defines the minimum digital value returned by the A/D. | |
| max_digital | This long integer value defines the maximum digital value returned by the A/D. | |
| zero_offset | This long integer value defines the offset or zero value reading returned by the A/D. | |
| min_analog | This floating point value defines the minimum analog input to the A/D. | |
| max_analog | This floating point value defines the maximum analog input to the A/D. | |
| min_sample_rate | This floating point value specifies the minimum sampling rate supported by the A/D. | |
| max_sample_rate | This floating point value specifies the maximum single channel sampling rate supported by the A/D. | |
| max_scan_rate | This floating point value specifies the maximum multi-channel (scanning) sampling rate supported by the A/D. | |
| num_exp_boards | This unsigned short integer value defines the number of analog input expansion boards connected to the A/D. | |
| total_channels | This unsigned short integer value specifies the total number of analog inputs available on the A/D. | |
| max_scan_length | This unsigned short integer value defines the maximum scan length of the A/D. This is the maximum length of the channel list for A/D requests. | |
| gain_array_length | This unsigned short integer value specifies the number of available A/D gain settings. The application must allocate an array of length gain_array_length before executing DaqGetADGainInfo. | |
| calibration_modes | This unsigned short integer value specifies the supported calibration modes of the A/D sub-system. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the A/D supports auto-calibration. |
| | 0x0002 | When set to 1, this bit indicates the A/D supports auto-zero. |

Figure 19. A/D converter configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_devices;
unsigned short status;

struct ADC_configuration ADC_info;

char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*****  Open the DAQ-1201  (see DaqOpenDevice).  *****/

/*****  Get the A/D configuration.  *****/

ADC_device = 0;
status = DaqGetADCfgInfo(logical_device, ADC_device, &ADC_info);
if (status != 0)
   {
   printf("Error getting A/D configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the A/D configuration.  *****/

printf("A/D number %d ", ADC_device);
printf("has a resolution of %d bits,\n", ADC_info.resolution);
if (ADC_info.signal_type == 1)
   printf("is configured for unipolar and ");
else
   printf("is configured for bipolar and ");
if (ADC_info.input_mode == 1)
   printf("single-ended operation,\n");
else
   printf("differential operation,\n");
switch (ADC_info.calibration_modes)
   {
    case 1:  printf("supports auto-calibration,\n");
             break;
    case 2:  printf("supports auto-zero,\n");
             break;
    case 3:  printf("supports auto-calibration and auto-zero,\n");
             break;
   }
printf("has a max scan length of %d channels,\n", ADC_info.max_scan_length);
printf("supports %d gain settings,\n", ADC_info.gain_array_length);
printf("and has %d expansion boards attached ", ADC_info.num_exp_boards);
printf("for a total of %d analog inputs.\n", ADC_info.total_inputs);
printf("\n");
printf("The A/D returns values in the range %ld ", ADC_info.min_digital);
printf("to %ld\n", ADC_info.max_digitla);
printf("which corresponds to an input range of %f ", ADC_info.min_analog);
printf("to %f volts.\n", ADC_info.max_analog);
printf("\n");
printf("A single input may be sampled up to %f Hz ", ADC_info.max_sample_rate);
printf("and multiple inputs up to %f Hz.\n", ADC_info.max_scan_rate);
printf("The minimum sampling rate is %f Hz\n", ADC_info.min_sample_rate);
}
```

## 13.18 DaqGetAddressOf

DaqGetAddressOf is a DAQDRIVE utility function used to get the address of a variable for programming language which do not support pointers. The variable's address is returned as an unsigned long integer suitable for storage in any of the DAQDRIVE structures.

```
unsigned long DaqGetAddressOf(void far *variable)
```

variable - This void pointer specifies the variable for which the address is desired. variable is declared as a void to allow it to point to data of any type.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{

char far *device_type = "DAQ-1201";
unsigned short channel_array[3] = {0, 1, 2};

unsigned long address_of_string;
unsigned long address_of_array;

address_of_string = DaqAddressOf((void far*)device_type);
address_of_array  = DaqAddressOf((void far*)&channel_array[0]);
}
```

## 13.19 DaqGetADGainInfo

DaqGetADGainInfo returns an array of the gain settings supported by the A/D converter specified by ADC_device on the adapter specified by logical_device. The length of the array is determined by the gain_array_length variable returned by the DaqGetADCfgInfo command.

```
unsigned short DaqGetADGainInfo(unsigned short logical_device,
                                unsigned short ADC_device,
                                float far *gain_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_device - This unsigned short integer value is used to select one of the A/D converters on the target hardware device.

gain_array - This pointer defines the first element of an array of floating point values where the available A/D gain settings will be stored. The application must allocate the array used to store these gain settings. The length of the array is determined by the gain_array_length variable returned by the DaqGetADCfgInfo command.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_device;
unsigned short status;
unsigned short i;

struct ADC_configuration ADC_info;

float far *gain_array;

char far *device_type = "DAQ-1201";
char far *config_file = "daq-1201.dat";

/*****  Open the DAQ-1201  (see DaqOpenDevice).  *****/

/*****  Get the A/D configuration.  *****/

ADC_device = 0;
status = DaqGetADCfgInfo(logical_device, ADC_device, &ADC_info);

/*****  Create an array to hold the gain settings.  *****/

gain_array = _fmalloc(ADC_info.gain_array_length * sizeof(float));

/*****  Get the available gain settings.  *****/

status = DaqGetADGainInfo(logical_device, ADC_device, gain_array);
if (status != 0)
   {
   printf("Error getting A/D gain settings.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the available gain settings.  *****/

printf("The DAQ-1201 supports the following A/D gain settings:\n");

for (i = 0; i < ADC_info.gain_array_length; i++)
   printf("     gain[%d] = %f\n", i, gain_array[i]);
}
```

## 13.20 DaqGetDACfgInfo

DaqGetDACfgInfo returns the configuration of the D/A converter specified by DAC_device on the adapter specified by logical_device.

```
unsigned short DaqGetDACfgInfo(unsigned short logical_device,
                               unsigned short DAC_device,
                               struct DAC_configuration far *DAC_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

DAC_device - This unsigned short integer value is used to select one of the D/A converters on the target hardware device.

DAC_info - This structure pointer defines the address of a D/A configuration structure where the configuration of the specified D/A converter will be stored.

```
struct DAC_configuration
   {
   unsigned short resolution;
   unsigned short signal_type;
   unsigned short data_coding;
            long  min_digital;
            long  max_digital;
            long  zero_offset;
            float min_analog;
            float max_analog;
            float min_sample_rate;
            float max_sample_rate;
            float max_scan_rate;
   unsigned short reference_source;
            float reference_voltage;
   unsigned short gain_array_length;
   unsigned short calibration_modes;
   };
```

| | | |
|---|---|---|
| resolution | This unsigned short integer value specifies the resolution of the D/A converter in bits. | |
| signal_type | This unsigned short integer value specifies the D/A output signal type. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the D/A output is bipolar. |
| | 0x0002 | When set to 1, this bit indicates the D/A output is unipolar. |
| data_coding | This unsigned short integer value specifies the D/A data coding format. | |
| | Value | Description |
| | 0 | Indicates data is in two's complement format. |
| | 1 | Indicates data is in binary format. |
| min_digital | This long integer value defines the minimum digital value accepted by the D.A. | |
| max_digital | This long integer value defines the maximum digital value accepted by the D/A. | |
| zero_offset | This long integer value defines the offset or zero value of the D/A. | |
| min_analog | This floating point value defines the minimum analog output from the D/A. | |
| max_analog | This floating point value defines the maximum analog output from the D/A. | |
| min_sample_rate | This floating point value specifies the minimum sampling rate supported by the D/A. | |
| max_sample_rate | This floating point value specifies the maximum single channel sampling rate supported by the D/A. | |
| max_scan_rate | This floating point value specifies the maximum multi-channel (scanning) sampling rate supported by the D/A. | |
| reference_source | This unsigned short integer value defines the source of the D/A's reference voltage. | |
| reference_voltage | This floating point value specifies the value of the D/A's reference voltage. | |
| gain_array_length | This unsigned short integer value specifies the number of available D/A gain settings. The application must allocate an array of length gain_array_length before executing DaqGetDAGainInfo. | |
| calibration_modes | This unsigned short integer value specifies the supported calibration modes of the D/A sub-system. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the D/A supports auto-calibration. |
| | 0x0002 | When set to 1, this bit indicates the D/A supports auto-zero. |

Figure 20. D/A converter configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short DAC_device;
unsigned short status;

struct DAC_configuration DAC_info;
char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B (see DaqOpenDevice).  *****/

/*****  Get a D/A configuration.  *****/

DAC_device = 6;
status = DaqGetDACfgInfo(logical_device, DAC_device, &DAC_info);
if (status != 0)
   {
   printf("Error getting D/A configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the D/A configuration.  *****/

printf("D/A number %d ", DAC_device);
printf("has a resolution of %d bits,\n", DAC_info.resolution);
if (DAC_info.signal_type == 1)
   printf("is configured for unipolar operation,\n");
else
   printf("is configured for bipolar operation,\n");
if (DAC_info.reference_source == 0)
   printf("with an internal reference voltage");
else
   printf("with an external reference voltage");
printf("of %2.5f volts.\n", DAC_info.reference_voltage);
printf("It supports digital values from %ld ", DAC_info.min_digital);
printf("to %ld\n", DAC_info.max_digital);
printf("which produce analog voltages from %f ", DAC_info.min_analog);
printf("to %f volts.\n", DAC_info.max_analog);
printf("Data can be output up to %f Hz", DAC_info.max_sample_rate);
printf("for a single channel or up to %f Hz", DAC_info.max_scan_rate);
printf("on multiple channels.");
printf("The D/A supports %d gain settings\n", DAC_info.gain_array_length);
switch (DAC_into.calibration_modes)
   {
   case 0:  printf("but does not offer self-calibration.\n");
            break;
   case 1:  printf("and an auto-calibration mode.\n");
            break;
   case 2:  printf("and an auto-zero mode.\n");
            break;
   case 3:  printf("plus auto-calibration and auto-zero modes.\n");
            break;
   }
}
```

## 13.21 DaqGetDAGainInfo

DaqGetDAGainInfo returns an array of the gain settings supported by the D/A converter specified by DAC_device on the adapter specified by logical_device. The length of the array is determined by the gain_array_length variable returned by the DaqGetDACfgInfo command.

```
unsigned short DaqGetDAGainInfo(unsigned short logical_device,
                                unsigned short DAC_device,
                                float far *gain_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

DAC_device - This unsigned short integer value is used to select one of the D/A converters on the target hardware device.

gain_array - This pointer defines the first element of an array of floating point values where the available D/A gain settings will be stored. The application must allocate the array used to store these gain settings. The length of the array is determined by the gain_array_length variable returned by the DaqGetDACfgInfo command.

```
#include "daqdrive.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short DAC_device;
unsigned short status;
unsigned short i;

struct DAC_configuration DAC_info;

float ref_voltage;
float far *gain_array;

char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B (see DaqOpenDevice).  *****/

/*****  Get the D/A configuration.  *****/

DAC_device = 1;
status = DaqGetDACfgInfo(logical_device, DAC_device, &DAC_info);

/*****  Create an array to hold the gain settings.  *****/

gain_array = _fmalloc(DAC_info.gain_array_length * sizeof(float));

/*****  Get the available gain settings.  *****/

status = DaqGetDAGainInfo(logical_device, DAC_device, gain_array);
if (status != 0)
   {
   printf("Error getting D/A gain settings.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the available gain settings.  *****/

printf("The DA8P-12B supports the following D/A gain settings:\n");

for (i = 0; i < DAC_info.gain_array_length; i++)
   printf("     gain[%d] = %f\n", i, gain_array[i]);
}
```

## 13.22 DaqGetDeviceCfgInfo

DaqGetDeviceCfgInfo returns the basic configuration of the adapter specified by logical_device.

```
unsigned short DaqGetDeviceCfgInfo(unsigned short logical_device,
                                   struct device_configuration far *dev_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

dev_info - This structure pointer defines the address of a device configuration structure where the configuration of the specified logical device will be stored.

```
struct device_configuration
   {
   unsigned short base_address;
            short IRQ;
            short DMA1;
            short DMA2;
   unsigned short ADC_devices;
   unsigned short DAC_devices;
   unsigned short digio_devices;
   unsigned short timer_devices;
   };
```

| | |
|---|---|
| base_address | This unsigned short integer value specifies the base I/O address of the device. |
| IRQ | This unsigned short integer value specifies the IRQ level for the device. A value of -1 indicates no IRQ level is defined. |
| DMA1 | This unsigned short integer value specifies the primary DMA channel for the device. A value of -1 indicates no DMA channel is defined. |
| DMA2 | This unsigned short integer value specifies the secondary DMA channel for the device. A value of -1 indicates no DMA channel is defined. |
| ADC_devices | This unsigned short integer value specifies the number of A/D converters on the device. |
| DAC_devices | This unsigned short integer value specifies the number of D/A converters on the device. |
| digio_devices | This unsigned short integer value specifies the number of digital I/O channels on the device. |
| timer_devices | This unsigned short integer value specifies the number of counter / timer channels on the device. |

Figure 21. Device configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

struct device_configuration dev_info;

char far *device_type = "DAQP-208";
char far *config_file = "c:\\daqp-208\\daqp-208.dat";

/*****  Open the DAQP-208 (see DaqOpenDevice).  *****/

/*****  Get the DAQP-208 configuration.  *****/

status = DaqGetDeviceCfgInfo(logical_device, &dev_info);
if (status != 0)
   {
   printf("Error getting device configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the DAQP-208 configuration.  *****/

printf("The DAQP-208 is located at address %4XH,\n", dev_info.base_address);
printf("with interrupt level %d,\n", dev_info.IRQ);
printf("and DMA channels %d and %d.\n\n", dev_info.DMA1, dev_info.DMA2);

printf("The DAQP-208 contains %d A/D converter(s),\n", dev_info.ADC_devices);
printf("%d D/A converter(s),\n", dev_info.DAC_devices);
printf("%d digital I/O device(s),\n", dev_info.digio_devices);
printf("and %d counter/timer channel(s).\n", dev_info.timer_devices);
}
```

## 13.23 DaqGetDigioCfgInfo

DaqGetDigioCfgInfo returns the configuration of the digital I/O channel specified by digio_device on the adapter specified by logical_device.

```
unsigned short DaqGetDigioCfgInfo(unsigned short logical_device,
                                  unsigned short digio_device,
                                  struct digio_configuration far *digio_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

digio_device - This unsigned short integer value is used to select one of the digital I/O channels on the target hardware device.

digio_info - This structure pointer defines the address of a digital I/O configuration structure where the configuration of the specified digital I/O channel will be stored.

```
struct digio_configuration
   {
   unsigned short data_size;
   unsigned short io_mode;
   };
```

| | | |
|---|---|---|
| data_size | This unsigned short integer value specifies the size of the digital I/O channel in bits. | |
| io_mode | This unsigned short integer value specifies the operating mode of the digital I/O channel. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the digital I/O channel is configured for input mode. |
| | 0x0002 | When set to 1, this bit indicates the digital I/O channel is configured for output mode. |
| | 0x0003 | When both bits are set to 1, the digital I/O channel can operate in input or output mode (bi-directional operation). |

Figure 22. Digital I/O configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
include "iop241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short digio_device;
unsigned short status;

struct digio_configuration digio_info;

char far *device_type = "IOP-241";
char far *config_file = "c:\\iop-241\\iop-241.dat";

/*****  Open the IOP-241 (see DaqOpenDevice).  *****/

/*****  Get a digital I/O channel configuration.  *****/

digio_device = 0;
status = DaqGetDigioCfgInfo(logical_device, digio_device, &digio_info);
if (status != 0)
   {
   printf("Error getting digital configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the digital I/O configuration.  *****/

printf("Digital I/O channel %d ", digio_device);
printf("is %d bits wide,\n", digio_info.data_size);
switch(digio_info.io_mode)
   {
   case 1:  printf("and is configured for input mode.\n");
            break;
   case 2:  printf("and is configured for output mode.\n");
            break;
   case 3:  printf("and is configured for bi-directional operation.\n");
            break;
   }
}
```

## 13.24 DaqGetExpCfgInfo

DaqGetExpCfgInfo returns the configuration of the expansion board specified by exp_device which is connected to the A/D converter specified by ADC_device on the adapter specified by logical_device.

