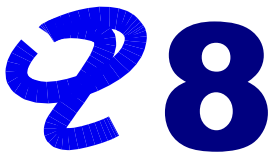

Data Acquisition System

User's Guide

Version 1.0

Another Quality Product
by Quanser Consulting

How to contact Quanser Consulting:

	(905) 940-3575	Telephone
	(905) 940-3576	Facsimile
	80 Esna Park Drive Markham, ON Canada L3R 2K8	Mail
	http://www.quanser.com	Web
	mailto://info@quanser.com	General information

Q8 Data Acquisition System User's Guide

Printed in Canada.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Overview

The Q8 is a versatile and powerful measurement and control board with an extensive range of input and output support. A wide variety of devices with analog and digital sensors as well as quadrature encoders are easily connected to the Q8. This single-board solution is ideal for use in control systems and complex measurement applications.

No other board offers this combination of features and performance! It is ideal for control systems & measurement of complex systems:

- High resolution - 14-bit inputs
- High speed sampling up to 350kHz
- Simultaneous sampling of A/D, digital and encoder inputs
- Extensive I/O: 8 each of A/D, D/A, encoders & 32 digital I/O on the same board
- Integrated with MATLAB/Simulink/RTW via Quanser's WinCon and SLX solutions
- PWM outputs on-board

This single board integration enables you to turn your PC into a powerful Desktop Control Station. A schematic of the Q8 architecture is shown in Figure 1 below. Designed by control engineers for control engineers, the Q8 provides:

- 8 x 14-bit programmable analog inputs
- Up to 350 kS/s sampling frequency on 2 A/D channel (100 kHz on all 8 channels)
- 8x 12-bit D/A voltage outputs
- 8 quadrature encoder inputs
- 32 programmable digital I/O channels
- Simultaneous sampling of both analog, digital and encoder sections
- 2x 32-bit dedicated counter/timers, including watchdog functionality
- 8x 24-bit reconfigurable encoder counter/timers
- 2x on-board PWM outputs
- 32-bit, 33 MHz PCI bus interface
- Supports Quanser real-time control software WinCon (2000/XP) & SimuLinuxRT (RTLinux)
- Totem Pole digital I/O for high speed
- Easy synchronization of multiple Q8 boards

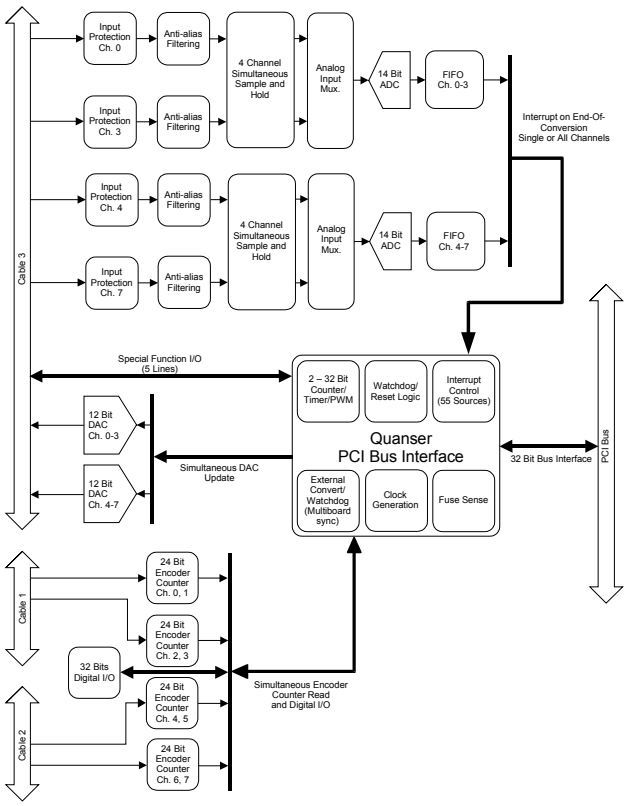

Figure 1 Q8 Architecture

Analog Inputs

The Q8 has two analog-to-digital (A/D) converters on board. Each A/D converter handles four channels, for a total of 8 single-ended analog inputs. Each A/D samples all four channels simultaneously and holds the sampled signals while it converts the analog value to a 14-bit digital code. The results are stored in an onboard FIFO queue that can be read by software. A/D conversions can be initiated manually, or using one of the 32-bit general purpose counters on board. Each A/D converter can interrupt the PC for each conversion, or when all conversions are complete. Interrupting when all conversions are complete is particularly useful for control, because the general purpose counter can be programmed to trigger A/D conversions at the controller's sampling rate and the "conversions-complete" interrupt can be used to invoke the interrupt service routine (ISR). The ISR can then read the results directly from the A/D FIFO without having to poll the A/D.

Each channel includes anti-aliasing filtering and input protection against ESD and improper connections. The inputs may range from -10V to +10V.

Analog Outputs

The Q8 has two 12-bit digital-to-analog (D/A) converters on board. Each D/A converter outputs four channels, for a total of 8 analog outputs. The D/A converters are double-buffered, so new output values can be preloaded in to the D/A converters, and all analog outputs updated simultaneously.

Each analog output channel can be individually programmed to be in the range 0V to 10V, -5V to 5V, or -10V to +10V. Internal output amplifiers in the D/A converters allow the outputs to sink 5mA and to drive up to a 500pF load.

Digital I/O

The Q8 has 32 channels of digital I/O. The channels are individually programmable as an input or output. All 32 channels may be read or written simultaneously. The outputs are TTL and CMOS compatible.

Encoder Inputs

The Q8 contains four encoder chips, each handling two channels, for a total of 8 encoder inputs. All four encoder chips can be accessed in a single 32-bit operation, and both channels can be accessed at the same time per chip. Hence, all eight encoder inputs may be processed simultaneously.