```
unsigned short DaqGetExpCfgInfo(unsigned short logical_device,
                                unsigned short ADC_device,
                                unsigned short exp_device,
                                struct exp_configuration far *exp_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_device - This unsigned short integer value is used to select one of the A/D converters on the target hardware device.

exp_device - This unsigned short integer value is used to select one of the expansion boards connected to the A/D converter on the target hardware device.

exp_info - This structure pointer defines the address of an expansion board configuration structure where the configuration of the specified expansion board will be stored.

```
struct exp_configuration
   {
   unsigned short signal_type;
   unsigned short input_mode;
   unsigned short num_mux_channels;
            float max_sample_rate;
            float max_scan_rate;
   unsigned short gain_array_length;
   };
```

| | | |
|---|---|---|
| signal_type | This unsigned short integer value specifies the expansion board input signal type. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the expansion board input is bipolar. |
| | 0x0002 | When set to 1, this bit indicates the expansion board input is unipolar. |
| input_mode | This unsigned short integer value specifies the expansion board input mode. | |
| | Value | Description |
| | 0x0001 | When set to 1, this bit indicates the expansion board input is differential. |
| | 0x0002 | When set to 1, this bit indicates the expansion board input is single-ended. |
| num_mux_channels | This unsigned short integer value specifies the number of multiplexer channels on the expansion board. | |
| max_sample_rate | This floating point value specifies the maximum single channel sampling rate supported by the expansion board. | |
| max_scan_rate | This floating point value specifies the maximum multi-channel (scanning) sampling rate supported by the expansion board. | |
| gain_array_length | This unsigned short integer value specifies the number of available expansion board gain settings. The application must allocate an array of length gain_array_length before executing DaqGetExpGainInfo. | |

Figure 23. Analog input expansion board configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_device;
unsigned short exp_device;
unsigned short status;

struct exp_configuration exp_info;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201 (see DaqOpenDevice).  *****/

/*****  Get the expansion board configuration.  *****/

ADC_device = 0;
exp_device = 0;
status = DaqGetExpCfgInfo(logical_device, ADC_device, exp_device, &exp_info);
if (status != 0)
   {
   printf("Error getting exp. board configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the expansion board configuration.  *****/

printf("Expansion board number %d on A/D number %d\n", exp_device, ADC_device);
if (exp_info.signal_type == 1)
   printf("is configured for unipolar and ");
else
   printf("is configured for bipolar and ");
if (exp_info.input_mode == 1)
   printf("single-ended operation,\n");
else
   printf("differential operation,\n");
printf("has %d analog inputs,\n", exp_info.num_mux_channels);
printf("and supports %d gain settings.\n", exp_info.gain_array_length);
}
```

## 13.25 DaqGetExpGainInfo

DaqGetExpGainInfo returns an array of the gain settings supported by the expansion board specified by exp_device connected to the A/D converter specified by ADC_device on the adapter specified by logical_device. The length of the array is determined by the gain_array_length variable returned by the DaqGetExpCfgInfo command.

```
unsigned short DaqGetExpGainInfo(unsigned short logical_device,
                                 unsigned short ADC_device,
                                 unsigned short exp_device,
                                 float far *gain_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_device - This unsigned short integer value is used to select one of the A/D converters on the target hardware device.

exp_device - This unsigned short integer value is used to select one of the expansion boards connected to the A/D converter on the target hardware device.

gain_array - This pointer defines the first element of an array of floating point values where the available expansion board gain settings will be stored. The application must allocate the array used to store these gain settings. The length of the array is determined by the gain_array_length variable returned by the DaqGetExpCfgInfo command.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_device;
unsigned short exp_device;
unsigned short status;
unsigned short i;

struct exp_configuration exp_info;

float far *gain_array;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201 (see DaqOpenDevice).  *****/

/*****  Get the expansion board configuration.  *****/

ADC_device = 0;
exp_device = 1;
status = DaqGetExpCfgInfo(logical_device, ADC_device, exp_device, &exp_info);

/*****  Create an array to hold the gain settings.  *****/

gain_array = _fmalloc(exp_info.gain_array_length * sizeof(float));

/*****  Get the available gain settings.  *****/

status = DaqGetExpGainInfo(logical_device, ADC_device, exp_device, gain_array);
if (status != 0)
   {
   printf("Error getting expansion board gain settings.\n");
   printf("Status code %d.\n",status);
   exit(status);
   }

/*****  Display the available gain settings.  *****/

printf("Expansion board #%d supports the following gains:\n", exp_device);

for (i = 0; i < exp_info.gain_array_length; i++)
   printf("     gain[%d] = %f\n", i, gain_array[i]);
}
```

## 13.26 DaqGetRuntimeError

DaqGetRuntimeError returns the last run-time error encountered by the request specified by request_handle.

```
unsigned short DaqGetRuntimeError(unsigned short request_handle,
                                  unsigned short far *error_code)
```

request_handle - This unsigned short integer variable is used to define which request's error status to retrieve. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

error_code - This pointer defines an unsigned short integer where the error code from the last run-time error will be stored. Chapter 14 provides an explanation of these error codes. DAQDRIVE resets the request's error_code to 0 each time the request is armed.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Output a 20 point waveform to a D/A channel.                         */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short error_code;
unsigned short i;
         short data_array[20];
unsigned long  event_mask;

struct DAC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  define D/A output channel and calculate output data.  *****/

/*****  Prepare data structure for analog output.  *****/

/*****  Prepare the D/A request structure.  *****/

/*****  Request D/A output (See DaqAnalogOutput).  *****/

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request.  *****/

/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & COMPLETE_EVENT) != 0)
   printf("Request complete.\n");
else
   {
   status = DaqGetRuntimeError(request_handle, &error_code);
   printf("Run-time error #%d.  Request aborted.\n", error_code);
   }

/*****  Release the request.  *****/

status = DaqReleaseRequest(request_handle);
if (status != 0)
   printf("Could not release configuration.  Status code %d.\n"),status);

/*****  Close the device.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.27 DaqGetSigConCfgInfo

DaqGetSigConCfgInfo returns the configuration of the signal conditioner attached to the analog input channel specified by ADC_channel on the adapter specified by logical_device.

```
unsigned short DaqGetSigConCfgInfo(unsigned short logical_device,
                                   unsigned short ADC_channel,
                                   struct sigcon_configuration far *sigcon_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_channel - This unsigned short integer value is used to select one of the analog input channels on the target hardware device.

sigcon_info - This structure pointer defines the address of a signal conditioner configuration structure where the configuration of the specified signal conditioner will be stored.

```
struct sigcon_configuration
   {
   unsigned long  device_type;
            float max_sample_rate;
            float max_scan_rate;
            float min_analog_input;
            float max_analog_input;
   unsigned short input_units;
            float min_analog_output;
            float max_analog_output;
   unsigned short num_parameters;
   };
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| device_type | This unsigned short integer value is currently unused. | | | | | | | | | |
| max_sample_rate | This floating point value specifies the maximum single channel sampling rate supported by the signal conditioner. | | | | | | | | | |
| max_scan_rate | This floating point value specifies the maximum multi-channel (scanning) sampling rate supported by the expansion board. | | | | | | | | | |
| min_analog_input | This floating point value specifies the minimum value that can be recorded by this signal conditioner. The input unit of measure is determined by input_units. | | | | | | | | | |
| max_analog_input | This floating point value specifies the maximum value that can be recorded by this signal conditioner. The input unit of measure is determined by input_units. | | | | | | | | | |
| input_units | This unsigned short integer value specifies the expansion board input mode. | | | | | | | | | |
| | Value | Units | Value | Units | Value | Units | Value | Units | Value | Units |
| | 0 | V | 6 | Hz | 12 | G | 18 | ppm | 24 | mm |
| | 1 | A | 7 | ME | 13 | mV | 19 | ppb | 25 | inch / sec |
| | 2 | ºC | 8 | RPM | 14 | mA | 20 | ft / sec | 26 | CFM |
| | 3 | ºF | 9 | ft / sec2 | 15 | ºK | 21 | l / sec | 27 | GPM |
| | 4 | Kg | 10 | m / sec2 | 16 | lbs | 22 | in | 28 | LPM |
| | 5 | g | 11 | % | 17 | psi | 23 | ft | | |
| min_analog_output | This floating point value specifies the minimum output voltage produced by this signal conditioner. This is the value produced when the input is min_analog_input. | | | | | | | | | |
| max_analog_output | This floating point value specifies the maximum output voltage produced by this signal conditioner. This is the value produced when the input is max_analog_input. | | | | | | | | | |
| num_parameters | This unsigned short integer value specifies the number of parameters required to mathematically model this signal conditioner. These parameters are the coefficients of a (num_parameters - 1) order equation. The application must allocate an array of length num_parameters before executing DaqGetSigConParamInfo. | | | | | | | | | |

Figure 24. Analog input signal conditioner board configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_device;
unsigned short exp_device;
unsigned short status;

struct exp_configuration exp_info;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201 (see DaqOpenDevice).  *****/

/*****  Get the expansion board configuration.  *****/

ADC_device = 0;
exp_device = 0;
status = DaqGetExpCfgInfo(logical_device, ADC_device, exp_device, &exp_info);
if (status != 0)
   {
   printf("Error getting exp. board configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the expansion board configuration.  *****/

printf("Expansion board number %d on A/D number %d\n", exp_device, ADC_device);
if (exp_info.signal_type == 1)
   printf("is configured for unipolar and ");
else
   printf("is configured for bipolar and ");
if (exp_info.input_mode == 1)
   printf("single-ended operation,\n");
else
   printf("differential operation,\n");
printf("has %d analog inputs,\n", exp_info.num_mux_channels);
printf("and supports %d gain settings.\n", exp_info.gain_array_length);
}
```

## 13.28 DaqGetSigConParamInfo

DaqGetSigConParamInfo returns an array of parameters for the signal conditioner attached to the analog input specified by ADC_channel on the adapter specified by logical_device. These parameters are the coefficients of a (num_parameters - 1) order equation used to mathematically model this signal conditioner. The length of the array is determined by the num_parameters variable returned by the DaqGetSigConCfgInfo command.

```
unsigned short DaqGetSigConParamInfo(unsigned short logical_device,
                                     unsigned short ADC_channel,
                                     double far *parameter_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

ADC_channel - This unsigned short integer value is used to select one of the analog input channels on the target hardware device.

parameter_array - This pointer defines the first element of an array of double precision floating point values where the signal conditioner parameters will be stored. The application must allocate the array used to store these parameters. The length of the array is determined by the num_parameters variable returned by the DaqGetSigConCfgInfo command.

```
#include "daqdrive.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short ADC_device;
unsigned short exp_device;
unsigned short status;
unsigned short i;

struct exp_configuration exp_info;

float far *gain_array;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201 (see DaqOpenDevice).  *****/

/*****  Get the expansion board configuration.  *****/

ADC_device = 0;
exp_device = 1;
status = DaqGetExpCfgInfo(logical_device, ADC_device, exp_device, &exp_info);

/*****  Create an array to hold the gain settings.  *****/

gain_array = _fmalloc(exp_info.gain_array_length * sizeof(float));

/*****  Get the available gain settings.  *****/

status = DaqGetExpGainInfo(logical_device, ADC_device, exp_device, gain_array);
if (status != 0)
   {
   printf("Error getting expansion board gain settings.\n");
   printf("Status code %d.\n",status);
   exit(status);
   }

/*****  Display the available gain settings.  *****/

printf("Expansion board #%d supports the following gains:\n", exp_device);

for (i = 0; i < exp_info.gain_array_length; i++)
   printf("     gain[%d] = %f\n", i, gain_array[i]);
}
```

## 13.29 DaqGetTimerCfgInfo

DaqGetTimerCfgInfo returns the configuration of the counter / timer channel specified by timer_device on the adapter specified by logical_device.

```
unsigned short DaqGetTimerCfgInfo(unsigned short logical_device,
                                  unsigned short timer_device,
                                  struct timer_configuration far *timer_info)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

timer_device - This unsigned short integer value is used to select one of the counter / timer channels on the target hardware device.

timer_info - This structure pointer defines the address of a counter / timer configuration structure where the configuration of the specified counter / timer channel will be stored.

```
struct timer_configuration
   {
   unsigned short  data_size;
            double internal_clock_rate;
            double min_rate;
            double max_rate;
   };
```

| | |
|---|---|
| data_size | This unsigned short integer value specifies the size of the counter/timer channel in bits. |
| internal_clock_rate | This double precision floating point value specifies the frequency of the on-board clock input to the counter/timer. |
| min_rate | This double precision floating point value specifies the minimum output frequency of the counter/timer when using the internal clock source. |
| max_rate | This double precision floating point value specifies the maximum output frequency of the counter/timer when using the internal clock source. |

Figure 25. Counter/timer configuration structure definition.

```
#include "daqdrive.h"
#include "userdata.h"
#include "iop241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short timer_device;
unsigned short status;

struct timer_configuration timer_info;

char far *device_type = "IOP-241";
char far *config_file = "c:\\iop-241\\iop-241.dat";

/*****  Open the IOP-241 (see DaqOpenDevice).  *****/

/*****  Get a digital I/O channel configuration.  *****/

digio_device = 0;
status = DaqGetTimerCfgInfo(logical_device, timer_device, &timer_info);
if (status != 0)
   {
   printf("Error getting digital configuration.  Status code %d.\n",status);
   exit(status);
   }

/*****  Display the digital I/O configuration.  *****/

printf("Counter timer channel %d ", timer_device);
printf("is %d bits wide,\n", timer_info.data_size);
printf("has an internal clock rate of %f \n", timer_info.internal_clock_rate);
printf("Hz, which can produce output rates between %f", timer_info.min_rate);
printf("and %f Hz.\n", timer_info.max_rate);
}
```

## 13.30 DaqNotifyEvent

DaqNotifyEvent allows the application program to install a procedure that DAQDRIVE will execute each time an event occurs and should be executed before the request is armed. The format of the command is shown below.

```
unsigned short DaqNotifyEvent(unsigned short request_handle,
                              void (far pascal *event_procedure)
                                                (unsigned short,
                                                 unsigned short,
                                                 unsigned short),
                              unsigned long event_mask)
```

request_handle - This unsigned short integer variable is used to define which request is to use the event procedure defined by event_procedure. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput or DaqDigitalOutput.

event_procedure - This pointer defines the starting address of the procedure to be executed when an event occurs. event_procedure is defined in the following section.

event_mask - This unsigned long integer value is used to specify which events the application wishes to be notified of. event_mask is defined as a bit mask - setting a specific bit to logic 1 enables notification of the corresponding event. The bit definitions of event mask are given below.

| DAQDRIVE Constant | Value | Description |
|---|---|---|
| NO_EVENTS | 0x00000000 | Disable all event notification. |
| TRIGGER_EVENT | 0x00000001 | Enable notification of trigger events. |
| COMPLETE_EVENT | 0x00000002 | Enable notification of complete events. |
| BUFFER_EMPTY_EVENT | 0x00000004 | Enable notification of buffer empty events. |
| BUFFER_FULL_EVENT | 0x00000008 | Enable notification of buffer full events. |
| SCAN_EVENT | 0x00000010 | Enable notification of scan events. |
| USER_BREAK_EVENT | 0x20000000 | Enable notification of user break events. |
| TIMEOUT_EVENT | 0x40000000 | Enable notification of time-out events. |
| RUNTIME_ERROR_EVENT | 0x80000000 | Enable notification of run-time error events. |

### 13.30.1 The user-defined event procedure

The application programmer must create the procedure to be executed for event notification. This procedure must be a far pascal compatible procedure of type void (does not return a value) and it must accept three unsigned short integer parameters: request_handle, event_type, and error_code. A sample C declaration of this procedure is shown below.

```
void far pascal event_procedure(unsigned short request_handle,
                                unsigned short event_type,
                                unsigned short error_code)
```

When executed, DAQDRIVE provides the event procedure with the request's request_handle, the type of event which has occurred (see the table below), and an event error code. This error code is set to 0 for all events except the run-time error event where it is used to specify the type of error encountered as defined in chapter 14. Since the request_handle is provided to the event procedure, a single event procedure may be used to service events from multiple requests.

| DAQDRIVE Constant | Value | Description |
|---|---|---|
| EVENT_TYPE_TRIGGER | 0 | This call to the notification procedure is the result of a trigger event. |
| EVENT_TYPE_COMPLETE | 1 | This call to the notification procedure is the result of a complete event. |
| EVENT_TYPE_BUFFER_EMPTY | 2 | This call to the notification procedure is the result of a buffer empty event. |
| EVENT_TYPE_BUFFER_FULL | 3 | This call to the notification procedure is the result of a buffer full event. |
| EVENT_TYPE_SCAN | 4 | This call to the notification procedure is the result of a scan event. |
| EVENT_TYPE_USER_BREAK | 29 | This call to the notification procedure is the result of a user break event. |
| EVENT_TYPE_TIMEOUT | 30 | This call to the notification procedure is the result of a time-out event. |
| EVENT_TYPE_RUNTIME_ERROR | 31 | This call to the notification procedure is the result of a run-time error event. |

```
#include "daqdrive.h"
#include "userdata.h"

/*****  Define an event procedure  *****/

void far pascal my_event_procedure(unsigned short request_handle,
                                   unsigned short event_type,
                                   unsigned short error_code)
{
switch(event_type)
   {
   case EVENT_TYPE_TRIGGER:
      /*****  process trigger events  *****/
      break;

   case EVENT_TYPE_COMPLETE:
      /*****  process complete events  *****/
      break;

   case EVENT_TYPE_RUNTIME_ERROR:
      /*****  process run-time error events  *****/
      break;
   }
}

/*****  Define the main procedure  *****/

void main()
{
unsigned short request_handle;
unsigned short status;
unsigned long  event_mask;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Request an operation. (gets a request_handle)  *****/

/*****  Define events to be notified.  *****/

event_mask = TRIGGER_EVENT | COMPLETE_EVENT | RUNTIME_ERROR_EVENT;

/*****  Install notification procedure.  *****/

status = DaqNotifyEvent(request_handle, my_event_procedure, event_mask);
if (status != 0)
   printf("Error installing notification.\n");

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/
```

## 13.31 DaqOpenDevice

DaqOpenDevice reads the adapter description file generated by the DAQDRIVE configuration utilities, initializes the hardware device according to the contents of the file, and prepares DAQDRIVE for use with the device.

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The following sections describe the DaqOpenDevice procedure for linking C applications directly to the DAQDRIVE libraries, for applications using DAQDRIVE under Windows, and for applications using the DOS memory resident (TSR) version of DAQDRIVE.

### 13.31.1 DaqOpenDevice - C Library Version

The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries. Consult the target hardware's appendix for the required settings of PROCEDURE, the device_type variable, and the name of the include file (.h) which defines the open command for the target device.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short far *logical_device,
                             char far *device_type,
                             char far *config_file)
```

PROCEDURE - This is a constant used by the macro to define the "open" procedure of the driver to be accessed and must be entered exactly as it is defined in the target hardware's appendix. PROCEDURE is used with the token pasting operator to generate a unique "open" procedure for each type of hardware device.

logical_device - This pointer specifies the address of an unsigned short integer where the logical device number assigned by DAQDRIVE will be stored. The application should initialize the integer pointed to by logical_device to 0.

device_type - This pointer defines the starting address of a character array (string) which describes the hardware device to be opened. The target hardware's appendix contains the valid settings for device_type.

config_file - This pointer defines the starting address of a character array (string) which defines the name of the DAQDRIVE configuration file to be used. This character string must contain the drive, path, filename, and extension of the desired configuration file.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B.  *****/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DA8P-12B operations here.  *****/

/*****  Close the DA8P-12B.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "DAQP-16";
char far *config_file = "c:\\daqp-16\\daqp-16.dat";

/*****  Open the DAQP-16.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DAQP-16 operations here.  *****/

/*****  Close the DAQP-16.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

### 13.31.2 DaqOpenDevice - Windows Version

The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs). Consult the target hardware's appendix for the required settings of the DLL_name and device_type variables.

```
unsigned short DaqOpenDevice(char far *DLL_name,
                             unsigned short far *logical_device,
                             char far *device_type,
                             char far *config_file)
```

DLL_name - This pointer defines the starting address of a character array (string) specifying the hardware dependent DLL required for the desired adapter. The name of this DLL is contained in the target hardware's appendix. If DLL_name does not specify a path, Windows will search for the DLL in the following order: the current directory, the Windows directory, the Windows system directory, the application's directory, the system's PATH, and any mapped network drives.

logical_device - This pointer specifies the address of an unsigned short integer where the logical device number assigned by DAQDRIVE will be stored. The application should initialize the integer pointed to by logical_device to 0.

device_type - This pointer defines the starting address of a character array (string) which describes the hardware device to be opened. The target hardware's appendix contains the valid settings for device_type.

config_file - This pointer defines the starting address of a character array (string) which defines the name of the DAQDRIVE configuration file to be used. This character string must contain the drive, path, filename, and extension of the desired configuration file.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "DAQP-208";
char far *config_file = "c:\\daqp-208\\daqp-208.dat";
char far *DLL_name = "c:\\daqp-208\\daqpwin.dll";

/*****  Open the DAQP-208.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DAQP-208 operations here.  *****/

/*****  Close the DAQP-208.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "IOP-241";
char far *config_file = "c:\\iop-241\\iop-241.dat";
char far *DLL_name = "iop-241.dll";

/*****  Open the IOP-241.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any IOP-241 operations here.  *****/

/*****  Close the IOP-241.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

### 13.31.3 DaqOpenDevice - TSR Version

The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident version of DAQDRIVE. Consult the target hardware's appendix for the required settings of the TSR_number and device_type variables.

```
unsigned short DaqOpenDevice(unsigned short TSR_number,
                             unsigned short far *logical_device,
                             char far *device_type,
                             char far *config_file)
```

TSR_number - This unsigned short integer variable specifies the interrupt service number for the desired adapter. TSR_number is defined in the target hardware's appendix and should not be confused with the software interrupt number where DAQDRIVE is installed.

logical_device - This pointer specifies the address of an unsigned short integer where the logical device number assigned by DAQDRIVE will be stored. The application should initialize the integer pointed to by logical_device to 0.

device_type - This pointer defines the starting address of a character array (string) which describes the hardware device to be opened. The target hardware's appendix contains the valid settings for device_type.

config_file - This pointer defines the starting address of a character array (string) which defines the name of the DAQDRIVE configuration file to be used. This character string must contain the drive, path, filename, and extension of the desired configuration file.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf005;
char far *device_type = "DAQP-16";
char far *config_file = "c:\\daqp-16\\daqp-16.dat";

/*****  Open the DAQP-16.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DAQP-16 operations here.  *****/

/*****  Close the DAQP-16.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf006;
char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform any DA8P-12B operations here.  *****/

/*****  Close the DA8P-12B.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.32 DaqPostMessageEvent (Windows Versions Only)

DaqPostMessageEvent is available only in the Windows version of DAQDRIVE. It installs a pre-defined messaging procedure using DaqNotifyEvent to post event messages to the application's window and should be executed before the request is armed. DaqNotifyEvent and DaqPostMessageEvent can not both be used on the same request.

```
unsigned short DaqPostMessageEvent(unsigned short request_handle,
                                   unsigned long event_mask,
                                   unsigned short window_handle)
```

request_handle - This unsigned short integer variable is used to define which request is to use the event procedure defined by event_procedure. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput or DaqDigitalOutput.

event_mask - This unsigned long integer value is used to specify which events the application wishes to be notified of. event_mask is defined as a bit mask - setting a specific bit to logic 1 enables notification of the corresponding event. The bit definitions of event mask are given below.

| DAQDRIVE Constant | Value | Description |
|---|---|---|
| NO_EVENTS | 0x00000000 | Disable all event notification. |
| TRIGGER_EVENT | 0x00000001 | Enable notification of trigger events. |
| COMPLETE_EVENT | 0x00000002 | Enable notification of complete events. |
| BUFFER_EMPTY_EVENT | 0x00000004 | Enable notification of buffer empty events. |
| BUFFER_FULL_EVENT | 0x00000008 | Enable notification of buffer full events. |
| SCAN_EVENT | 0x00000010 | Enable notification of scan events. |
| USER_BREAK_EVENT | 0x20000000 | Enable notification of user break events. |
| TIMEOUT_EVENT | 0x40000000 | Enable notification of time-out events. |
| RUNTIME_ERROR_EVENT | 0x80000000 | Enable notification of run-time error events. |

window_handle - This unsigned short integer value is the handle of the application program window (HWND).

### 13.32.1 The Event Message

When an event occurs, DAQDRIVE uses the Windows PostMessage procedure to send an event message to the window specified by window_handle (HWND). The message number (uMsg) is the sum of the event value specified in figure 8 and the pre-defined Windows constant WM_USER. The two message specific arguments, LPARAM and WPARAM, are used to specify the request's request_handle and an event error_code respectively. The error code is set to 0 for all events except the run-time error event where it is used to specify the type of error encountered as defined in chapter 14.

## 13.33 DaqReleaseRequest

The DaqReleaseRequest releases a previously defined request allowing the configured channels to be re-used. This is the reverse of the DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, and DaqDigitalOutput procedures. DaqReleaseRequest may be used on configurations that were never armed (DaqArmRequest), on requests that have been completed, or on requests that have otherwise been terminated.

```
unsigned short DaqReleaseRequest(unsigned short request_handle)
```

request_handle - This unsigned short integer variable is used to define which request is to be released. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput or DaqDigitalOutput.

```
#include "daqdrive.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel;
unsigned short status;
unsigned short error;
unsigned short i;
         short data_array[20];
unsigned long  event_mask;

struct DAC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the DA8P-12B(see DaqOpenDevice).  *****/
/*****  Request D/A output (See DaqAnalogOutput).  *****/
/*****  Arm the request (See DaqArmRequest).  *****/
/*****  Trigger the request.  *****/
/*****  Wait for completion or error.  *****/

event_mask = COMPLETE_EVENT | RUNTIME_ERROR_EVENT;
while((user_request.request_status & event_mask) == 0);

if ((user_request.request_status & COMPLETE_EVENT) != 0)
   printf("Request complete.\n");
else
   {
   status = GetRuntimeError(request_handle, &error);
   printf("Run-time error #%d.  Waveform aborted.\n", error);
   }

/*****  Release the request.  *****/

status = DaqReleaseRequest(request_handle);
if (status != 0)
   printf("Could not release configuration.  Status code %d.\n"),status);

/*****  Close the DA8P-12B.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.34 DaqResetDevice

DaqResetDevice returns the specified hardware device to its power-up state. In situations where more than one application program is using the target device, performing a reset could corrupt other tasks. Under these circumstances, DaqResetDevice will return an error indicating the device could not be reset in a multi-user environment.

```
unsigned short DaqResetDevice(unsigned short logical_device)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

**NOTE:**

Not all hardware devices respond to DaqResetDevice in the same manner. Consult the target hardware's appendix to determine the exact operation of this procedure.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B.  *****/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Perform DA8P-12B operations here.  *****/

/*****  Reset the DA8P-12B.  *****/

status = DaqResetDevice(logical_device);
if (status != 0)
   {
   printf("Error resetting device.  Status code %d.\n"),status);
   return(status);
   }

/*****  Perform additional DA8P-12 operations.  *****/

/*****  Close the DA8P-12B.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.35 DaqSingleAnalogInput

The DaqSingleAnalogInput procedure provides a simplified interface for inputting a single point from a single A/D converter channel. The format of the command is shown below. The analog input specified by channel_number on the adapter defined by logical_device is configured for the gain specified by gain_setting. The analog input is converted to a digital value which is returned to the address specified by input_value. This procedure executes the DAQDRIVE procedures DaqAnalogInput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleAnalogInput(unsigned short logical_device,
                                    unsigned short channel_number,
                                    float gain_setting,
                                    void far *input_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_number - This unsigned short integer value is used to specify which A/D converter channel on logical_device is to be converted.

gain_setting - This floating point value defines the gain setting for the channel specified by channel_number.

input_value - This void pointer specifies the address where the value input from the A/D converter is to be stored. input_value is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by input_value is the correct type for the target hardware as listed in the table below.

| Resolution | Configuration | data type |
|---|---|---|
| 1 to 8 bits | unipolar | unsigned char |
| | bipolar | signed char |
| 9 to 16 bits | unipolar | unsigned short |
| | bipolar | signed short |
| 17 to 32 bits | unipolar | unsigned long |
| | bipolar | signed long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short ADC_channel;
         short input_value;

         float gain_setting;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input one value from A/D channel 7 with a gain of 1.  *****/

ADC_channel  = 7;
gain_setting = 1.0;
status = DaqSingleAnalogInput(logical_device, ADC_channel,
                              gain_setting, &input_value);
if (status != 0)
   printf("Error reading from A/D.  Status code %d.\n",status);

/*****  Close the DAQ-1201.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n",status);
return(status);
}
```

## 13.36 DaqSingleAnalogInputScan

The DaqSingleAnalogInputScan procedure provides a simplified interface for inputting a single point from multiple A/D converter channels. The format of the command is shown below. The analog input channels specified by channel_array on the adapter defined by logical_device are configured for the gain settings specified by gain_array. The analog inputs are then converted to digital values which are returned to the array specified by input_array. There is a one-to-one correspondence between the number of analog input channels, the number of gain settings, and the number of samples. Therefore, array_length specifies the length of channel_array, gain_array, and input_array. This procedure executes the DAQDRIVE procedures DaqAnalogInput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleAnalogInputScan(unsigned short logical_device,
                                        unsigned short far *channel_array,
                                        float far *gain_array,
                                        unsigned short array_length,
                                        void far *input_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array - This pointer specifies the address of an unsigned short integer array containing the analog input channels on logical_device to be sampled.

gain_array - This pointer specifies the address of a floating point array defining the gain setting for the channels specified by channel_array.

array_length - This unsigned short integer value defines the length of channel_array, gain_array, and input_array.

input_array - This void pointer specifies the address of an array where the values input from the A/D converter are to be stored. input_array is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by input_array is the correct type for the target hardware as listed in the table below.

| Resolution | Configuration | data type |
|---|---|---|
| 1 to 8 bits | unipolar | unsigned char |
| | bipolar | signed char |
| 9 to 16 bits | unipolar | unsigned short |
| | bipolar | signed short |
| 17 to 32 bits | unipolar | unsigned long |
| | bipolar | signed long |

Figure 26. input_array data types as a function of analog input channel type.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short channel_array[3] = {   0,   1,   2 };
         float gain_array[3]    = { 1.0, 1.0, 8.0 };
         short input_array[3];
unsigned short array_length;

char far *device_type = "DAQP-208";
char far *config_file = "c:\\daqp-208\\daqp-208.dat";

/*****  Open the DAQP-208.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input one value from A/D channels 0, 1, and 2.  *****/

status = DaqSingleAnalogInputScan(logical_device, channel_array, gain_array,
                                  array_length, input_array);
if (status != 0)
   printf("Error reading from A/D.  Status code %d.\n",status);

/*****  Close the DAQP-208.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n",status);
return(status);
}
```

## 13.37 DaqSingleAnalogOutput

The DaqSingleAnalogOutput procedure provides a simplified interface for outputting a single point to a single D/A converter. The format of the command is shown below. The value specified by output_value is output to the D/A converter specified by channel_number on the adapter specified by logical_device. This procedure executes the DAQDRIVE procedures DaqAnalogOutput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleAnalogOutput(unsigned short logical_device,
                                     unsigned short channel_number,
                                     void far *output_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_number - This unsigned short integer value is used to specify which D/A converter channel on logical_device is to receive the output data.

output_value - This void pointer specifies the address of the data to be output to the D/A converter. output_value is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by output_value is the correct type for the target hardware as listed in the table below.

| Resolution | Configuration | data type |
|---|---|---|
| 1 to 8 bits | unipolar | unsigned char |
| | bipolar | signed char |
| 9 to 16 bits | unipolar | unsigned short |
| | bipolar | signed short |
| 17 to 32 bits | unipolar | unsigned long |
| | bipolar | signed long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short DAC_channel;
         short output_value;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Output one value to D/A channel 0.  *****/

DAC_channel  = 0;
output_value = 1024;
status = DaqSingleAnalogOutput(logical_device, DAC_channel, &output_value);
if (status != 0)
   printf("Error writing to D/A.  Status code %d.\n",status);

/*****  Close the DAQ-1201.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.38 DaqSingleAnalogOutputScan

The DaqSingleAnalogOutputScan procedure provides a simplified interface for outputting a single point to multiple D/A converter. The format of the command is shown below. The values specified by output_array are output to the D/A converters specified by channel_array on the adapter specified by logical_device. A D/A channel may appear in channel_array only once. There is a one-to-one correspondence between the number of analog output channels and the number of output values. Therefore, array_length specifies the length of both channel_array and output_array. This procedure executes the DAQDRIVE procedures DaqAnalogOutput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleAnalogOutputScan(unsigned short logical_device,
                                         unsigned short far *channel_array,
                                         unsigned short array_length,
                                         void far *output_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array - This pointer specifies the address of an unsigned short integer array containing the analog output channels on logical_device to be written.

array_length - This unsigned short integer value defines the length of channel_array and output_array.

output_array - This void pointer specifies the address of an array containing the data to be output to the analog output channels. output_array is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by output_array is the correct type for the target hardware as listed in the table below.

| Resolution | Configuration | data type |
|---|---|---|
| 1 to 8 bits | unipolar | unsigned char |
| | bipolar | signed char |
| 9 to 16 bits | unipolar | unsigned short |
| | bipolar | signed short |
| 17 to 32 bits | unipolar | unsigned long |
| | bipolar | signed long |

Figure 27. output_array data types as a function of analog output channel type.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short channel_array[2] = {    0,    1 };
         short output_array[2]  = { 1024, 2316 };
unsigned short array_length = 2;

char far *device_type = "DAQP-208";
char far *config_file = "c:\\daqp-208\\daqp-208.dat";

/*****  Open the DAQP-208.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Output values to D/A channels.  *****/

status = DaqSingleAnalogOutputScan(logical_device, channel_array,
                                   array_length, output_array);
if (status != 0)
   printf("Error writing to D/A.  Status code %d.\n",status);

/*****  Close the DAQP-208.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.39 DaqSingleDigitalInput

The DaqSingleDigitalInput procedure provides a simplified interface for inputting a single point from a single digital input channel. The format of the command is shown below. The digital input specified by channel_number on the adapter defined by logical_device is returned to the address specified by input_value. This procedure executes the DAQDRIVE procedures DaqDigitalInput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleDigitalInput(unsigned short logical_device,
                                     unsigned short channel_number,
                                     void far *input_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_number - This unsigned short integer value is used to specify which digital input channel on logical_device is to be read.

input_value - This void pointer specifies the address where the value read from the digital input channel is to be stored. input_value is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by input_value is the correct type for the target hardware as listed in the table below.

| Channel size (in bits) | data type |
|---|---|
| 1 to 8 bits | unsigned char |
| 9 to 16 bits | unsigned short |
| 17 to 32 bits | unsigned long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short digio_channel;
         short input_value;

char far *device_type = "DAQP-16";
char far *config_file = "c:\\daqp-16\\daqp-16.dat";

/*****  Open the DAQP-16.  *****/

device_number = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input one value from digital input channel 0.  *****/

digio_channel  = 0;
status = DaqSingleDigitalInput(logical_device, digio_channel, &input_value);
if (status != 0)
   printf("Error reading digital input.  Status code %d.\n",status);

/*****  Close the DAQP-16.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.40 DaqSingleDigitalInputScan

The DaqSingleDigitalInputScan procedure provides a simplified interface for inputting a single point from multiple digital input channels. The format of the command is shown below. The digital input channels specified by channel_array on the adapter defined by logical_device are returned to the array specified by input_array. There is a one-to-one correspondence between the number of digital input channels and the number of input values. Therefore, array_length specifies the length of both channel_array and input_array. This procedure executes the DAQDRIVE procedures DaqDigitalInput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleDigitalInputScan(unsigned short logical_device,
                                         unsigned short far *channel_array,
                                         unsigned short array_length,
                                         void far *input_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array - This pointer specifies the address of an unsigned short integer array containing the digital input channels on logical_device to be input.

array_length - This unsigned short integer value defines the length of channel_array and input_array.

input_array - This void pointer specifies the address of an array where the values read from the digital input channels are to be stored. input_array is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by input_array is the correct type for the target hardware as listed in the table below.

| Channel size (in bits) | data type |
|---|---|
| 1 to 8 bits | unsigned char |
| 9 to 16 bits | unsigned short |
| 17 to 32 bits | unsigned long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "iop241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short channel_array[3] = { 3, 0, 6 };
         short input_array[3];
unsigned short array_length = 3;

char far *device_type = "IOP-241";
char far *config_file = "c:\\iop-241\\iop-241.dat";

/*****  Open the IOP-241.  *****/

device_number = 0;
status = DaqOpenDevice(IOP241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input values from channels.  *****/

status = DaqSingleDigitalInputScan(logical_device, channel_array,
                                   array_length, input_array);
if (status != 0)
   printf("Error reading digital input.  Status code %d.\n",status);

/*****  Close the IOP-241.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.41 DaqSingleDigitalOutput

The DaqSingleDigitalOutput procedure provides a simplified interface for outputting a single point to a single digital output channel. The format of the command is shown below. The value specified by output_value is output to the digital output channel specified by channel_number on the adapter specified by logical_device. This procedure executes the DAQDRIVE procedures DaqDigitalOutput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleDigitalOutput(unsigned short logical_device,
                                      unsigned short channel_number,
                                      void far *output_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_number - This unsigned short integer value is used to specify which digital output channel on logical_device is to receive the output data.

output_value - This void pointer specifies the address of the data to be written to the digital output channel. output_value is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by output_value is the correct type for the target hardware as listed in the table below.

| Channel size (in bits) | data type |
|---|---|
| 1 to 8 bits | unsigned char |
| 9 to 16 bits | unsigned short |
| 17 to 32 bits | unsigned long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short digio_channel;
         short output_value;

char far *device_type = "DA8P-12B";
char far *config_file = "c:\\da8p-12b\\da8p-12b.dat";

/*****  Open the DA8P-12B.  *****/

logical_device = 0;
status = DaqOpenDevice(DA8P-12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Output one value to digital output channel 3.  *****/

digio_channel  = 3;
output_value = 1;
status = DaqSingleDigitalOutput(logical_device, digio_channel, &output_value);
if (status != 0)
   printf("Error writing to digital output.  Status code %d.\n",status);

/*****  Close the DA8P-12B.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.42 DaqSingleDigitalOutputScan

The DaqSingleDigitalOutputScan procedure provides a simplified interface for outputting a single point to multiple digital output channels. The format of the command is shown below. The values specified by the array output_array are output to the digital output channels specified by channel_array on the adapter defined by logical_device. There is a one-to-one correspondence between the number of digital output channels and the number of output values. Therefore, array_length specifies the length of both channel_array and output_array. This procedure executes the DAQDRIVE procedures DaqDigitalInput, DaqArmRequest, DaqTriggerRequest, and DaqReleaseRequest before returning to the calling application.