Each encoder input channel has a 24-bit counter that can be configured in a variety of modes, including module-N for frequency-divider applications. Full 4X quadrature counting is supported as well as a non-quadrature count/direction mode. The index pulse for each channel is also fully supported. The encoder inputs are digitally filtered, with the filter clock frequency individually programmable for each channel (the maximum rate is 16.7 MHz). The Q8 can interrupt on a large variety of events from the encoders, such as the index pulse, carry, borrow and compare flags, error conditions, etc.

PWM Outputs

The two 32-bit general purpose counters on the Q8 can be used as PWM outputs, with 30ns resolution. For example, each counter can generate a 10-bit, 16 kHz, PWM signal. If fewer bits of resolution are required, then a faster PWM frequency can be used. Table 1 below enumerates a few sample resolutions and the associated PWM frequencies.

Resolution	*PWM Frequency*
8 bits	65 kHz
10 bits	16 kHz
12 bits	4 kHz

Table 1 PWM frequency versus resolution

Since the two 32-bit counters can be synchronized, the two PWM outputs can also be synchronized. The two PWM outputs are part of the Special Function I/O lines of the Q8, illustrated in Figure 1 on page 2.

Counters and Watchdog Features

In addition to the eight 24-bit encoder counters, the Q8 has two 32-bit general purpose counters. These counters have a 30ns resolution and can be programmed to generate PWM or square wave outputs. Each counter may also be used as a periodic interrupt source. Each counter also has some special functions it can perform.

One counter may be used to trigger A/D conversions automatically. The counter actually triggers a set of conversions on any or all of the channels of an A/D converter. The conversions are then performed at the fastest rate (2.4 usecs per channel or 3.36 usecs per channel if simultaneous sampling is desired across both A/D converters).

The other counter may be used as a watchdog timer. When the counter output goes high, it can be programmed to reset all the analog outputs to zero volts, and to bring all the digital outputs high. Software typically resets the watchdog counter in its interrupt service routine before the counter output goes high. If the software fails, then the counter expires and the hardware resets all the outputs. The Q8 can interrupt the PC when a watchdog event occurs so that software can detect the event immediately.

Special Function I/O and Multiboard Synchronization

In addition to its 32 digital I/O channels, the Q8 has four additional channels for special functions: two digital inputs and two digital outputs.

The digital outputs may be programmed as square wave or PWM outputs from the two 32-bit general purpose counters.

One of the outputs, the WATCHDOG line, can also be programmed as a "watchdog output". This output reflects the status of the watchdog counter. If a watchdog event occurs, the output can be programmed to go low, in order to propagate watchdog events to other Q8

cards, or to disable other hardware, such as motor controllers, in the watchdog timer is not reset.

The two special function digital inputs are the EXT_INT line and the CNTR_EN line. The EXT_INT line can be used to generate interrupts from an external interrupt source, such as a limit switch or emergency stop button.

The EXT_INT can also trigger a watchdog event. In this mode, the EXT_INT line can be used to reset all the analog outputs and set all the digital outputs of the Q8 through hardware. Combined with the WATCHDOG line, this input can be used to propagate the watchdog event from one Q8 to another so that all Q8 cards in the chain are reset, in hardware, by a single watchdog counter. This input can also be used to reset the Q8 outputs in the event that an emergency stop button is pressed, for example, without any software intervention!

The CNTR_EN line is used to gate one of the 32-bit general purpose counters. A signal on the CNTR_EN line can stop the counter from counting. This input can also be programmed to trigger A/D conversions. Thus, A/D conversions can be triggered from an external source. For example, by using the general purpose counter of one Q8 to trigger A/D conversions, and then tying the output of that counter to the CNTR_EN input of another Q8, the two Q8 cards (or more) can be synchronized so that all the Q8 cards in the chain perform their A/D conversions at virtually the same time.

Fuse Sense

The Q8 terminal board is fused to prevent too much current from being drawn from the Q8 multifunction I/O card. The power to the terminal board, including any attached encoders is supplied through this fuse. If the fuse blows, it could have a serious effect upon a closed loop control system involving the encoders, because the encoders will appear to stop counting, even though the system may still be moving.

Thus, the Q8 senses the output of the fuse to ensure that it has not blown. If the fuse blows, the Q8 will automatically reset all the analog outputs and pull all digital outputs high – just like a watchdog event. An interrupt can also be generated to notify the application software. This feature is an important addition to the many safety feature of the Q8 data acquisition system.

Setting Up the Q8 System

This chapter describes how to set up the Q8 data acquisition system. Setting up your data acquisition system involves inserting the Q8 data acquisition card and connecting the Q8 terminal board to the card. The installation is very simple, but it is important that the instructions in this document be followed carefully.

Installing the Data Acquisition Card

The Q8 data acquisition card is susceptible to damage from static electricity. Always ground yourself when installing the card in your computer!

The Q8 data acquisition card is depicted in Figure 2. There are three connectors on the card: two 44-pin ribbon cable connectors and one 50-pin ribbon cable connector. These connectors are labelled `Cable 1, Cable 2` and `Cable 3`, as shown in the figure. The connectors are labelled on the terminal board also. Note that the connector closest to the mounting bracket is `Cable 3`, not `Cable 1`. Hence, from closest to the mounting bracket to farthest from the bracket, the connectors appear in the order: `Cable 3`, `Cable 2` and `Cable 1`.

1. Read all instructions before proceeding.
2. Make sure you are grounded. Quanser Consulting highly recommends that you wear proper ESD protection, such as a grounded wrist strap, when handling the Q8 data acquisition card. Quanser Consulting cannot take responsibility for mishandling of the card.
3. Connect the ribbon cables to the connectors, if they are not connected already. The cables are all different lengths. The shortest cable should be inserted into the `Cable 3` connector. Insert it first. **The connectors are stiff, and care must be taken not to bend any pins of the connectors. The ribbon cables should be inserted such that the red line on the outside edge of the ribbon cable is closest to the PCI bus connector on the Q8 card The connectors have a tab such that they only go in one way.** Do the `Cable 2` connector next. The longest ribbon cable plugs into the `Cable 1` connector. Plug it in last.
4. Unscrew the mounting bracket, taking note of its orientation, and then remount it so that the ribbon cables fit through the slot between the mounting bracket and the board. If the ribbon cables are already connected, then you do not need to perform this step.
5. Take the cover off of your computer.
6. Select one of the available PCI slots in your computer. In a tower unit, the components on the PCI card will typically face downward when installed in the PCI slot so that the card appears to be "upside-down". If you have other PCI cards already installed in your

computer, the Q8 card will be oriented the same way.

7. Feed the ribbon cables through the opening in the back of your computer next to the PCI slot you have selected. **The sides of the opening may be sharp, so be careful not to scrape the ribbon cables.** The easiest way to get the ribbon cables through the opening is to put one connector at a time through the slot, starting with the longest cable, `Cable 1`. Orient the connectors so that they fit more readily through the slot. Be sure to have the data acquisition card oriented the right way before feeding the ribbon cables through the opening.
8. Once the ribbon cables have been fed through the opening, insert the Q8 data acquisition card into the PCI slot. Excessive force is not required to insert the card. Attach the card's mounting bracket to the computer chassis using the mounting screw.
9. Take note of which ribbon cable is connected to which connector on the Q8 data acquisition card before putting the cover back on your PC. You may wish to label the ribbon cables with masking tape. In a tower unit, with the Q8 card mounted component-side downward, the topmost ribbon cable coming out of the back of the PC is `Cable 3`. The bottom-most cable is `Cable 1`.

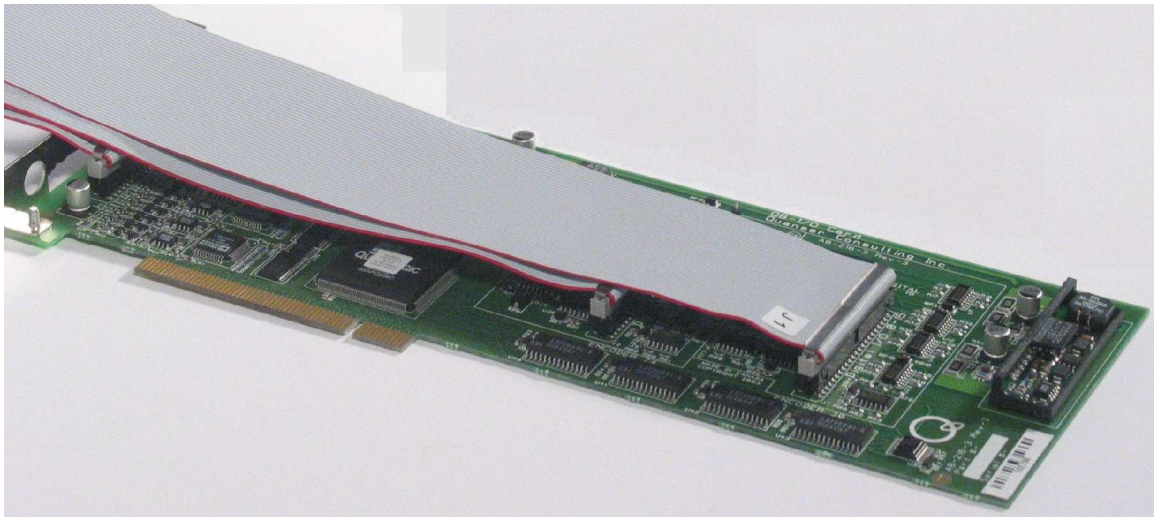

Figure 2 Q8 Data Acquisition Card

Connecting the Terminal Board

The Q8 terminal board is illustrated in Figure 3. The terminal board is designed to sit on top of a tower unit, if desired. However, it does not have to be used in this configuration. At this point, the Q8 data acquisition card should be installed in your PC, and the ribbon cables should be connected to the data acquisition card, as outlined in the previous section.

1. Plug `Cable 3` into the `Cable 3` connector on the Q8 terminal board. `Cable 3` is the shortest cable. In a tower unit, it will also be the topmost cable where it comes out of the PC. **The red line on the cable should be closest to the analog input RCA connectors and the encoder DIN connectors on the Q8 terminal board. The connectors have a tab such that they only go in one way.**
2. Plug `Cable 2` into the `Cable 2` connector the Q8 terminal board. `Cable 2` is the second shortest cable. In a tower unit, it is the middle cable where the ribbon cables exit the PC.
3. The last cable is `Cable 1`. It is the longest ribbon cable. Connect it to the `Cable 1` connector on the Q8 terminal board.
4. With the terminal board on top of the tower unit, the ribbon cables should not be twisted and should all lie neatly on top of one another. The cables are slightly offset from one another on the terminal board to match the offset of the connectors on the data acquisition card. The text on the terminal board should face the front of the computer.
5. Use the ribbon cable clamp provided to secure the ribbon cables together. The clamp comes with a peel-and-stick base so that you can mount the clamp to a clean surface, such as the top of your PC. Put the cables in the clamp before mounting it. If you do not wish to mount the clamp, then do not remove the peel-and-stick paper. Use of the clamp is unnecessary, but helps to keep your Q8 data acquisition system neat and tidy.