```
unsigned short DaqSingleDigitalOutputScan(unsigned short logical_device,
                                          unsigned short far *channel_array,
                                          unsigned short array_length,
                                          void far *output_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array - This pointer specifies the address of an unsigned short integer array containing the digital output channels on logical_device to be written.

array_length - This unsigned short integer value defines the length of channel_array and output_array.

output_array - This void pointer specifies the address of an array containing the values to be output to the channels specified by channel_array. output_array is declared as a void to allow it to point to data of any type. It is the application program's responsibility to ensure the data pointed to by output_array is the correct type for the target hardware as listed in the table below.

| Channel size (in bits) | data type |
|---|---|
| 1 to 8 bits | unsigned char |
| 9 to 16 bits | unsigned short |
| 17 to 32 bits | unsigned long |

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "iop241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short channel_array[4] = { 6, 3, 9, 13 };
         short output_array[4]  = { 0, 3, 1,  7 };
unsigned short array_length = 4;

char far *device_type = "IOP-241";
char far *config_file = "c:\\iop-241\\iop-241.dat";

/*****  Open the IOP-241.  *****/

device_number = 0;
status = DaqOpenDevice(IOP241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Output one value to digital output channel 3.  *****/

status = DaqSingleDigitalOutputScan(logical_device, channel_array,
                                    array_length, output_array);
if (status != 0)
   printf("Error writing to digital output.  Status code %d.\n",status);

/*****  Close the IOP-241.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.43 DaqSingleSigConInput

The DaqSingleSigConInput procedure provides a simplified interface for inputting a single sample from a single A/D converter channel and returning the result in "real world" engineering units. The analog input specified by channel_number on the adapter defined by logical_device is configured for the gain specified by gain_setting. A single sample is read from the analog input channel, converted to engineering units using the hardware configuration information stored within DAQDRIVE, and stored in the variable specified by input_value.

```
unsigned short DaqSingleSigConInput(unsigned short logical_device,
                                    unsigned short channel_number,
                                    float gain_setting,
                                    double far *input_value)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_number - This unsigned short integer value is used to specify which A/D converter channel on logical_device is to be converted.

gain_setting - This floating point value defines the gain setting for the channel specified by channel_number.

input_value - This pointer specifies the address of a double-precision floating point variable where the value input from the A/D converter is to be stored.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short ADC_channel;
         short input_value;

         float gain_setting;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input one value from A/D channel 7 with a gain of 1.  *****/

ADC_channel  = 7;
gain_setting = 1.0;
status = DaqSingleAnalogInput(logical_device, ADC_channel,
                              gain_setting, &input_value);
if (status != 0)
   printf("Error reading from A/D.  Status code %d.\n",status);

/*****  Close the DAQ-1201.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n",status);
return(status);
}
```

## 13.44 DaqSingleSigConInputScan

The DaqSingleSigConInputScan procedure provides a simplified interface for inputting a single point from multiple A/D converter channels and returning the results in "real world" engineering units. The analog input channels specified by channel_array on the adapter defined by logical_device are configured for the gain settings specified by gain_array. A single sample is read from each analog input channel, converted to engineering units using the hardware configuration information stored within DAQDRIVE, and stored in the array specified by input_value. There is a one-to-one correspondence between the number of analog input channels, the number of gain settings, and the number of samples. Therefore, array_length specifies the length of channel_array, gain_array, and input_array.

```
unsigned short DaqSingleSigConInputScan(unsigned short logical_device,
                                        unsigned short far *channel_array,
                                        float far *gain_array,
                                        unsigned short array_length,
                                        double far *input_array)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

channel_array - This pointer specifies the address of an unsigned short integer array containing the analog input channels on logical_device to be sampled.

gain_array - This pointer specifies the address of a floating point array defining the gain setting for the channels specified by channel_array.

array_length - This unsigned short integer value defines the length of channel_array, gain_array, and input_array.

input_array - This pointer specifies the address of a double-precision floating point array where the values input from the A/D converter are to be stored.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short channel_array[3] = {   0,   1,   2 };
         float gain_array[3]    = { 1.0, 1.0, 8.0 };
         short input_array[3];
unsigned short array_length;

char far *device_type = "DAQP-208";
char far *config_file = "c:\\daqp-208\\daqp-208.dat";

/*****  Open the DAQP-208.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Input one value from A/D channels 0, 1, and 2.  *****/

status = DaqSingleAnalogInputScan(logical_device, channel_array, gain_array,
                                  array_length, input_array);
if (status != 0)
   printf("Error reading from A/D.  Status code %d.\n",status);

/*****  Close the DAQP-208.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n",status);
return(status);
}
```

## 13.45 DaqStopRequest

The DaqStopRequest halts a request that is currently armed and / or triggered (see DaqArmRequest and DaqTriggerRequest). When DaqStopRequest is complete, the request is in the same state it was in after the configuration procedure (DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput) procedures.

```
unsigned short DaqStopRequest(unsigned short request_handle)
```

request_handle - This unsigned short integer variable is used to define which request is to be halted. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Input 1000 points from the A/D in background mode using interrupts.   */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;
         short data_array[1000];

struct ADC_request     user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the DAQP-16 (See DaqOpenDevice).  *****/

/*****  Prepare data structure for analog input.  *****/

/*****  Prepare the A/D request structure.  *****/

/*****  Request A/D input (See DaqAnalogInput).  *****/

/*****  Arm the request.  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   printf("Arm request error.  Status code %d.\n",status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger request error.  Status code %d.\n",status);
   DaqStopRequest(request_handle);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Abort the request.  *****/

status = DaqStopRequest(request_handle);
if (status != 0)
   printf("Request failed to stop.  Status code %d.\n",status);

/*****  Release the request.  *****/

status = DaqReleaseRequest(request_handle);
if (status != 0)
   printf("Could not release configuration.  Status code %d.\n"),status);

/*****  Close the DAQP-16.  *****/

status = DaqCloseDevice(logical_device);
if (status != 0)
   printf("Error closing device.  Status code %d.\n"),status);
return(status);
}
```

## 13.46 DaqTriggerRequest

When an operation has been configured for an internal trigger, DaqTriggerRequest is executed after the DaqArmRequest function to start the operation.

An error is returned to the application if DaqTriggerRequest is executed for an operation not configured for internal trigger. This error is non-fatal and program execution may continue.

```
unsigned short DaqTriggerRequest(unsigned short request_handle)
```

request_handle - This unsigned short integer variable is used to define which request is to be triggered. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Output 5 points to 5 digital output channels.                        */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short status;
         short data_array[5];

struct DIGOUT_request  user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare data structure for digital output.  *****/

/*****  Prepare the digital I/O request structure.  *****/

/*****  Request digital output.  *****/

request_handle = 0;
status = DaqDigitalOutput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("Digital I/O request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Arm the request.  *****/

status = DaqArmRequest(request_handle);
if (status != 0)
   {
   printf("Arm request error.  Status code %d.\n",status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }

/*****  Trigger the request.  *****/

status = DaqTriggerRequest(request_handle);
if (status != 0)
   {
   printf("Trigger error.  Status code %d.\n",status);
   DaqReleaseRequest(request_handle);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

## 13.47 DaqUserBreak

DaqUserBreak allows the application program to install a procedure (written by the application programmer) that DAQDRIVE will execute periodically during foreground mode operations. If the application wants to terminate the operation, the user-break procedure need only return a non-zero value. To continue the operation, the user-break procedure must return zero.

```
unsigned short DaqUserBreak(unsigned short request_handle,
                            unsigned short (far pascal *break_procedure)())
```

request_handle - This unsigned short integer variable is used to define which request is to use the user-break procedure defined by break_procedure. This is the value returned to the application by the configuration procedures DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput or DaqDigitalOutput.

break_procedure - This pointer defines the starting address of the procedure to be executed during foreground mode operations. break_procedure must be a 'far' pascal compatible procedure of type unsigned short that has no input parameters. A sample C declaration of this procedure is shown below.

```
unsigned short far pascal break_procedure()
```

```
#include "daqdrive.h"
#include "userdata.h"

/*****  Define a global counter variable  *****/

global_counter = 0;

/*****  Define a user-break procedure  *****/

unsigned short far pascal my_break_procedure()
{
global_counter++
if (global_counter < 10000)
   return(0);                    /* less than 10,000 --> keep going      */
else
   return(1);                    /* more than 10,000 --> abort operation */
}

/*****  Define the main procedure  *****/

void main()
{
unsigned short request_handle;
unsigned short status;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Request an operation. (gets a request_handle)  *****/

/*****  Install user-break procedure.  *****/

status = DaqUserBreak(request_handle, my_break_procedure);
if (status != 0)
   printf("Error installing user-break.\n");

/*****  Arm the request (See DaqArmRequest).  *****/

/*****  Trigger the request (See DaqTriggerRequest).  *****/
```

## 13.48 DaqVersionNumber

DaqVersionNumber returns the version numbers of the software drivers.

```
unsigned short DaqVersionNumber(unsigned short logical_device,
                                float far *DAQDRIVE_version,
                                float far *software_version,
                                float far *firmware_version)
```

logical_device - This unsigned short integer value is used to define the target hardware device. This is the value returned to the application by the DaqOpenDevice command.

DAQDRIVE_version - This pointer defines a floating point variable where the version of DAQDRIVE will be stored.

software_version - This pointer defines a floating point variable where the version of the hardware specific software driver will be stored.

firmware_version - This pointer defines a floating point variable where the version of the adapter's firmware will be stored. Adapters which have no firmware will set this value to 0.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

float DAQDRIVE_version;
float software_version;
float firmware_version;

char far *device_type = "DAQ-1201";
char far *config_file = "c:\\daq-1201\\daq-1201.dat";

/*****  Open the DAQ-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }

/*****  Get version numbers.  *****/

status = DaqVersionNumber(device_number, &DAQDRIVE_version,
                          &software_version, &firmware_version);
if (status != 0)
   printf("Error reading version numbers.  Status code %d.\n",status);

/*****  Display version information.  *****/

printf("DAQDRIVE version:          %5.2f\n", DAQDRIVE_version);
printf("DAQ-1201 driver version:   %5.2f\n", software_version);
printf("DAQ-1201 firmware version: %5.2f\n", firmware_version);
}
```

## 13.49 DaqWordsToBytes

DaqWordsToBytes performs the reverse of the DaqBytesToWords function, converting an unsigned short integer array of 16-bit "un-packed" values into an unsigned short integer array of 8-bit "packed" values. These functions are provided especially for languages that do not support 8-bit variable types.

DaqWordsToBytes reads the "un-packed" unsigned short integer values in array word_array, converts these values to their "packed" 8-bit values, and stores the results in array byte_array. For an array of four values, the packed and un-packed arrays appear as follows:

| | integer | | integer | | integer | | integer | |
|---|---|---|---|---|---|---|---|---|
| "un-packed" array | 14 | 0 | 2E | 0 | 6 | 0 | F7 | 0 |
| "packed" array | 14 | 2E | 6 | F7 | | | | |
| | byte | byte | byte | byte | | | | |

```
void DaqWordsToBytes(unsigned short far *word_array,
                     unsigned short far *byte_array,
                     unsigned long array_length)
```

word_array - This is a pointer to an unsigned short integer array where the "un-packed" values will be stored. word_array must be at least array_length short integers in length and may specify the same array as byte_array.

byte_array - This is a pointer to an unsigned short integer array containing the "packed" values to be converted. byte_array must be at least 'array_length ÷ 2' short integers (array_length bytes) in length and may specify the same array as word_array.

array_length - This is an unsigned long integer value defining the number of data points to be converted. word_array must be at least array_length short integers in length while byte_array must be at least 'array_length ÷ 2' short integers (array_length bytes) in length.

```
#include "daqdrive.h"
#include "userdata.h"

/*************************************************************************/
/*  Output 16 points to a digital output channel.                        */
/*************************************************************************/

unsigned short main()
{
unsigned short logical_device;
unsigned short request_handle;
unsigned short channel_num;
unsigned short status;
unsigned short data_array[16];
unsigned short array_index;

struct digio_request   user_request;
struct DAQDRIVE_buffer data_structure;

/*****  Open the device (see DaqOpenDevice).  *****/

/*****  Prepare output data.  *****/

for (array_index = 0; array_index < 16; array_index++)
   data_array[array_index] = array_index;

/*****  Pack data for output.  *****/

DaqWordsToBytes(data_array, data_array, 16);

/*****  Prepare data structure for digital output.  *****/

/*************************************************************************/
/* data is in data_array     data_array is 16 points long                */
/* output buffer 1 time      next_structure = NULL  (no more structures) */
/*************************************************************************/

data_structure.data_buffer    = data_array;
data_structure.buffer_length  = 16;
data_structure.buffer_cycles  = 1;
data_structure.next_structure = NULL;

/*****  Prepare the digital output request structure.  *****/

/*****  Request digital output.  *****/

request_handle = 0;
status = DaqDigitalOutput(logical_device, &user_request, &request_handle);
if (status != 0)
   {
   printf("Digital output request error.  Status code %d.\n",status);
   DaqCloseDevice(logical_device);
   exit(status);
   }
```

# 14 Error Messages

00 No Errors.
The procedure completed without error.

10 Error opening configuration file.
The DaqOpenDevice procedure could not open the configuration file specified by config_file. Verify the drive, path, and file name are correct.

11 File is not a valid DAQDRIVE configuration file.
The file specified as the hardware configuration file is not a valid DAQDRIVE configuration file. Select a different configuration file.

12 Configuration file invalid for specified adapter type.
The adapter specified by the device_type variable does not match the type of hardware defined by the configuration file. Select a different configuration file.

13 Error reading configuration file.
An error occurred while reading the adapter configuration file. If there are no problems with the disk drive, generate a new configuration file using the DAQDRIVE configuration utility.

14 End-Of-File encountered reading configuration file.
The end of the configuration file was reached unexpectedly. If there are no problems with the disk drive, generate a new configuration file using the DAQDRIVE configuration utility.

15 Invalid configuration file version.
The configuration file specified in the DaqOpenDevice procedure is too old for this version of DAQDRIVE. Create a new configuration file using the DAQDRIVE configuration utility.

30 Error loading DLL.
An error occurred while loading the hardware dependent dynamic link library (DLL). Verify the drive, path, and DLL file name are correct

31 Cannot locate the DAQDRIVE DLL open command.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

35 Cannot locate the DAQDRIVE TSR driver.
This error occurs when the hardware specific TSR is loaded before the DAQDRIVE TSR. When using the TSR drivers, the DAQDRIVE TSR must be loaded before any hardware TSRs.

39 DAQDRIVE is out of date.
The hardware specific driver in use requires a newer version of DAQDRIVE to operate. If you did not receive the latest version of DAQDRIVE, contact Omega's technical support department for assistance.