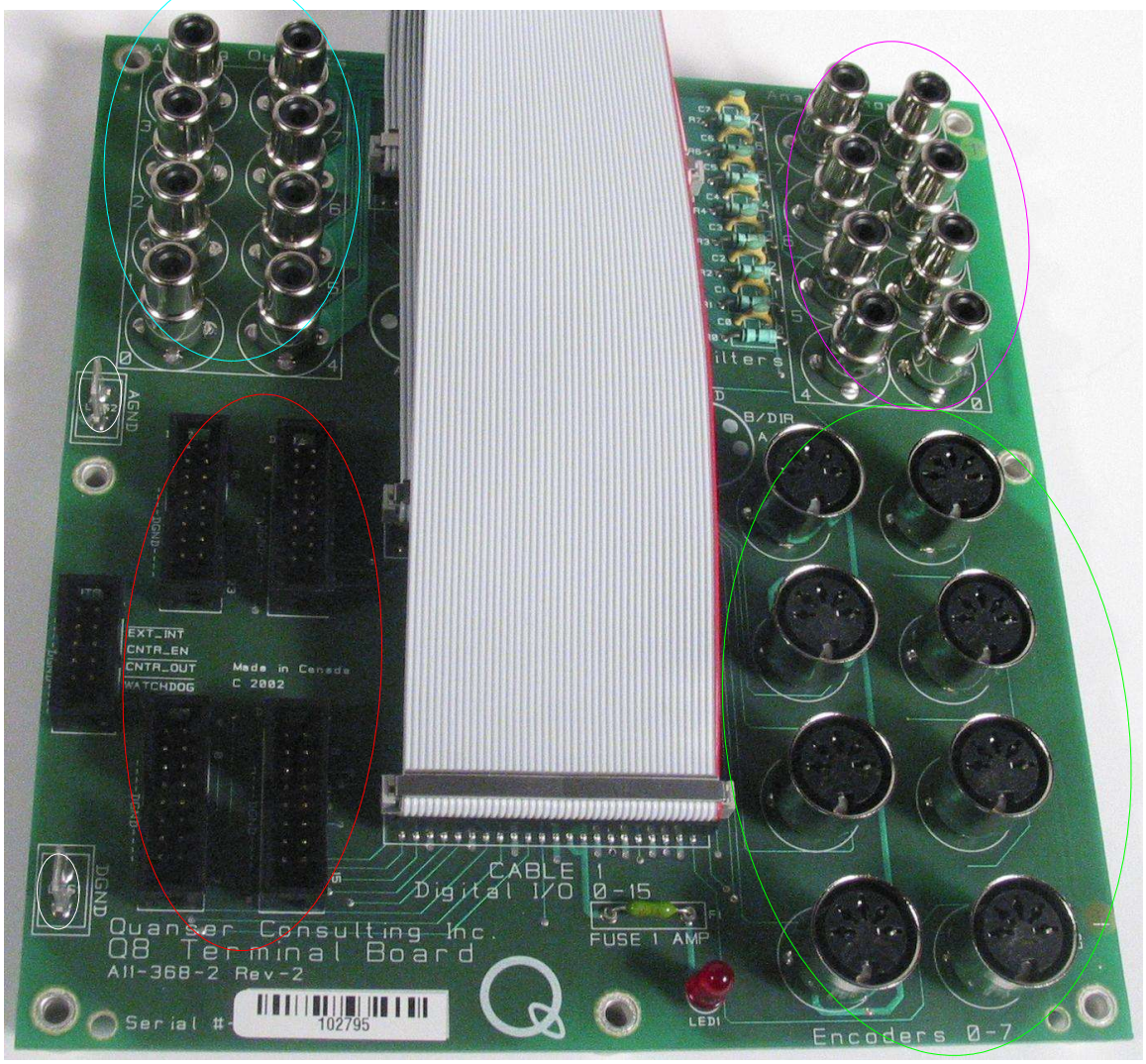

Figure 3 Q8 Terminal Board

Installing the Q8 Driver

To install the Windows 2000 or XP driver for the Q8 card, simply insert the card into your machine as per the instructions above, and then boot your computer. When Windows identifies the card and asks where to search for drivers, check the "Specify a location" option. Choose the subdirectory on the floppy or CD-ROM suitable for your operating system. For example, on the Q8 driver floppy, choose A:\Win2k for Windows 2000, or A:\WinXP for Windows XP. Continue with the installation until Windows has installed the driver for the card. The Q8 card shows up in Device Manager under the new "Data Acquisition Cards" section.

Windows 98/Me/2000/XP

Because the above operating systems support plug and play, the Plug and Play (PNP) Operating System option in the BIOS must be enabled so that plug and play components are handled by the operating system and not the BIOS.

Windows NT

Because Windows NT is not a plug and play operating system, the Plug and Play (PNP) Operating System option in the BIOS must be disabled so that plug and play components are handled by the BIOS.

Using the Q8 Terminal Board

The Q8 terminal board offers a convenient and flexible system for connecting your experiments to the Q8 data acquisition system. The board supports eight analog outputs, eight single-ended analog inputs, eight singled-ended encoder inputs, 32 digital I/O channels and 4 special purpose digital I/O channels, providing two PWM outputs, an interrupt line, A/D triggering, watchdog firing and counter gating. Low-pass analog filtering is also available on the board. The Q8 terminal board offers the flexibility you need!

Digital I/O

The Q8 data acquisition card supports 32 digital I/O channels. Each individual channel is software programmable as an input or an output. These 32 channels are organized into four banks of 8 channels each on the Q8 terminal board. The digital I/O connectors are shown in light red in Figure 3 above.

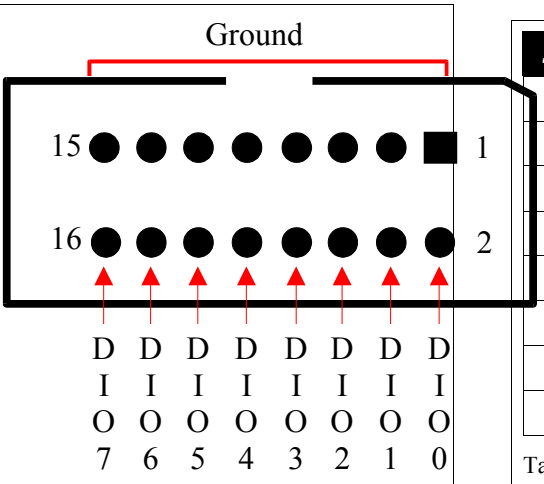

Figure 4 Digiital I/O Connector

Signal	*Pin*	*Pin*	*Signal*
GND	**1**	**2**	DIO0
GND	**3**	**4**	DIO1
GND	**5**	**6**	DIO2
GND	**7**	**8**	DIO3
GND	**9**	**10**	DIO4
GND	**11**	**12**	DIO5
GND	**13**	**14**	DIO6
GND	**15**	**16**	DIO7

Table 2 Digital I/O Connector Pin Assignment

Figure 4 depicts a top-view of the digital I/O connector for channels 0 through 7. The even-numbered pins are the signal pins, and the odd-numbered pins are all ground pins. The pin assignments are enumerated in Table 2. Note that digital channels 0 and 7 are labeled on the terminal board. The side of the connector that is all digital ground (odd-numberd pins) is labelled ----DGND---- on the terminal board. The other digital I/O connectors have analogous pin assignments. Standard 16-pin ribbon cable connectors may be used to connect to the digital I/O channels

Encoder Inputs

There are eight single-ended encoder inputs on the Q8 terminal board. No extra circuitry is required. Differential encoders may be connected using additonal circuitry. For example, the US Digital E5D, H5D, S5D, E6D, H6D and S6D encoders use an industry standard 26LS31 line driver IC for their differential outputs. This line driver is RS-422 compatible, so an RS-422 receiver IC may be used to convert the differential encoder signals to single-ended encoder signals for input to the Q8 terminal board. Please contact the encoder manufacturer for further information.

Encoder Connection

Single-ended encoders use three signals to supply a bidirectional count: an A channel, a B channel and an I channel, or index pulse. The index pulse is not necessary for generating encoder counts, but is convenient for calibration. The round DIN connectors on the terminal board are used for the encoders. The encoder connectors are the round connectors shown in light green in Figure 3 on page 10.

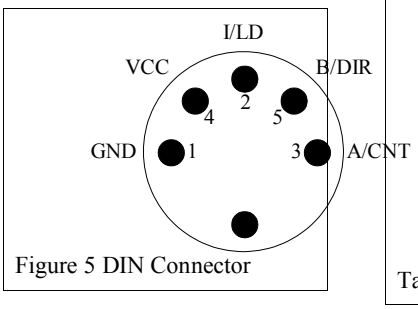

Figure 5 DIN Connector

Pin	Signal
1	GND
2	I
3	A
4	5V
5	B

Table 3 Encoder Pin Assignment

The pin assignments for the round DIN connector are illustrated in Figure 5 and on the terminal board itself. The pins are also enumerated in Table 3. There are no complementary signals for single-ended encoders. **Note that the pin numbering of the round DIN connector is not intuitive.** The pins are **not** numbered sequentially in a clockwise direction. The Q8 card allows the encoders to be used as 24-bit digital up/down counters as well. In this "non-quadrature" mode, the A input becomes a count input and the B input becomes a direction input. Every rising or falling edge of the CNT input (set in software), the 24-bit counter will count up or down, according to the DIR input. The index input in this case becomes an asynchronous load signal, which may be programmed to latch the counter value or to load the counter value from a preload register.

Analog Inputs

The Q8 terminal board supports eight single-ended analog inputs. The range of each analog input is ±10V. **Do not exceed the voltage range of the analog inputs.** While the Q8 does come with input protection circuitry that should protect it from even the worst abuses, Quanser does not guarantee proper operation of the analog inputs after the maximum voltage range has been exceeded.

Analog Input Channels 0-7

Analog input channels 0 through 7 are grouped together on the Q8 terminal board. The RCA connectors for analog inputs 0-7 are shown in light magenta in Figure 3.

Beside the RCA connectors is a group of resistors and capacitors labelled `R0` through `R7` and `C0` through `C7` respectively. The resistors and capacitors are part of the optional low-pass filtering. The Q8 terminal board is normally shipped with no capacitors installed (or capacitors with a very small value) and zero Ohm resistors.

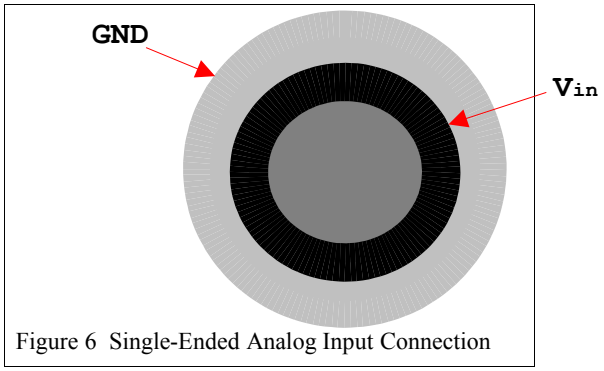

Figure 6 Single-Ended Analog Input Connection

To connect an analog input to the RCA connector, wire the RCA cable so that the outer part of the connector is connected to ground and the inner portion of the connector carries the signal you wish to measure, as shown in Figure 6.

Analog channels 1 through 7 are analogous.

Low-Pass Analog Filtering

Measured signals may be noisy and digital filtering of the converted analog signal is only effective if the analog signal is sampled fast enough to avoid aliasing. Hence, it is sometimes desirable to use a low-pass analog filter *prior* to analog-to-digital conversion. The Q8 terminal board provides this option.

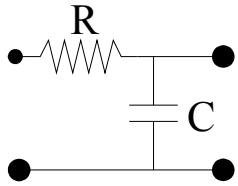

Figure 7 Analog low-pass filtering

Consider the analog input circuitry in Figure 7. The capacitor `C0` may not be provided on the terminal board, but the holes for a capacitor are provided, as is the wiring shown in the figure. Hence, to add low-pass filtering to an analog input, install a capacitor in location `C0` on the terminal board and replace the 0Ω resistor with a normal resistor. Note the resistor and capacitor must be soldered onto the terminal board. **Quanser Consulting takes no responsibility for damage to the terminal board caused by improper installation of components by the user**.