50 Auto-configuration support not available.
The required PCMCIA Card and Socket Services or Plug-and- Play support software is not installed on the system. The user must specify the hardware configuration in the configuration file or the required software drivers must be installed on the system.

51 Invalid device type.
An invalid device type was specified in the configuration file. If there are no apparent problems reading the file, generate a new configuration file using the DAQDRIVE configuration utility.

60 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

70 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

71 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

72 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

73 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

74 Configuration file error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

100 Invalid logical device number.
An adapter could not be found with the specified logical device number. Make sure the DaqOpenDevice procedure executed successfully and that the logical device number matches the value returned by the DaqOpenDevice procedure.

120 No logical devices defined.
There are no adapters currently "opened". Make sure the DaqOpenDevice procedure is executed without error before any other procedures are called.

150 Invalid request handle.
A request could not be found with the specified request handle. Make sure the configuration procedure (DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput) executed successfully and that the request handle matches the value returned by the configuration procedure.

200 No interrupt level defined for adapter.
The requested operation requires a hardware interrupt (IRQ) and no interrupt level was defined for the adapter in the hardware configuration file.

201 Interrupt in-use by another device.
The requested operation requires a hardware interrupt (IRQ) that is currently in use by another device. This operation must be requested again after the other device has relinquished control of the interrupt.

205 Internal interrupt error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

250 No DMA channel defined for adapter.
The requested operation requires one or more DMA channels and no DMA channels were defined for the adapter in the hardware configuration file.

251 DMA channel in-use by another device.
The requested operation requires one or more DMA channels that are currently in use by other device(s). This operation must be requested again after the other device(s) have relinquished control of the DMA channels.

255 Internal DMA error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

300 Memory allocation error.
An error occurred while DAQDRIVE was attempting to allocate memory for internal use. Generally this error only occurs when there is no more memory available in the system. Remove any unnecessary device drivers and memory resident programs and execute the application again.

310 Memory release error.
An error occurred while DAQDRIVE was attempting to release memory previously allocated for internal use. If this error occurs, the system has become unstable.

400 Channel in-use by another request.
One or more channels specified by this request are currently in use by other request(s). This request must wait until the other request(s) are complete and have made their channels available.

410 Timer in-use by another request.
One or more timer channels required for the requested operation are currently in use by other request(s). This request must wait until the other request(s) are complete and have made their timer channels available.

450 Hardware dependent resource in-use by another device.
A hardware specific resource required for the requested operation is currently in use by another request. This request must wait until the other request is complete and relinquishes control of the resource. Consult the target hardware's appendix to determine the cause of this error.

500 Invalid procedure call for a request that is not configured.
The procedure cannot be executed because the request has not been configured. Make sure the configuration procedure (DaqAnalogInput, DaqAnalogOutput, DaqDigitalInput, or DaqDigitalOutput) executed successfully.

600 Invalid procedure call for a request that is not armed.
The procedure cannot be executed because the request has not been armed. Make sure the DaqArmRequest procedure executed successfully.

650 Invalid procedure call for a request that is armed.
The procedure cannot be executed because the request has been armed. The request may be removed from the arm state by executing the DaqStopRequest procedure.

700 Trigger command invalid with specified trigger source.
The DaqTriggerRequest procedure was executed on a request that was not configured for a software trigger. This is not a critical error and the application program may continue.

800 Invalid re-configuration request.
The re-configuration request can not be processed because the channel list was modified. All parameters except the channel list may be modified by a re-configuration request. To modify the channel list, the request must be released (DaqReleaseRequest) and a new configuration must be requested.

1000 Requested function not supported by target hardware.
The requested operation can not be performed on the target hardware. Consult the target hardware's appendix to determine which parameter(s) are not supported by the adapter.

1050 Invalid operation in multi-user mode.
The procedure could not be executed because more than one application is currently operating on the adapter. This error is generally reported by procedures that effect the state of the hardware (e.g. DaqResetDevice).

1100 Invalid channel number.
One or more values in the request's channel list is out of range.

1101 Invalid array length.
The specified array length is 0 or larger than the maximum allowable array size.

1150 Duplicate entries in channel list.
A logical channel number appears in the channel list more than once. Each channel may appear in the channel list only once for the type of operation requested.

1160 Invalid channel sequence.
The sequence of channels specified in the channel list is not supported by the hardware. Consult the hardware specific appendices for restrictions on channel lists / sequences.

1280 Invalid gain.
The adapter can not be configured for the gain requested. Consult the hardware specific appendices for valid gain selections.

1300 Invalid data buffer length.
The data buffer must be defined to hold an integer number of scans of the channel list. For example, if the channel list contains three channels, the data buffer must be defined to hold 3, 6, 9, ... samples.

1320 Invalid output value.
One or more values specified for output is not in the valid range for the corresponding channel(s).

1350 DMA mode data buffer crosses page boundary.
One or more data buffers allocated for the DMA mode operation span a physical page boundary. Data buffers must be contained in a single memory page for DMA use.

1351 DMA mode data buffer defined on odd address.
One or more data buffers allocated for the DMA mode operation are aligned on an odd address. When using 16-bit DMA transfers, all data buffers must reside on even addresses (word aligned).

1352 Internal DMA error.
This is an internal DAQDRIVE error. If this error is received, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

1400 Invalid trigger source.
The trigger source specified is not one of DAQDRIVE's trigger source selections.

1401 Trigger source not supported.
The trigger source specified is not supported by the adapter for the requested operation. Consult the hardware specific appendices for supported trigger sources.

1410 Invalid trigger slope.
The trigger slope specified is not one of DAQDRIVE's trigger slope selections.

1411 Trigger slope not supported.
The trigger slope specified is not supported by the adapter for the requested operation. Consult the hardware specific appendices for supported trigger slopes.

1420 Invalid trigger channel.
The trigger channel specified is not supported by the adapter for the requested operation. Consult the hardware specific appendices for supported trigger channels.

1430 Invalid analog trigger voltage.
The analog trigger voltage is not in the valid range for the adapter. Consult the hardware specific appendices for the valid analog trigger voltage ranges.

1500 Invalid data transfer mode.
The data transfer mode specified is not one of DAQDRIVE's data transfer mode selections.

1600 Invalid clock source.
The clock source specified is not one of DAQDRIVE's clock source selections.

1601 Clock source not supported.
The clock source specified is not supported by the adapter for the requested operation. Consult the hardware specific appendices for supported clock sources.

1650 Invalid sampling rate.
The sampling rate is not in the valid range for the adapter. Consult the hardware specific appendices for the valid sampling rate ranges.

1700 Invalid calibration mode.
The calibration mode specified is not one of DAQDRIVE's calibration selections.

1710 Adapter does not support auto-calibration.
Auto-calibration is not supported on the channel or device specified. Consult the hardware specific appendices for supported calibration modes.

1720 Adapter does not support auto-zero.
Auto-zero is not supported on the channel or device specified. Consult the hardware specific appendices for supported calibration modes.

3000 Hardware failure.
An unidentified hardware failure has occurred. Check all hardware connections, switch settings, and jumper settings. If no problems are detected, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

3010 A/D converter failure.
A hardware failure has occurred in the A/D sub-system of the adapter. Check all hardware connections, switch settings, and jumper settings. If no problems are detected, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

3020 D/A converter failure.
A hardware failure has occurred in the D/A sub-system of the adapter. Check all hardware connections, switch settings, and jumper settings. If no problems are detected, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

3030 Digital I/O failure.
A hardware failure has occurred in the digital I/O sub-system of the adapter. Check all hardware connections, switch settings, and jumper settings. If no problems are detected, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

3040 Counter/timer failure.
A hardware failure has occurred in the counter/timer sub-system of the adapter. Check all hardware connections, switch settings, and jumper settings. If no problems are detected, contact Omega's technical support department. If possible, have the hardware device type and software version numbers available when calling.

5000 Buffer over-run.
During the input operation, a sample was input from the adapter that could not be stored in the data buffer(s) because the data buffer(s) were full. The input operation was terminated.

5010 Buffer under-run.
During the output operation, a sample could not be output to the adapter because the data buffer(s) were empty. The output operation was terminated.

5100 FIFO over-run.
During the input operation, a sample was input that could not be transferred into the adapter's data FIFO because the FIFO was full. The input operation was terminated.

5110 FIFO under-run.
During the output operation, a sample could not be retrieved from the adapter's data FIFO because the FIFO was empty. The output operation was terminated.

5200 Request time-out.
The requested operation was terminated because the user specified time-out interval was exceeded while waiting to process the request.

5300 User break.
The requested operation was terminated because a non-zero value was returned from the user break procedure.

# Appendix A: PXB-241

## A.1 Distribution Software

### A.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more PXB-241s, the application must be linked with the appropriate DAQDRIVE library and one of the following PXB-241 libraries:

For Microsoft Visual C/C++

- PXB241MS.LIB - small model PXB-241 library
- PXB241MM.LIB - medium model PXB-241 library
- PXB241MC.LIB - compact model PXB-241 library
- PXB241ML.LIB - large model PXB-241 library

For Borland C/C++

- PXB241BS.LIB - small model PXB-241 library
- PXB241BM.LIB - medium model PXB-241 library
- PXB241BC.LIB - compact model PXB-241 library
- PXB241BL.LIB - large model PXB-241 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file PXB241.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the PXB-241 driver.

### A.1.2 Creating DOS Applications Using The TSR Drivers

Before running a PXB-241 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the PXB-241 TSR with the command line:

```
PXB-241
```

This file, PXB-241.EXE, is located in the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the PXB-241 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the PXB-241 driver will not be installed.

### A.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more PXB-241s is executed, it must be able to dynamically link to the DAQDRIVE and PXB-241 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and PXB-241 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## A.2 Configuring The PXB-241

Before DAQDRIVE can operate the PXB-241, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### A.2.1 General Configuration

The PXB-241's base address must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 3f8H with 8 interval. The base address value should reflect the DIP switch setting of SW1 (refer to the PXB-241 Hardware Manual).

### A.2.2 Digital I/O Configuration

The PXB-241 has 24 bits of digital I/O. The 24 bits are Port A, Port B, and Port C which are 8255A mode 0 equivalent. The 24 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group type. The group type are Port A, Port B, Port C bit 0 to 3, and Port C bit 4 to 7. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 23.

## A.3 Opening The PXB-241

### A.3.1 Using the PXB-241 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a PXB-241, the application program must include PXB241.H. In addition, the constant PROCEDURE must be replaced by the PXB241(exactly and without quotes) and the device_type variable must be defined as "PXB-241" for a PXB-241 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "pxb241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PXB-241";
char *config_file = "PXB-241.dat";

/*****  Open the PXB-241.  *****/

logical_device = 0;
status = DaqOpenDevice(PXB241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### A.3.2 Using the PXB-241 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a PXB-241, the TSR_number variable must be set to the value F008 hexadecimal (61, 448 decimal) and the device_type variable must be defined as "PXB-241" for a PXB-241 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf008;
char *device_type = "PXB-241";
char *config_file = "PXB-241.dat";

/*****  Open the pxb-241.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### A.3.3 Using the PXB-241 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a PXB-241, the DLL_name variable must specify the PXB-241 dynamic link library (PXB241.DLL) and the device_type variable must be defined as "PXB-241" for a PXB-241 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PXB-241";
char *config_file = "PXB-241.dat";
char *DLL_name    = "PXB-241.dll";

/*****  Open the PXB-241.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## A.4 Digital Input

The PXB-241 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## A.5 Digital Output

The PXB-241 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix B: PXB-721

## B.1 Distribution Software

### B.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more PXB-721s, the application must be linked with the appropriate DAQDRIVE library and one of the following PXB-721 libraries:

For Microsoft Visual C/C++

- PXB721MS.LIB - small model PXB-721 library
- PXB721MM.LIB - medium model PXB-721 library
- PXB721MC.LIB - compact model PXB-721 library
- PXB721ML.LIB - large model PXB-721 library

For Borland C/C++

- PXB721BS.LIB - small model PXB-721 library
- PXB721BM.LIB - medium model PXB-721 library
- PXB721BC.LIB - compact model PXB-721 library
- PXB721BL.LIB - large model PXB-721 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file PXB721.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the PXB-721 driver.

### B.1.2 Creating DOS Applications Using The TSR Drivers

Before running a PXB-721 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the PXB-721 TSR with the command line:

PXB-721

This file, PXB-721.EXE, is located in the \TSR directory of the PXB-721 distribution diskette.

When the PXB-721 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the PXB-721 driver will not be installed.

### B.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more PXB-721s is executed, it must be able to dynamically link to the DAQDRIVE and PXB-721 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and PXB-721 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## B.2 Configuring The PXB-721

Before DAQDRIVE can operate the PXB-721, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### B.2.1 General Configuration

The PXB-721's base address must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 3f0H with 10H interval. The base address value should reflect the DIP switch setting of SW1 (refer to the PXB-721 Hardware Manual).

### B.2.2 Digital I/O Configuration

The PXB-721 has 72 bits of digital I/O which derived from three 8255A chips. Each 8255A has 24 bits (Port A, Port B, and Port C, which are 8255A mode 0 equivalent). The 72 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group type. The group type are Port A, Port B, Port C bit 0 to 3, and Port C bit 4 to 7. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 71.

## B.3 Opening The PXB-721

### B.3.1 Using the PXB-721 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a PXB-721, the application program must include PXB721.H. In addition, the constant PROCEDURE must be replaced by the PXB721(exactly and without quotes) and the device_type variable must be defined as "PXB-721" for a PXB-721 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "pxb721.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PXB-721";
char *config_file = "PXB-721.dat";

/*****  Open the PXB-721.  *****/

logical_device = 0;
status = DaqOpenDevice(PXB721, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### B.3.2 Using the PXB-721 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a PXB-721, the TSR_number variable must be set to the value F009 hexadecimal (61, 449 decimal) and the device_type variable must be defined as "PXB-721" for a PXB-721 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf009;
char *device_type = "PXB-721";
char *config_file = "PXB-721.dat";

/*****  Open the PXB-721.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### B.3.3 Using the PXB-721 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a PXB-721, the DLL_name variable must specify the PXB-721 dynamic link library (PXB721.DLL) and the device_type variable must be defined as "PXB-721" for a PXB-721 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PXB-721";
char *config_file = "PXB-721.dat";
char *DLL_name    = "PXB-721.dll";

/*****  Open the PXB-721.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## B.4 Digital Input

The PXB-721 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## B.5 Digital Output

The PXB-721 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix C: PIO-241

## C.1 Distribution Software

### C.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more PIO-241s, the application must be linked with the appropriate DAQDRIVE library and one of the following PIO-241 libraries:

For Microsoft Visual C/C++

- PIO241MS.LIB - small model PIO-241 library
- PIO241MM.LIB - medium model PIO-241 library
- PIO241MC.LIB - compact model PIO-241 library
- PIO241ML.LIB - large model PIO-241 library

For Borland C/C++

- PIO241BS.LIB - small model PIO-241 library
- PIO241BM.LIB - medium model PIO-241 library
- PIO241BC.LIB - compact model PIO-241 library
- PIO241BL.LIB - large model PIO-241 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file PIO241.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the PIO-241 driver.

### C.1.2 Creating DOS Applications Using The TSR Drivers

Before running a PIO-241 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the PIO-241 TSR with the command line:

PIO-241

This file, PIO-241.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the PIO-241 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the PIO-241 driver will not be installed.

### C.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more PIO-241s is executed, it must be able to dynamically link to the DAQDRIVE and PIO-241 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and PIO-241 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## C.2 Configuring The PIO-241

Before DAQDRIVE can operate the PIO-241, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### C.2.1 General Configuration

The PIO-241's base address must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 3f0H with 10H interval. The base address value should reflect the DIP switches setting of SW1 and SW2 (refer to the PIO-241 Hardware Manual).

### C.2.2 Digital I/O Configuration

The PIO-241 has 24 bits of digital I/O. The 24 bits are grouped into three 8-bit ports. Each bit may be programmed as either input or output.. The 24 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group with the same input or output type. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 23.

## C.3 Opening The PIO-241

### C.3.1 Using the PIO-241 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a PIO-241, the application program must include PIO241.H. In addition, the constant PROCEDURE must be replaced by the PIO241(exactly and without quotes) and the device_type variable must be defined as "PIO-241" for a PIO-241 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "pio241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PIO-241";
char *config_file = "PIO-241.dat";

/*****  Open the PIO-241.  *****/

logical_device = 0;
status = DaqOpenDevice(PIO241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### C.3.2 Using the PIO-241 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a PIO-241, the TSR_number variable must be set to the value F00A hexadecimal (61, 450 decimal) and the device_type variable must be defined as "PIO-241" for a PIO-241 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf00a;
char *device_type = "PIO-241";
char *config_file = "PIO-241.dat";

/*****  Open the PIO-241.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### C.3.3 Using the PIO-241 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a PIO-241, the DLL_name variable must specify the PIO-241 dynamic link library (PIO241.DLL) and the device_type variable must be defined as "PIO-241" for a PIO-241 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "PIO-241";
char *config_file = "PIO-241.dat";
char *DLL_name    = "PIO241.dll";

/*****  Open the PIO-241.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## C.4 Digital Input

The PIO-241 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## C.5 Digital Output

The PIO-241 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix D: IOP-241

## D.1 Distribution Software

### D.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more IOP-241s, the application must be linked with the appropriate DAQDRIVE library and one of the following IOP-241 libraries:

For Microsoft Visual C/C++

- IOP241MS.LIB - small model IOP-241 library
- IOP241MM.LIB - medium model IOP-241 library
- IOP241MC.LIB - compact model IOP-241 library
- IOP241ML.LIB - large model IOP-241 library

For Borland C/C++

- IOP241BS.LIB - small model IOP-241 library
- IOP241BM.LIB - medium model IOP-241 library
- IOP241BC.LIB - compact model IOP-241 library
- IOP241BL.LIB - large model IOP-241 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file IOP241.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the IOP-241 driver.