The simple resistor-capacitor circuit of Figure 7 has a transfer function of:

$$\frac{\omega_0}{s+\omega_0} \quad \text{where} \quad \omega_0 = \frac{1}{RC} \ \text{rad/s} \quad \text{and} \quad f_0 = \frac{\omega_0}{2\pi} = \frac{1}{2\pi RC} \ \text{Hz}$$

Select the resistor and capacitor values to achieve the cutoff frequency desired. For example, a cutoff frequency of 10 rad/s can be achieved using a 10KΩ resistor and a 10μF capacitor. Analog channels 1 through 7 are analogous.

Be sure to take into account the desired resolution. For example, if the normal 3dB bandwidth, as calculated above is used, the magnitude of the transfer function at the cutoff frequency is $1/\sqrt{2}$ or about 0.71. In other words, the magnitude of the signal is reduced to about 71% of its magnitude at the cutoff frequency. This magnitude reduction means that the expected 14-bit resolution is not achieved at the cutoff frequency because the filter has introduced a significant gain error at that frequency.

To achieve *n*-bit resolution over the full-scale range of the analog input, up to the cutoff frequency, the magnitude of the transfer function must be $(1 - 2^{-n})$ at the cutoff frequency. For a desired *n*-bit bandwidth of ω_n, the cutoff-frequency ω_0 of the low-pass filter must then be:

$$\omega_0 = \left(\frac{\alpha}{\sqrt{1-\alpha^2}} \right) \omega_n \quad \text{where} \quad \alpha = 1 - 2^{-n}$$

For example, to maintain 14-bit resolution at the cutoff frequency, the magnitude of the transfer function must be $(1 - 2^{-14})$ or 0.99994. This magnitude is achieved over a 100 rad/s bandwidth by using a low-pass filter with a 3dB bandwidth of 9051 rad/s!

Analog Outputs

The Q8 data acquistion card supports eight analog outputs. The analog outputs have a ±10V range. The analog outputs are available on the Q8 terminal board as RCA connectors. The outputs are grouped on the terminal board as shown in light cyan in Figure 3 on page 10.

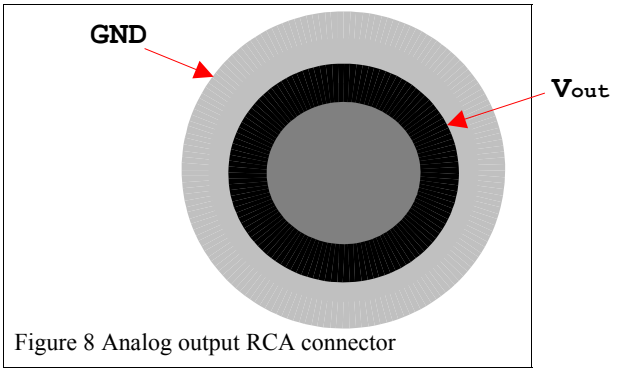

Figure 8 Analog output RCA connector

The outer contact of the RCA connector is connected to ground, while the inner contact carries the analog output signal, as depicted in Figure 8. **Be careful not to short-circuit the analog outputs.**

Special Function I/O

The Q8 data acquisition card supports 4 special function I/O lines or "control lines". These signals are available on the Control header of the Q8 terminal board. There are two outputs: CNTR_OUT and WATCHDOG, and two inputs: EXT_INT and CNTR_EN. The Control header is between the two ground lugs (AGND and DGND) in Figure 3 above, to the left of the digital I/O headers.

The 5V pin may be used to power external devices, provided the current requirements are small (eg. 10mA). **Be careful not to short-circuit the 5V line, or you will blow the fuse.** The Q8 multifunction I/O card detects failure of the fuse and zeros all analog outputs, and sets all digital outputs high, when the fuse blows. The fuse must be replaced before normal operation can be resumed.

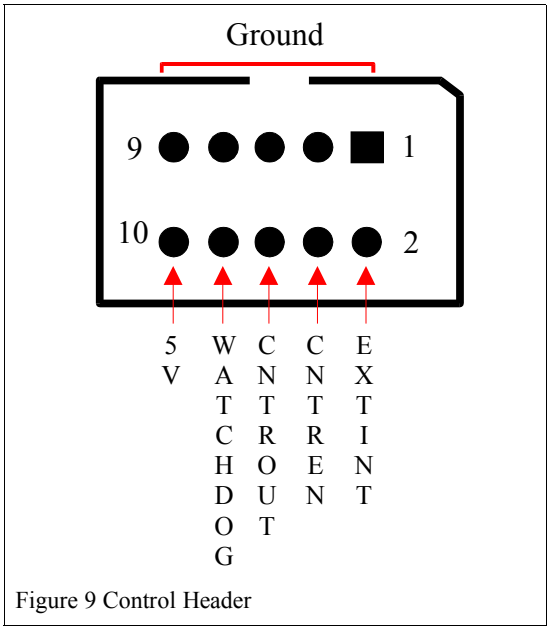

Figure 9 Control Header

Signal	Pin	Pin	Signal
GND	1	2	EXT_INT
GND	3	4	CNTR_EN
GND	5	6	CNTR_OUT
GND	7	8	WATCHDOG
GND	9	10	+5V

Table 4 Control Header Pin Assignment

Figure 9 depicts a top-view of the Control header. The even-numbered pins are the signal pins, and the odd-numbered pins are all ground pins. The pin assignments are enumerated in Table 4. Note that each line is labeled on the terminal board. The side of the connector that is all digital ground (odd-numberd pins) is labelled `----DGND----` on the terminal board. A standard 10-pin ribbon cable connector may be used to connect to the Control header.

Power

Power to the Q8 terminal board is supplied from the PC. Only the 5V power line is used. To prevent damage to the PC, the 5V line is passed through a 1A fuse before it goes to any of the circuitry on the terminal board. The power LED on the terminal board indicates whether the 5V power signal from the PC is working. Due to the protection circuitry on the terminal board and the Q8 card itself, the power at the terminal board will have a voltage level of approximately 4.7V.

Note that the fuse is sensed by the Q8 card itself. If the fuse voltage drops below about 4V, then all analog outputs will be reset to zero, and all digital outputs will be pulled high. If enabled, an interrupt will also be generated by the card to inform software of the fuse failure. The fuse status may also be polled in software. This "watchdog" capability of the fuse is an important safety feature of the Q8 data acquisition system.

Ground Lugs

There are two ground lugs provided on the Q8 terminal board for your convenience: one for analog ground and one for digital ground. They are shown in **white** in the top, left corner of Figure 3 on page 10. The ground lugs may be used for connecting the ground wire of an oscilloscope probe or other measurement devices.

Using the Q8 in Simulink

This chapter describes the Simulink blocks supplied with Quanser's WinCon and Simulinux products that are used to interface with the Q8 data acquisition system. The Simulink blocks are contained in the Quanser Toolbox library.

Launching the Quanser Toolbox

The Quanser Toolbox may be opened from the Matlab command window by typing 'qc-tools' (without the quotes) at the prompt, or by double-clicking on the Library Browser item under Quanser Toolbox in the Launch Pad (Matlab 6 and above) as shown below in Figure 14.

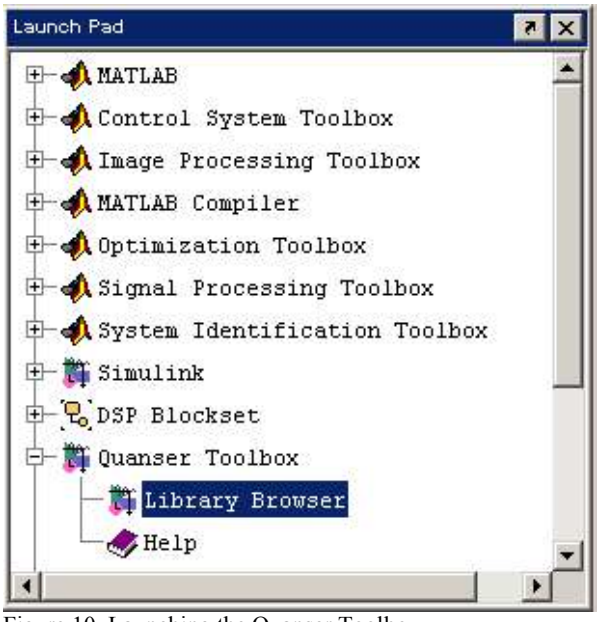

Figure 10 Launching the Quanser Toolbox

The Q8 Library

To open the block-set used to interface with the Q8, double-click on the Quanser Consulting Q8 Series block in the Quanser Toolbox. The block is depicted on the left.

Double-clicking on this block opens the window illustrated in Figure 11 below. There is a block for each of the major functional components of the

Q8 data acquisition system: digital inputs and outputs, analog inputs and outputs, encoder inputs, the watchdog timer and the hardware time-bases.

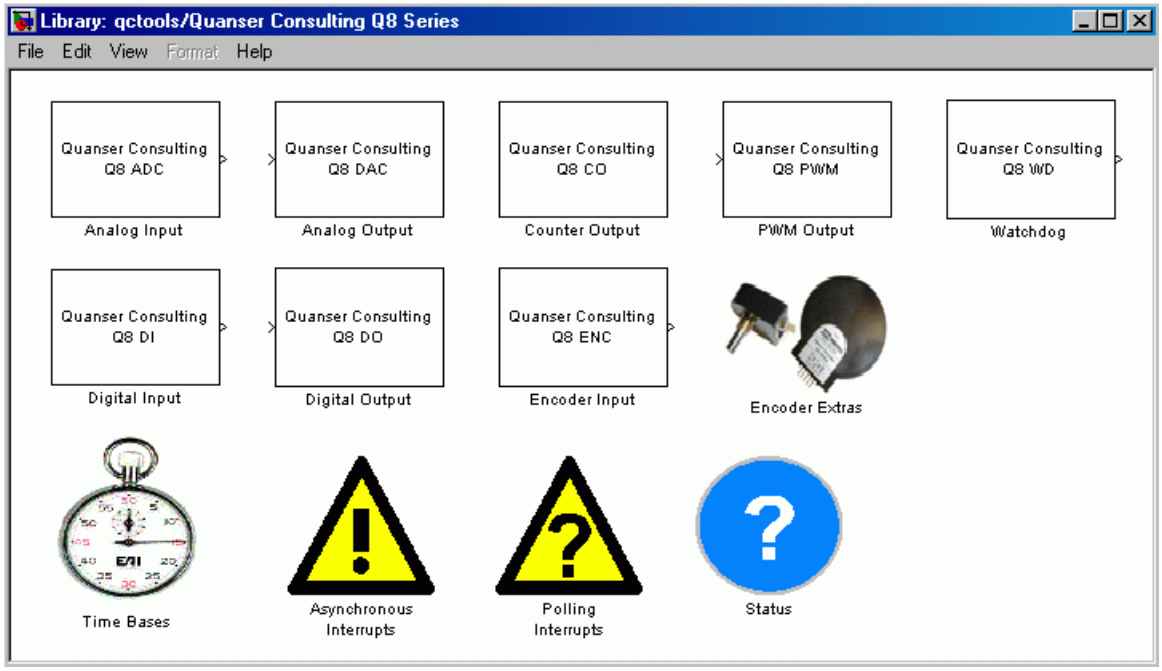

Figure 11 Q8 Library

Note that each block also comes with extensive online help for your convenience. The blocks may be used to interface with more than one Q8 card in the same Simulink diagram, if desired. The blocks are discussed in the following subsections.