### D.1.2 Creating DOS Applications Using The TSR Drivers

Before running an IOP-241 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the IOP-241 TSR with the command line:

IOP-241

This file, IOP-241.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the IOP-241 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the IOP-241 driver will not be installed.

### D.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more IOP-241s is executed, it must be able to dynamically link to the DAQDRIVE and IOP-241 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and IOP-241 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## D.2 Configuring The IOP-241

Before DAQDRIVE can operate the IOP-241, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### D.2.1 General Configuration

The IOP-241's base address must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 3f8H with 8 interval. The defined card base address should be set using the IOP-241 Enabler or Client Driver (refer to the IOP-241 Hardware Manual).

### D.2.2 Digital I/O Configuration

The IOP-241 has 24 bits of digital I/O. The 24 bits are grouped into three 8-bit ports. Each bit may be programmed as either input or output.. The 24 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group with the same input or output type. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 23.

## D.3 Opening The IOP-241

### D.3.1 Using the IOP-241 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a IOP-241, the application program must include IOP241.H. In addition, the constant PROCEDURE must be replaced by the IOP241(exactly and without quotes) and the device_type variable must be defined as "IOP-241" for a IOP-241 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "iop241.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "IOP-241";
char *config_file = "IOP-241.dat";

/*****  Open the IOP-241.  *****/

logical_device = 0;
status = DaqOpenDevice(IOP241, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### D.3.2 Using the IOP-241 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a IOP-241, the TSR_number variable must be set to the value F00B hexadecimal (61, 451 decimal) and the device_type variable must be defined as "IOP-241" for a IOP-241 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf00b;
char *device_type = "IOP-241";
char *config_file = "IOP-241.dat";

/*****  Open the IOP-241.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### D.3.3 Using the IOP-241 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a IOP-241, the DLL_name variable must specify the IOP-241 dynamic link library (IOP241.DLL) and the device_type variable must be defined as "IOP-241" for a IOP-241 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "IOP-241";
char *config_file = "IOP-241.dat";
char *DLL_name    = "IOP-241.dll";

/*****  Open the IOP-241.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## D.4 Digital Input

The IOP-241 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## D.5 Digital Output

The IOP-241 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix E: DAQ-12

## E.1 Distribution Software

### E.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more DAQ-12s, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQ-12 libraries:

For Microsoft Visual C/C++

- DAQ12MS.LIB - small model DAQ-12 library
- DAQ12MM.LIB - medium model DAQ-12 library
- DAQ12MC.LIB - compact model DAQ-12 library
- DAQ12ML.LIB - large model DAQ-12 library

For Borland C/C++

- DAQ12BS.LIB - small model DAQ-12 library
- DAQ12BM.LIB - medium model DAQ-12 library
- DAQ12BC.LIB - compact model DAQ-12 library
- DAQ12BL.LIB - large model DAQ-12 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQ12.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQ-12 driver.

### E.1.2 Creating DOS Applications Using The TSR Drivers

Before running a DAQ-12 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the DAQ-12 TSR with the command line:

```
DAQ12TSR
```

This file, DAQ12TSR.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DAQ-12 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQ-12 driver will not be installed.

### E.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQ-12s is executed, it must be able to dynamically link to the DAQDRIVE and DAQ-12 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQ-12 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## E.2 Configuring The DAQ-12

Before DAQDRIVE can operate the DAQ-12, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### E.2.1 General Configuration

The DAQ-12's base address, interrupt level and DMA channels must be defined in the general configuration window of the configuration utility. These selections must reflect the configuration of switches SW1 and SW2 and jumpers J8, J9, J10, and J11 as defined in the DAQ-12 Hardware Reference Manual.

### E.2.2 A/D Converter Configuration

The DAQ-12's A/D converter must be configured for single-ended or differential input, bipolar or unipolar operation, and pre-scaler enabled or disabled. These selections must reflect the configuration of jumpers J1, J6, and J7 as defined in the DAQ-12 Hardware Reference Manual.

### E.2.3 D/A Converter Configuration

The DAQ-12's D/A converter parameters are device type (bipolar or unipolar), reference source (internal or external ), reference voltage, and gain. These selections must reflect the configuration of jumpers J4 and J5 as defined in the DAQ-12 Hardware Reference Manual.

### E.2.4 Digital I/O Configuration

The DAQ-12 contains 8 bits of digital I/O. The first 4 bits are fixed output and last 4 bits are fixed input. The logical channel assignments begin with digital I/O bit 0 and continue through to digital I/O bit 7.

### E.2.5 Timer Configuration

The DAQ-12 does not have any user-definable timer parameters.

## E.3 Opening The DAQ-12

### E.3.1 Using the DAQ-12 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQ-12, the application program must include DAQ12.H. In addition, the constant PROCEDURE must be replaced by DAQ12 (exactly and without quotes) and the device_type variable must be defined as "DAQ-12".

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "daq-12";
char *config_file = "daq-12.dat";

/*****  Open the daq-12.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### E.3.2 Using the DAQ-12 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQ-12, the TSR_number variable must be set to the value F001 hexadecimal (61, 441 decimal) and the device_type variable must be defined as "DAQ-12" for a DAQ-12 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf001;
char *device_type = "daq-12";
char *config_file = "daq-12.dat";

/*****  Open the daq-12.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### E.3.3 Using the DAQ-12 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char far *DLL_name,
                             unsigned short far *logical_device,
                             char far *device_type,
                             char far *config_file)
```

In order to open a DAQ-12, the DLL_name variable must specify the DAQ-12 dynamic link library (DAQ12.DLL) and the device_type variable must be defined as "DAQ-12".

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DAQ-12";
char *config_file = "DAQ-12.dat";
char *DLL_name    = "DAQ12.dll";

/*****  Open the DAQ12.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## E.4 Analog Input

The DAQ-12 supports analog input requests with the following restrictions:

gain_array_ptr - because the DAQ-12 only supports one gain setting per acquisition, all of the values specified by gain_array_ptr must be the same.

trigger_source - only the INTERNAL_TRIGGER and TTL_TRIGGER selections are supported.

IO_mode - DMA_FOREGROUND and DMA_BACKGROUND modes are only supported for single channel operations (i.e. when array_length = 1).

clock_source - only the INTERNAL_CLOCK source is supported.

sample_rate - must be in the range 2.33 mHz (2.33e-3) ≤ sample_rate ≤ 200 kHz (200e3) for single channel operations or 153 Hz ≤ sample_rate ≤ [1 / (10µsec * array_length)] for multiple channel operations (i.e. array_length > 1)

calibration - only the NO_CALIBRATION selection is supported.

## E.5 Analog Output

The DAQ-12 supports analog output requests with the following restrictions:

array_length - only single channel operations are supported. Therefore array_length must equal 1.

trigger_source - only the INTERNAL_TRIGGER source is supported.

clock_source - only the INTERNAL_CLOCK source is supported.

sample_rate - must be greater than 153Hz. The maximum value of sample_rate is dependent upon the speed of the computer used.

calibration - only the NO_CALIBRATION selection is supported.

## E.6 Digital Input

The DAQ-12 supports digital input requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - only single point operations are supported. Therefore, number_of_scans must equal 1.

## E.7 Digital Output

The DAQ-12 supports digital output requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER is supported

IO_mode - only FOREGROUND_CPU mode is supported.

number_of_scans - only single point operations are supported. Therefore, number_of_scans must equal 1.

# Appendix F: DAQ-16

## F.1 Distribution Software

### F.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more DAQ-16s, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQ-16 libraries:

For Microsoft Visual C/C++

- DAQ16MS.LIB - small model DAQ-16 library
- DAQ16MM.LIB - medium model DAQ-16 library
- DAQ16MC.LIB - compact model DAQ-16 library
- DAQ16ML.LIB - large model DAQ-16 library

For Borland C/C++

- DAQ16BS.LIB - small model DAQ-16 library
- DAQ16BM.LIB - medium model DAQ-16 library
- DAQ16BC.LIB - compact model DAQ-16 library
- DAQ16BL.LIB - large model DAQ-16 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQ16.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQ-16 driver.

### F.1.2 Creating DOS Applications Using The TSR Drivers

Before running a DAQ-16 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the DAQ-16 TSR with the command line:

```
DAQ16TSR
```

This file, DAQ16TSR.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DAQ-16 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQ-16 driver will not be installed.

### F.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQ-16s is executed, it must be able to dynamically link to the DAQDRIVE and DAQ-16 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQ-16 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## F.2 Configuring The DAQ-16

Before DAQDRIVE can operate the DAQ-16, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### F.2.1 General Configuration

The DAQ-16's base address, interrupt level and DMA channels must be defined in the general configuration window of the configuration utility. These selections must reflect the configuration of switches SW1 and SW2 and jumpers J8, J9, J10, and J11 as defined in the DAQ-16 Hardware Reference Manual.

### F.2.2 A/D Converter Configuration

The DAQ-16's A/D converter must be configured for bipolar or unipolar operation and a gain value must be specified. These selections must reflect the configuration of jumpers J5 and J7 as defined in the DAQ-16 Hardware Reference Manual.

Furthermore, the data format jumper must be configured according to the input mode. If the A/D is configured for bipolar operation, the data format must be set to 2's complement using jumper J5. If the A/D is configured for unipolar operation, the data format must be set to binary using jumper J5. Jumper J6 determines input voltage range, which can be set as 10V, 5V or 2.5V. Once the input range setting is made, one should choose DAQ16_0.DAT ( for 10V ), DAQ16_1.DAT ( for 5V ), or DAQ16_2.DAT ( for 2.5V ) configuration data file.

### F.2.3 D/A Converter Configuration

The DAQ-16's D/A converter parameters are device type (bipolar or unipolar), reference source (internal or external ), and reference voltage. These selections must reflect the configuration of jumpers J3 and J4 as defined in the DAQ-16 Hardware Reference Manual.

### F.2.4 Digital I/O Configuration

The DAQ-16 contains 8 bits of digital I/O. The first 4 bits are fixed output and last 4 bits are fixed input. The logical channel assignments begin with digital I/O bit 0 and continue through to digital I/O bit 7.

### F.2.5 Timer Configuration

The DAQ-16 does not have any user-definable timer parameters.

## F.3 Opening The DAQ-16

### F.3.1 Using the DAQ-16 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQ-16, the application program must include DAQ16.H. In addition, the constant PROCEDURE must be replaced by DAQ16 (exactly and without quotes) and the device_type variable must be defined as "DAQ-16".

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq16.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "daq-16";
char *config_file = "daq-16.dat";

/*****  Open the DAQ-16.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ16, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### F.3.2 Using the DAQ-16 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQ-16, the TSR_number variable must be set to the value F002 hexadecimal (61, 442 decimal) and the device_type variable must be defined as "DAQ-16" for a DAQ-16 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf002;
char *device_type = "daq-16";
char *config_file = "daq-16.dat";

/*****  Open the daq-16.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### F.3.3 Using the DAQ-16 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DAQ-16, the DLL_name variable must specify the DAQ-16 dynamic link library (DAQ16.DLL) and the device_type variable must be defined as "DAQ-16".

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DAQ-16";
char *config_file = "DAQ-16.dat";
char *DLL_name    = "DAQ16.dll";

/*****  Open the DAQ-16.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## F.4 Analog Input

The DAQ-16 supports analog input requests with the following restrictions:

gain_array_ptr - because the DAQ-16 only supports one gain setting per acquisition, all of the values specified by gain_array_ptr must be the same. In addition, the value specified for the gain must match the value stored in the DAQ-16 configuration file.

trigger_source - only the INTERNAL_TRIGGER and TTL_TRIGGER sources are supported.

IO_mode - DMA_FOREGROUND and DMA_BACKGROUND modes are only supported for single channel operations (i.e. when array_length = 1).

clock_source - only the INTERNAL_CLOCK source is supported.

sample_rate - must be in the range 2.33 mHz (2.33e-3) ≤ sample_rate ≤ 100 kHz (100e3) for single channel operation or 153 Hz ≤ sample_rate ≤ [1/(10µsec * array_length)] for multiple channel operations (i.e. array_length > 1)

calibration - only the NO_CALIBRATION selection is supported.

## F.5 Analog Output

The DAQ-16 supports analog output requests with the following restrictions:

array_length - only single channel operations are supported. Therefore, array_length must equal 1.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported.

clock_source - only the INTERNAL_CLOCK source is supported.

sample_rate - must be greater than 153Hz. The maximum value of sample_rate is dependent upon the speed of the computer used.

calibration - only the NO_CALIBRATION selection is supported.

## F.6 Digital Input

The DAQ-16 supports digital input requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - only single point operations are supported. Therefore, number_of_scans must equal 1.

## F.7 Digital Output

The DAQ-16 supports digital output requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER is supported

IO_mode - only the FOREGROUND_CPU mode is supported.

number_of_scans - only single point operations are supported. Therefore, number_of_scans must equal 1.

# Appendix G: DAQ-801/802

## G.1 Distribution Software

### G.1.1 Creating DOS Applications Using The C Libraries

To generate an application that controls one or more DAQ-801/802s, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQ-801/802 libraries:

For Microsoft Visual C/C++

- DAQ800MS.LIB - small model DAQ-800 library
- DAQ800MM.LIB - medium model DAQ-800 library
- DAQ800MC.LIB - compact model DAQ-800 library
- DAQ800ML.LIB - large model DAQ-800 library

For Borland C/C++

- DAQ800BS.LIB - small model DAQ-800 library
- DAQ800BM.LIB - medium model DAQ-800 library
- DAQ800BC.LIB - compact model DAQ-800 library
- DAQ800BL.LIB - large model DAQ-800 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQ800.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQ-800 driver.

### G.1.2 Creating DOS Applications Using The TSR Drivers

Before running a DAQ-801/802 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in the section 2.4. Once the DAQDRIVE TSR is installed, the user can install the DAQ-801/802 TSR with the command line:

DAQ-800

This file, DAQ-800.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DAQ-801/802 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQ-801/802 driver will not be installed.

### G.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQ-801/802s is executed, it must be able to dynamically link to the DAQDRIVE and DAQ-800 series Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQ-800 series virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## G.2 Configuring The DAQ-801/802

Before DAQDRIVE can operate the DAQ-801/802, a configuration data file must be generated by the DAQDRIVE configuration utility. The DAQDRIVE configuration utility is discussed in section 2.2.

### G.2.1 General Configuration

The DAQ-801/802's base address and interrupt level must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 7FF0H with 10H interval. The base address value should reflect the DIP switch setting of SW1 and SW2 (refer to the DAQ-801/802 Hardware Manual).

### G.2.2 A/D Converter Configuration

The only A/D converter parameter needed to be set in DAQ-801/802 is device type (Bipolar or Unipolar).

### G.2.3 D/A Converter Configuration

The DAQ-801/802's D/A converter parameters are device type (bipolar or unipolar), reference source (internal or external ), reference voltage, and gain (gain of 1 or 2). These parameters should reflect the jumper setting of J2 and J4 of the board (refer to the DAQ-801/802 Hardware Manual).

### G.2.4 Digital I/O Configuration

The DAQ-801/802 has 32 bits of digital I/O. The first 24 bits are Port A, Port B, and Port C which are 8255A mode 0 equivalent. In the I/O port portion of the configuration window, the first 4 bits are fixed output and last 4 bits are fixed input. The 32 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group type. The group type are Port A, Port B, Port C bit 0 to 3, Port C bit 4 to 7, 4-bit fixed input and 4-bit fixed output. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 31.

### G.2.5 Timer Configuration

The DAQ-801/802 does not have any user-definable timer parameters.

## G.3 Opening The DAQ-801/802

### G.3.1 Using the DAQ-801/802 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQ-801/802, the application program must include DAQ800.H. In addition, the constant PROCEDURE must be replaced by DAQ800 (exactly and without quotes) and the device_type variable must be defined as "DAQ-801" for a DAQ-801 adapter or "DAQ-802" for a DAQ-802 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq800.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "daq-801";
char *config_file = "daq-801.dat";

/*****  Open the daq-801.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ800, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### G.3.2 Using the DAQ-801/802 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsigned short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQ-801/802, the TSR_number variable must be set to the value F003 hexadecimal (61, 443 decimal) and the device_type variable must be defined as "DAQ-801" for a DAQ-801 adapter or "DAQ-802" for a DAQ-802 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf003;
char *device_type = "DAQ-801";
char *config_file = "daq-801.dat";

/*****  Open the DAQ-801.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### G.3.3 Using the DAQ-801/802 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DAQ-801/802, the DLL_name variable must specify the DAQ-801/802 dynamic link library (DAQ800.DLL) and the device_type variable must be defined as "DAQ-801" for a DAQ-801 adapter or "DAQ-802" for a DAQ-802 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DAQ-802";
char *config_file = "DAQ-802.dat";
char *DLL_name    = "DAQ800.dll";

/*****  Open the DAQ802.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## G.4 Analog Input

The DAQ-801/802 supports analog input requests with the following restrictions:

channel_array_ptr - In the channel array, the channel numbers must be in sequential order from start channel to stop channel. If the start channel number is greater than stop channel number, it will wrap from channel 7 to channel 0 and continue to the end channel. For example, both {2,3,4,5,6} and {6,7,0,1,2} are valid channel lists.

trigger_source - INTERNAL_TRIGGER, TTL_TRIGGER, and ANALOG_TRIGGER sources are supported.

trigger_channel - trigger_channel MUST equal the first channel in the channel list.

trigger_voltage - The trigger voltage must be within the valid analog input range of trigger_channel.

IO_mode - Only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported.

clock_source - Only the INTERNAL_CLOCK source is supported.

sample_rate - sample_rate must be in the range 5.82e-4 Hz to 40 KHz (4e4).

## G.5 Analog Output

The DAQ-801/802 supports analog output requests with the following restrictions:

array_length - only single channel operations are supported. Therefore array_length must equal 1.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported.

clock_source - Only the INTERNAL_CLOCK source is supported.

sample_rate - The minimum value of sample_rate is 38.15Hz. The maximum value of sample_rate is depending on the speed of the computer used.

calibration - Only the NO_CALIBRATION selection is supported.

## G.6 Digital Input

The DAQ-801/802 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## G.7 Digital Output

The DAQ-801/802 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix H: DAQ-1201/1202

## H.1 Distribution Software

### H.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more DAQ-1201/1202s, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQ-1201/1202 libraries:

For Microsoft Visual C/C++

- DQ1200MS.LIB - small model DAQ-1200 library
- DQ1200MM.LIB - medium model DAQ-1200 library
- DQ1200MC.LIB - compact model DAQ-1200 library
- DQ1200ML.LIB - large model DAQ-1200 library

For Borland C/C++

- DQ1200BS.LIB - small model DAQ-1200 library
- DQ1200BM.LIB - medium model DAQ-1200 library
- DQ1200BC.LIB - compact model DAQ-1200 library
- DQ1200BL.LIB - large model DAQ-1200 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQ1200.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQ-1200 driver.

### H.1.2 Creating DOS Applications Using The TSR Drivers

Before running a DAQ-1201/1202 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR is installed, the user can install the DAQ-1200 TSR with the command line:

DAQ-1200

This file, DAQ-1200.EXE, is installed into the DAQDRIVE\TSR directory by teh DAQDRIVE installation program.

When the DAQ-1200 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQ-1200 driver will not be installed.

### H.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQ-1201/1202s is executed, it must be able to dynamically link to the DAQDRIVE and DAQ-1200 series Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQ-1200 series virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## H.2 Configuring The DAQ-1201/1202

Before DAQDRIVE can operate the DAQ-1201/1202, a configuration data file must be generated by the DAQDRIVE configuration utility program DAQCFGW.EXE for Microsoft Windows. The DAQDRIVE configuration utility is discussed in section 2.2.

### H.2.1 General Configuration

The DAQ-1201/1202's base address, interrupt level and DMA channels must be defined in the general configuration window of the configuration utility. The base address range is from 0 to 7FF0H with 10H interval. The base address value should reflect the DIP switch setting of SW1 and SW2 (refer to the DAQ-1201/1202 Hardware Manual).

### H.2.2 A/D Converter Configuration

The A/D converter parameters in DAQ-1201/1202 are device type (Bipolar or Unipolar), differential or single-ended.

### H.2.3 D/A Converter Configuration

The DAQ-1201/1202's D/A converter parameters are device type (bipolar or unipolar), reference source (internal or external ), reference voltage, and gain (gain of 1 or 2). These parameters should reflect the jumper settings of J4 and J5 of the board (refer to the DAQ-1201/1202 Hardware Manual).

### H.2.4 Digital I/O Configuration

The DAQ-1201/1202 has 32 bits of digital I/O. The first 24 bits are Port A, Port B, and Port C which are 8255A mode 0 equivalent. In the I/O port portion of the configuration window, the first 4 bits are fixed output and last 4 bits are fixed input. The 32 bits of digital I/O may be grouped into any combination of logical channels as long as the channels are in the same group type. The group type are Port A, Port B, Port C bit 0 to 3, Port C bit 4 to 7, 4-bit fixed input and 4-bit fixed output. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 31.

### H.2.5 Timer Configuration

The DAQ-1201/1202 does not have any user-definable timer parameters.

## H.3 Opening The DAQ-1201/1202

### H.3.1 Using the DAQ-1201/1202 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQ-1201/1202, the application program must include DAQ1200.H. In addition, the constant PROCEDURE must be replaced by the DAQ1200 (exactly and without quotes) and the device_type variable must be defined as "DAQ-1201" for a DAQ-1201 adapter or "DAQ-1202" for a DAQ-1202 adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daq1200.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "daq-1201";
char *config_file = "daq-1201.dat";

/*****  Open the daq-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQ1200, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### H.3.2 Using the DAQ-1201/1202 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsinged short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQ-1201/1202, the TSR_number variable must be set to the value F004 hexadecimal (61, 444 decimal) and the device_type variable must be defined as "DAQ1201" for a DAQ-1201 adapter or "DAQ1202" for a DAQ-1202 adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf004;
char *device_type = "daq-1201";
char *config_file = "daq-1201.dat";

/*****  Open the daq-1201.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### H.3.3 Using the DAQ-1201/1202 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DAQ-1201/1202, the DLL_name variable must specify the DAQ-1201/1202 dynamic link library (DAQ1200.DLL) and the device_type variable must be defined as "DAQ-1201" for a DAQ-1201 adapter or "DAQ-1202" for a DAQ-1202 adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DAQ-1202";
char *config_file = "DAQ-1202.dat";
char *DLL_name    = "DAQ1200.dll";

/*****  Open the DAQ1202.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## H.4 Analog Input

The DAQ-1201/1202 supports analog input requests with the following restrictions:

channel_array_ptr - Up to 512 channels array is supported.

trigger_source - INTERNAL_TRIGGER, TTL_TRIGGER, and ANALOG_TRIGGER sources are supported.

trigger_channel - trigger_channel MUST equal the first channel in the channel list.

trigger_voltage - The trigger voltage must be within the valid analog input range of trigger_channel.

clock_source - Only the INTERNAL_CLOCK source is supported.

sample_rate - sample_rate must be in the range 2.33e-5 Hz to 400 KHz (4e5).

calibration - Only the NO_CALIBRATION selection is supported.

## H.5 Analog Output

The DAQ-1201/1202 supports analog output requests with the following restrictions:

array_length - only single channel operations are supported. Therefore array_length must equal 1.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

clock_source - Only the INTERNAL_CLOCK source is supported.

sample_rate - The minimum value of sample_rate is 38.15Hz. The maximum value of sample_rate is depending on the speed of the computer used.

calibration - Only the NO_CALIBRATION selection is supported.

## H.6 Digital Input

The DAQ-1201/1202 supports digital input requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER source is supported.

IO_mode - Only the FOREGROUND_CPU data transfer mode is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

## H.7 Digital Output

The DAQ-1201/1202 supports digital output requests with the following restrictions:

channel_array_ptr - A channel may only appear once in the channel list.

trigger_source - Only the INTERNAL_TRIGGER is supported

IO_mode - Only the FOREGROUND_CPU is supported.

number_of_scans - Only single point operations are supported, therefore, number_of_scans must equal 1.

# Appendix I: DAQP-12 / DAQP-16

## I.1 Distribution Software

### I.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more DAQP-12, DAQP-12H, or DAQP-16 cards, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQP libraries:

For Microsoft Visual C/C++

- DAQP_CS.LIB - small model DAQP library
- DAQP_CM.LIB - medium model DAQP library
- DAQP_CC.LIB - compact model DAQP library
- DAQP_CL.LIB - large model DAQP library

For Borland C/C++

- DAQP_BS.LIB - small model DAQP library
- DAQP_BM.LIB - medium model DAQP library
- DAQP_BC.LIB - compact model DAQP library
- DAQP_BL.LIB - large model DAQP library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQP.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQP driver.

### I.1.2 Creating DOS Applications Using the TSR Driver

Before running a DAQP-12, DAQP-12H, or DAQP-16 application that uses the TSR driver, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR installed, the user may then install the DAQP TSR driver with the command line:

```
DAQPTSR
```

This file, DAQPTSR.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DAQP TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQP TSR driver will not be installed.

### I.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQP-12, DAQP-12H, or DAQP-16 cards is executed, it must be able to dynamically link to the DAQDRIVE and DAQP series Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQP series virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## I.2 Configuring The DAQP-12 / DAQP-12H / DAQP-16

Before DAQDRIVE can operate the DAQP-12, DAQP-12H, or DAQP-16, a configuration data file must be generated by the DAQDRIVE configuration utility program. The DAQDRIVE configuration utility is discussed in section 2.2.

### I.2.1 General Configuration

The DAQP-12's, DAQP-12H's, or DAQP-16's base address and interrupt level must be defined in the general configuration window of the configuration utility. If the base address is set to 0, DAQDRIVE will obtain the adapter's base address and interrupt level from the PCMCIA Card and Socket Services software.

NOTE: To operate in auto-configuration mode, the system must have the DAQP Client Driver and a version of Card and Socket Services software installed.

### I.2.2 A/D Converter Configuration

The DAQP-12, DAQP-12H, and DAQP-16 analog input channels are bipolar only. A differential or single-ended input option can be selected with the configuration utility. The gains are fully programmable and selected at run-time.

### I.2.3 Digital I/O Configuration

The DAQP-12, DAQP-12H, and DAQP-16 all have 4 bits of digital input and 4 bits of digital output. The default channel grouping is using the 4 digital output bits as channel 0 and the 4 digital input bits as channel 1.

### I.2.4 Timer Configuration

There are no user-definable timer parameters on DAQP-12, DAQP-12H, or DAQP-16.

## I.3 Opening The DAQP-12 / DAQP-12H / DAQP-16

### I.3.1 Using the DAQP-12 / DAQP-12H / DAQP-16 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQP-12, DAQP-12H, or DAQP-16 the application program must include DAQP.H. In addition, the constant PROCEDURE must be replaced by DAQP (exactly and without quotes) and the device_type variable must be defined as "DAQP-12", "DAQP-12H", or "DAQP-16" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-12,  set device_type to "DAQP-12"
// for DAQP-12H, set device_type to "DAQP-12H"
// for DAQP-16,  set device_type to "DAQP-16"

char *device_type = "DAQP-12";
char *config_file = "daqp-12.dat";

/*****  Open the DAQP-12.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### I.3.2 Using the DAQP-12 / DAQP-12H / DAQP-16 with the DOS TSR Driver

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The DOS TSR version of DaqOpenDevice is intended for DOS applications that interface to the "Terminate & Stay memory-Resident" (TSR) version of the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(unsigned short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQP-12, DAQP-12H, or DAQP-16 card, the TSR_number variable must be set to the value F005 hexadecimal (61,445 decimal). The device_type variable must be defined as "DAQP-12", "DAQP-12H", or "DAQP-16" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-12,  set device_type to "DAQP-12"
// for DAQP-12H, set device_type to "DAQP-12H"
// for DAQP-16,  set device_type to "DAQP-16"

unsigned short TSR_number = 0xf005;
char *device_type = "DAQP-16";
char *config_file = "daqp-16.dat";

/*****  Open the DAQP-16.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### I.3.3 Using the DAQP-12 / DAQP-12H / DAQP-16 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DAQP-12, DAQP-12H, or DAQP-16, the DLL_name variable must specify the DAQP dynamic link library (DAQPWIN.DLL) and the device_type variable must be defined as "DAQP-12", "DAQP-12H", or "DAQP-16" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-12,  set device_type to "DAQP-12"
// for DAQP-12H, set device_type to "DAQP-12H"
// for DAQP-16,  set device_type to "DAQP-16"

char *device_type = "DAQP-12";
char *config_file = "daqp-12.dat";
char *DLL_name    = "daqpwin.dll";

/*****  Open the DAQP-12.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## I.4 Analog Input

The DAQP-12, DAQP-12H, and DAQP-16 support analog input requests with the following restrictions:

channel_array_ptr - requests operating on two or more analog input channels. There is no restrictions on the number of times an analog input channel may appear in the channel list.

trigger_source - only the INTERNAL_TRIGGER and TTL_TRIGGER sources are supported. If the TTL_TRIGGER is selected, the trigger signal must be applied on the external trigger (shared with digital input 0) input.

IO_mode - only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported. DMA modes are NOT supported.

clock_source - both INTERNAL_CLOCK and EXTERNAL_CLOCK sources are supported. If the external clock is selected, the clock input has to be introduced from the external clock (shared with digital input bit 2) input. The minimum clock pulse width is 200 ns (or the maximum clock frequency is 5 MHz). There is no limit on the maximum clock width (or the minimum clock frequency).

sample_rate - sample_rate must NOT be over 100 kHz (100e3). If the internal clock is used, the minimum sampling rate is 0.06 Hz. The minimum sampling rate will be the external clock frequency divided by 16,777,215.

calibration - only the NO_CALIBRATION selection is supported.

*When the data acquisition is in background mode, the DAQDRIVE low level driver will select EOS (end of scan) interrupt if the data flow is less than 1000 samples per second (sampling rate times number of channels in the scan list), otherwise it uses the FIFO threshold interrupt. The FIFO thershold will always be set as an integer multiple of the scan length, as close as possible to the half full level.*

## I.5 Analog Output

Neither the DAQP-12, DAQP-12H, nor the DAQP-16 have any analog output channels. All analog output requests will return with a function not supported error.

## I.6 Digital Input

The DAQP-12, DAQP-12H, and DAQP-16 only supports single scan digital input requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

## I.7 Digital Output

The DAQP-12, DAQP-12H, and DAQP-16 support single scan digital output requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

# Appendix J: DAQP-208 / DAQP-208H / DAQP-308

## J.1 Distribution Software

### J.1.1 Creating DOS Applications Using the C Libraries

To generate an application that controls one or more DAQP-208, DAQP-208H, or DAQP-308 cards, the application must be linked with the appropriate DAQDRIVE library and one of the following DAQP libraries:

For Microsoft Visual C/C++

- DAQP_CS.LIB - small model DAQP library
- DAQP_CM.LIB - medium model DAQP library
- DAQP_CC.LIB - compact model DAQP library
- DAQP_CL.LIB - large model DAQP library

For Borland C/C++

- DAQP_BS.LIB - small model DAQP library
- DAQP_BM.LIB - medium model DAQP library
- DAQP_BC.LIB - compact model DAQP library
- DAQP_BL.LIB - large model DAQP library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DAQP.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DAQP driver.

### J.1.2 Creating DOS Applications Using the TSR Driver

Before running a DAQP-208, DAQP-208H, or DAQP-308 application that uses the TSR driver, the user must first load the DAQDRIVE TSR as discussed in section 2.4. Once the DAQDRIVE TSR installed, the user may then install the DAQP TSR driver with the command line:

```
DAQPTSR
```

This file, DAQPTSR.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DAQP TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DAQP TSR driver will not be installed.

### J.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DAQP-208, DAQP-208H, or DAQP-308 cards is executed, it must be able to dynamically link to the DAQDRIVE and DAQP series Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DAQP series virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## J.2 Configuring The DAQP-208 / DAQP-308 / DAQP-308

Before DAQDRIVE can operate the DAQP-208, DAQP-208H, or DAQP-308, a configuration data file must be generated by the DAQDRIVE configuration utility program. The DAQDRIVE configuration utility is discussed in section 2.2.

### J.2.1 General Configuration

The DAQP-208's, DAQP-208H's, or DAQP-308's base address and interrupt level must be defined in the general configuration window of the configuration utility. If the base address is set to 0, DAQDRIVE will obtain the adapter's base address and interrupt level from the PCMCIA Card and Socket Services software.

NOTE: To operate in auto-configuration mode, the system must have the DAQP Client Driver and a version of Card and Socket Services software installed.

### J.2.2 A/D Converter Configuration

The DAQP-208, DAQP-208H, and DAQP-308 analog input channels are bipolar only. A differential or single-ended input option can be selected with the configuration utility. The gains are fully programmable and selected at run-time.

### J.2.3 D/A Converter Configuration

There are no user-definable D/A parameters on DAQP-208, DAQP-208H, or DAQP-308.

### J.2.4 Digital I/O Configuration

The DAQP-208, DAQP-208H, and DAQP-308 all have 4 bits of digital input and 4 bits of digital output. The default channel grouping is using the 4 digital output bits as channel 0 and the 4 digital input bits as channel 1.

### J.2.5 Timer Configuration

There are no user-definable timer parameters on DAQP-208, DAQP-208H, or DAQP-308.

## J.3 Opening The DAQP-208 / DAQP-308

### J.3.1 Using the DAQP-208 / DAQP-308 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DAQP-208, DAQP-208H, or DAQP-308 the application program must include DAQP.H. In addition, the constant PROCEDURE must be replaced by DAQP (exactly and without quotes) and the device_type variable must be defined as "DAQP-208", "DAQP-208H", or "DAQP-308" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "daqp.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-208,  set device_type to "DAQP-208"
// for DAQP-208H, set device_type to "DAQP-208H"
// for DAQP-308,  set device_type to "DAQP-308"

char *device_type = "DAQP-208";
char *config_file = "daqp-208.dat";

/*****  Open the DAQP-208.  *****/

logical_device = 0;
status = DaqOpenDevice(DAQP, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### J.3.2 Using the DAQP-208 / DAQP-308 with the DOS TSR Driver

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The DOS TSR version of DaqOpenDevice is intended for DOS applications that interface to the "Terminate & Stay memory-Resident" (TSR) version of the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(unsigned short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DAQP-208, DAQP-208H, or DAQP-308 card, the TSR_number variable must be set to the value F005 hexadecimal (61,452 decimal) and the device_type variable must be defined as "DAQP-208", "DAQP-208H", or "DAQP-308" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-208,  set device_type to "DAQP-208"
// for DAQP-208H, set device_type to "DAQP-208H"
// for DAQP-308,  set device_type to "DAQP-308"

unsigned short TSR_number = 0xf00C;
char *device_type = "DAQP-308";
char *config_file = "daqp-308.dat";

/*****  Open the DAQP-308.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### J.3.3 Using the DAQP-208 / DAQP-208H / DAQP-308 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DAQP-208, DAQP-208H, or DAQP-308, the DLL_name variable must specify the DAQP dynamic link library (DAQPWIN.DLL) and the device_type variable must be defined as "DAQP-208", "DAQP-208H", or "DAQP-308" depending on the hardware in use.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

// for DAQP-208,  set device_type to "DAQP-208"
// for DAQP-208H, set device_type to "DAQP-208H"
// for DAQP-308,  set device_type to "DAQP-308"

char *device_type = "DAQP-208H";
char *config_file = "daqp208h.dat";
char *DLL_name    = "daqpwin.dll";

/*****  Open the DAQP-208H.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## J.4 Analog Input

The DAQP-208, DAQP-208H, and DAQP-308 support analog input requests with the following restrictions:

channel_array_ptr - requests operating on two or more analog input channels. There is no restrictions on the number of times an analog input channel may appear in the channel list.

trigger_source - only the INTERNAL_TRIGGER, TTL_TRIGGER and ANALOG_TRIGGER sources are supported.

trigger_channel - trigger_channel MUST be the first channel in the channel list.

trigger_voltage - The trigger voltage must be within the valid analog input range of the trigger_channel.

IO_mode - only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported. DMA modes are NOT supported

clock_source - both INTERNAL_CLOCK and EXTERNAL_CLOCK sources are supported. If the external clock is selected, the clock input has to be introduced from the external clock (shared with digital input bit 2) input. The minimum clock pulse width is 200 ns (or the maximum clock frequency is 5 MHz). There is no limit on the maximum clock width (or the minimum clock frequency).

sample_rate - sample_rate must NOT exceed 100 kHz (100e3). If the internal clock is used, the minimum sampling rate is 0.06 Hz. The minimum sampling rate is the clock source frequency divided by 16,777,215.

calibration - only the NO_CALIBRATION selection is supported.

## J.5 Analog Output

Both D/A channels of the DAQP-208, DAQP-208H, and DAQP-308 support analog output requests with the following restrictions:

channel_array_ptr - If there are two channels in the list, they MUST be different from each other.

trigger_source - only the INTERNAL_TRIGGER and TTL_TRIGGER sources are supported.

trigger_slope - only the RISING_EDGE TTL trigger is supported.

IO_mode - only the FOREGROUNG_CPU and BACKGROUNG_IRQ data transfer modes are supported.

clock_source - both INTERNAL_CLOCK and EXTERNAL_CLOCK sources are supported. If the external clock is selected, the clock input has to be introduced from the external clock (shared with digital input bit 2) input. The minimum clock pulse width is 200 ns (or the maximum clock frequency is 5 MHz). There is no limit on the maximum clock width (or the minimum clock frequency).

sample_rate - sample_rate must NOT exceed 100 kHz (100e3). If the internal clock is used, the minimum sampling rate is 15.3 Hz. The minimum sampling rate is the clock source frequency divided by 65,535.

calibration - only the NO_CALIBRATION selection is supported.

## J.6 Digital Input

Both DAQP-208 and DAQP-308 only support single scan digital input requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

## J.7 Digital Output

Both DAQP-208 and DAQP-308 only support single scan digital output requests with the following restrictions:

channel_array_ptr - a channel may only appear once in the channel list.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU data transfer mode is supported.

# Appendix K: DA8P-12

## K.1 Distribution Software

### K.1.1 Creating DOS Applications Using The C Libraries

To generate an application that controls one or more DA8P-12s, the application must be linked with the appropriate DAQDRIVE library and one of the following DA8P-12 libraries:

For Microsoft Visual C/C++

- DA8P12CS.LIB - small model DA8P-12 library
- DA8P12CM.LIB - medium model DA8P-12 library
- DA8P12CC.LIB - compact model DA8P-12 library
- DA8P12CL.LIB - large model DA8P-12 library

For Borland C/C++

- DA8P12BS.LIB - small model DA8P-12 library
- DA8P12BM.LIB - medium model DA8P-12 library
- DA8P12BC.LIB - compact model DA8P-12 library
- DA8P12BL.LIB - large model DA8P-12 library

The selected libraries **MUST** match the compiler and memory model specified for the application program. These libraries are installed into the DAQDRIVE\C_LIBS directory by the DAQDRIVE installation program.

The application program must also include the file DA8P-12.H installed into the DAQDRIVE\C_LIBS directory. This file defines the "open" procedure for the C library version of the DA8P-12 driver.

### K.1.2 Creating DOS Applications Using The TSR Drivers

Before running a DA8P-12 application that uses the TSR drivers, the user must first load the DAQDRIVE TSR as discussed in the DAQDRIVE User's Manual. Once the DAQDRIVE TSR is installed, the user can install the DA8P-12 TSR with the command line:

DA8P-12

This file, DA8P-12.EXE, is installed into the DAQDRIVE\TSR directory by the DAQDRIVE installation program.

When the DA8P-12 TSR driver is executed, it will search for the DAQDRIVE TSR in memory and install itself on the same software interrupt. If the DAQDRIVE TSR is not loaded in memory, an error will be reported and the DA8P-12 driver will not be installed.

### K.1.3 Creating Windows Applications

When a Microsoft Windows application that controls one or more DA8P-12s is executed, it must be able to dynamically link to the DAQDRIVE and DA8P-12 Dynamic Link Libraries (DLLs). 32-bit Windows 95/98 applications must also be able to link to the DAQDRIVE and DA8P-12 virtual device drivers (VxDs). Windows searches for any required DLL and/or VxD files in the following locations:

1. the current directory
2. the Windows directory
3. the Windows\System directory
4. the directory of the application program
5. all directories specified by the PATH environment variable
6. all directories mapped to network drives

The DAQDRIVE installation program copies all of the necessary DLLs and/or VxDs into the WINDOWS\SYSTEM directory.

## K.2 Configuring The DA8P-12

Before DAQDRIVE can operate the DA8P-12, a configuration data file must be generated by the DAQDRIVE configuration utility. The DAQDRIVE configuration utility is discussed in section 2.2.

### K.2.1 General Configuration

The DA8P-12's base address and interrupt level must be defined in the general configuration window of the configuration utility. If the base address is set to 0, DAQDRIVE will obtain the DA8P-12's base address and interrupt level from the PCMCIA Card and Socket Services software.

NOTE: To operate in auto-configuration mode, the system must have the DA8P-12's Client Driver and a version of Card and Socket Services software installed.

### K.2.2 D/A Converter Configuration

The DA8P-12 does not have any user-definable D/A converter parameters. The DA8P-12B is factory configured for 8 bipolar outputs. The DA8P-12U is factory configured for 8 unipolar outputs.

### K.2.3 Digital I/O Configuration

The DA8P-12 has 8 bits of digital I/O which may be grouped into any combination of logical channels. The logical channel assignments begin with digital I/O bit 0 and continue through digital I/O bit 7. After all of the logical channels have been defined, each channel may be individually configured for input, output, or input/output modes.

### K.2.4 Timer Configuration

The DA8P-12 does not have any user-definable timer parameters.

## K.3 Opening The DA8P-12

### K.3.1 Using the DA8P-12 with the C libraries

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The C library version of DaqOpenDevice is intended for DOS applications that are written in C and linked directly to the DAQDRIVE libraries.

```
unsigned short DaqOpenDevice(PROCEDURE,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

This version of DaqOpenDevice is implemented as a C macro and uses the token pasting operator to create a unique "open" command for the desired adapter. In order to open a DA8P-12, the application program must include DA8P-12.H. In addition, the constant PROCEDURE must be replaced by DA8P12 (exactly and without quotes) and the device_type variable must be defined as "DA8P-12B" for a bipolar adapter or "DA8P-12U" for a unipolar adapter.

```
#include "daqdrive.h"
#include "daqopenc.h"
#include "userdata.h"
#include "da8p-12.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DA8P-12B";
char *config_file = "da8p-12b.dat";

/*****  Open the DA8P-12.  *****/

logical_device = 0;
status = DaqOpenDevice(DA8P12, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### K.3.2 Using the DA8P-12 with the TSR drivers

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The TSR version of DaqOpenDevice is intended for DOS applications that interface to the memory resident (TSR) version of the DAQDRIVE drivers.

```
unsigned short DaqOpenDevice(unsigned short TSR_number,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

Each hardware device supported by DAQDRIVE has been assigned a unique TSR_number value to be used with the DaqOpenDevice procedure. In order to open a DA8P-12, the TSR_number variable must be set to the value F006 hexadecimal (61, 446 decimal) and the device_type variable must be defined as "DA8P-12B" for a bipolar adapter or "DA8P-12U" for a unipolar adapter.

```
#include "daqdrive.h"
#include "daqopent.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

unsigned short TSR_number = 0xf006;
char *device_type = "DA8P-12B";
char *config_file = "da8p-12b.dat";

/*****  Open the DA8P-12.  *****/

logical_device = 0;
status = DaqOpenDevice(TSR_number, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

### K.3.3 Using the DA8P-12 with Windows

DaqOpenDevice is the only procedure that is implemented differently depending upon the type of interface between DAQDRIVE and the application program. The Windows version of DaqOpenDevice is intended for Windows applications that interface to the DAQDRIVE dynamic link libraries (DLLs).

```
unsigned short DaqOpenDevice(char *DLL_name,
                             unsigned short *logical_device,
                             char *device_type,
                             char *config_file)
```

In order to open a DA8P-12, the DLL_name variable must specify the DA8P-12 dynamic link library (DA8P-12.DLL) and the device_type variable must be defined as "DA8P-12B" for a bipolar adapter or "DA8P-12U" for a unipolar adapter.

```
#include "daqdrive.h"
#include "daqopenw.h"
#include "userdata.h"

unsigned short main()
{
unsigned short logical_device;
unsigned short status;

char *device_type = "DA8P-12B";
char *config_file = "da8p-12b.dat";
char *DLL_name    = "da8p-12.dll";

/*****  Open the DA8P-12.  *****/

logical_device = 0;
status = DaqOpenDevice(DLL_name, &logical_device, device_type, config_file);
if (status != 0)
   {
   printf("Error opening configuration file.  Status code %d.\n",status);
   exit(status);
   }
```

## K.4 Analog Input

The DA8P-12 does not support any analog input functions. All analog input requests will return with a function not supported error.

## K.5 Analog Output

The DA8P-12 supports analog output requests with the following restrictions:

channel_array_ptr - requests operating on more than one analog output channel use the DA8P-12's simultaneous output mode. This restricts channels to a single appearance in the channel list.

trigger_source - only the INTERNAL_TRIGGER and TTL_TRIGGER sources are supported. If the TTL_TRIGGER is selected for a single channel request, the trigger signal must be applied on the DA8P-12's external event input. If the TTL_TRIGGER is specified for a multiple channel request, the trigger signal must be applied on the external load control input.

trigger_slope - only RISING_EDGE TTL triggers are supported.

IO_mode - only the FOREGROUND_CPU and BACKGROUND_IRQ data transfer modes are supported.

clock_source - only the INTERNAL_CLOCK source is supported.

sample_rate - sample_rate must be in the range 500 nHz (500e-9) to 100 KHz (100e3).

calibration - only the NO_CALIBRATION selection is supported.

## K.6 Digital Input

The DA8P-12 supports digital input requests with the following restrictions:

channel_array_ptr - a channel may appear only once in the channel list.

number_of_scans - only one value may be input from each channel per request. Therefore, number_of_scans must equal 1.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU mode is supported.

## K.7 Digital Output

The DA8P-12 supports digital output requests with the following restrictions:

channel_array_ptr - a channel may appear only once in the channel list.

number_of_scans - only one value may be output to each channel per request. Therefore, number_of_scans must equal 1.

trigger_source - only the INTERNAL_TRIGGER source is supported.

IO_mode - only the FOREGROUND_CPU mode is supported.

DAQDRIVE
Users Manual
Version 2.32
January 14, 1999
Part No. 940-0100-232

# DAQDRIVE User's Manual Notes

## README FIRST!

This manual uses WordPro 97 Document Fields extensively to create different company specific versions of the document. The table on the following page specifies the Document Field names and the values used for each version of the manual. Document field values may be edited by selecting **File|Document Properties|Document** off the menu. However, macros to automatically switch all the fields for each company have been written, which is much easier. See the **Edit|Scripts & Macros** menu to run or edit one of these macros. The macros are not linked to the table on the following page, so if you are going to change any of the Document Field values, you must edit both locations.

## Current Document Field Settings:

```
Company Name:          Omega Engineering, Inc.
Company Alias:         Omega
Footer Name:
DAQDRIVE Name          DAQDRIVE

Card1_Name       DAQP-16
Card2_Name       DAQ-1201
Card3_Name       DAQP-208
Card4_Name       DA8P-12B
Card5_Name       IOP-241
```

**Problems found with WordPro 97 using Divisions, why this is one huge document.**

1. BookMarks have problems spanning Divisions across multiple files.
2. BookMark power fields are broke if you rename a Division.
3. External files (documents) are referenced by a hard coded path to the external file(s). If you move the project, must re-locate each external division file individually.

| Document Field | x | Quatech | Keithley | Omega |
|---|---|---|---|---|
| Company Name | x | Quatech Inc. | Keithley Instruments, Inc. | Omega Engineering, Inc. |
| Company Alias | x | Quatech | Keithley | Omega |
| Footer Name | x | Quatech Inc. | Keithley Instruments, Inc. | <blank> |
| DAQDRIVE Name | x | DAQDRIVE | DAQDRIVE | DAQDRIVE |
| DAQDRIVE_Files | x | DAQDRIVE | DAQDRIVE | DAQDRIVE |
| DAQDRV_C_Libs | x | DAQDRV | DAQDRV | DAQDRV |
| | | | | |
| Card1_Name | | DAQP-16 | KPCMCIA-16AI | DAQP-16 |
| Card1_Dev_Type | | DAQP-16 | KPC-16AI | DAQP-16 |
| Card1_Config_File | | DAQP-16.DAT | KPC-16AI.DAT | DAQP-16.DAT |
| Card1_C_Libs | | DAQP | KAI | DAQP |
| Card1_C_Hfile | | DAQP.H | KAI.H | DAQP.H |
| Card1_C_Open | | DAQP | KAI | DAQP |
| Card1_DLL | | DAQPWIN.DLL | KPC-AI.DLL | DAQPWIN.DLL |
| Card1_TSR | | DAQPTSR | KAITSR | DAQPTSR |
| | | | | |
| Card2_Name | | DAQ-1201 | KPCMCIA-12AIAO | DAQ-1201 |
| Card2_Dev_Type | | DAQ-1201 | K-12AIAO | DAQ-1201 |
| Card2_Config_File | | DAQ-1201.DAT | K-12AIAO.DAT | DAQ-1201.DAT |
| Card2_C_Libs | | DQ1200 | KAI | DQ1200 |
| Card2_C_Hfile | | DAQ1200.H | KAI.H | DAQ1200.H |
| Card2_C_Open | | DAQ1200 | KAI | DAQ1200 |
| Card2_DLL | | DAQ1200.DLL | KPC-AI.DLL | DAQ1200.DLL |
| Card2_TSR | | DAQ-1200 | KAITSR | DAQ-1200 |
| | | | | |
| Card3_Name | | DAQP-208 | KPCMCIA-12AIAO | DAQP-208 |
| Card3_Dev_Type | | DAQP-208 | K-12AIAO | DAQP-208 |
| Card3_Config_File | | DAQP-208.DAT | K-12AIAO.DAT | DAQP-208.DAT |
| Card3_C_Libs | | DAQP | KAI | DAQP |
| Card3_C_Hfile | | DAQP.H | KAI.H | DAQP.H |
| Card3_C_Open | | DAQP | KAI | DAQP |
| Card3_DLL | | DAQPWIN.DLL | KPC-AI.DLL | DAQPWIN.DLL |
| Card3_TSR | | DAQPTSR | KAITSR | DAQPTSR |
| | | | | |
| Card4_Name | | DA8P-12B | KPCMCIA-8AOB | DA8P-12B |
| Card4_Dev_Type | | DA8P-12B | KPC-8AOB | DA8P-12B |
| Card4_Config_File | | DA8P-12B.DAT | KPC-8AOB.DAT | DA8P-12B.DAT |
| Card4_C_Libs | | DA8P12 | K8AO | DA8P12 |
| Card4_C_Hfile | | DA8P-12.H | K8AO.H | DA8P-12.H |
| Card4_C_Open | | DA8P-12 | K8AO | DAQP |
| Card4_DLL | | DA8P-12.DLL | KPC-8AO.DLL | DA8P-12.DLL |
| Card4_TSR | | DA8P-12 | K8AO | DA8P-12 |
| | | | | |
| Card5_Name | | IOP-241 | KPCMCIA-PIO24 | IOP-241 |
| Card5_Dev_Type | | IOP-241 | K-PIO24 | IOP-241 |
| Card5_Config_File | | IOP-241.DAT | K-PIO24.DAT | IOP-241.DAT |
| Card5_C_Libs | | IOP241 | KPIO24 | IOP241 |
| Card4_C_Hfile | | IOP241.H | KPIO24.H | IOP241.H |
| Card1_C_Open | | IOP241 | KPIO24 | IOP241 |
| Card5_DLL | | IOP-241.DLL | K-PIO24.DLL | IOP-241.DLL |
| Card5_TSR | | IOP-241 | KPIO24 | IOP-241 |