Analog Input

The Analog Input block is used to interface with the analog-to-digital converters on the Q8. The Analog Input block may be used to read more than one analog input at a time. In this case, the output from the analog input block will be a vector with one entry for each analog input read. To change the analog input parameters, double-click on the block after dragging it into your Simulink diagram. The dialog box shown in Figure 15 below will appear.

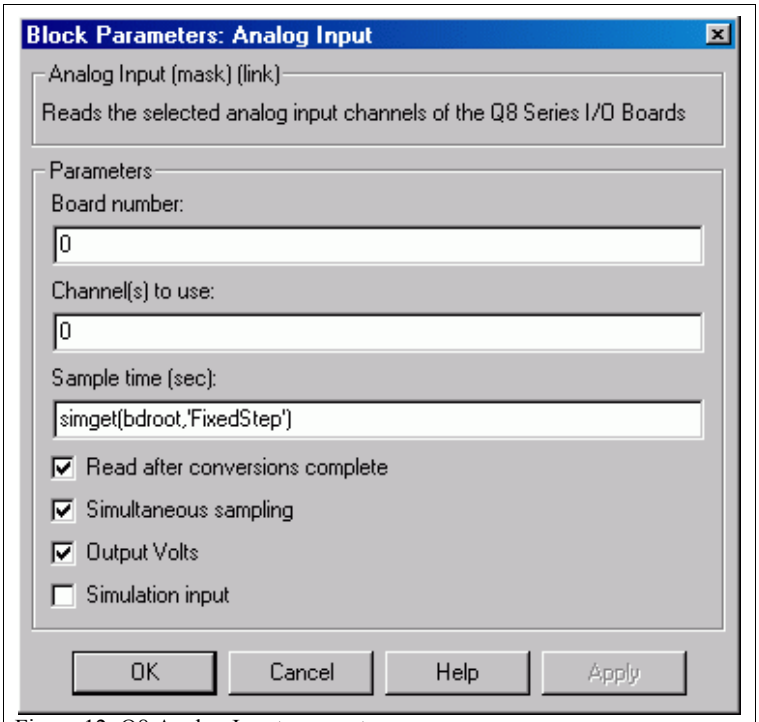

Figure 12 Q8 Analog Input parameters

The first field allows you to specify which Q8 I/O card to use. The Q8 card is a memory-mapped PCI card. The first card is referred to as board number 0 by SimuLinux. The next card is referred to as board number 1, etc. Hence, if there is more than one Q8 card in your system, simply specify the board number of the card you want. Since the correspondence between the board number and the slot in which the I/O cards are placed depends on the computer manufacturer, the easiest way to determine which card is board number 0, and which card is board number 1, etc. is to output different constant voltages on each board. Which card is which may then be determined easily using a voltmeter. All the Q8 blocks have a Board Number parameter.

The second field identifies the analog input channel, or channels, you wish to read. Specify a scalar number from 0 to 7 to read one of the eight single-ended analog input channels. To read more than one channel, specify a vector. For example, entering [5 2] will cause the Analog Input block to read analog channels 5 and 2. The output from the Analog Input block will be a vector, where the first element corresponds to analog channel 5, and the second element corresponds to analog channel number 2.

In most cases, you will probably want to specify the channels in ascending order. You can do this using the convenient ':' vector shorthand notation of MATLAB. For example, to read

analog channels 2 through 4, simply enter 2:4 into the Channel(s) to Use field.

The third parameter is the sample time. All the Q8 blocks include a sample time parameter so that they may be used in discrete-time systems. The default sample time is set to the base sampling rate of your system. You will not have to change this default unless you are developing a multi-rate system. The expression simget(bdroot,'FixedStep') returns the base sampling period as a real number, so you can multiply this expression by an integer to get a multiple of the base sampling period. Alternatively, you can simply enter a real number corresponding to the sampling period for this block. The sample time must be a positive scalar value.

The fourth parameter is 'Read after conversion complete'. Checking this option causes the Q8 driver to read the A/D conversion results from the A/D onboard FIFOs after all A/D conversions are complete. If this option is not checked, then the Q8 driver reads the conversion results as soon as they are available, not waiting the the FIFOs to fill. While it is marginally slower to read after conversions are complete, this option yields slightly better noise performance because it avoids use of the digital logic of the A/D chips while conversions are taking place.

The fifth parameter is 'Simultaneous Sampling'. When this option is checked, the two A/D chips are driven by a common external clock, guaranteeing simultaneous sampling of all channels.

The sixth parameter is 'Output Volts'. The output of the Analog Output block is in units of Volts when this option is checked. If this option is not checked, then the output is the raw 14-bit conversion result from the A/D chip. The 14-bit twos-complement conversion result is sign-extended.

The final parameter is the Simulation Input checkbox. This parameter has no effect on the real-time code. It is only used for simulation. When this box is checked, the Analog Input block will have an input port. During normal simulation, this input will be quantized according to the A/D resolution and fed to the output port. Thus, a model of the plant may be connected to this input so that the performance of the system may be simulated with quantization effects taken into account.

Analog Output

The Analog Output block is used to interface with the digital-to-analog converters on the Q8 data acquisition card. The Analog Output block may be used to write to more than one analog output at a time. In this case, the input to the Analog Output block must be a vector with one entry for each analog output written. You may put more than one Analog Output block into your diagram, if you prefer.

To change the analog output parameters, double-click on the block after dragging it into your Simulink diagram. The dialog box shown in Figure 13 below will appear. Like the Analog Input block, there is a field for the Board Number. See the Analog Input section for a discussion of this parameter.

The Channel(s) to Use field identifies the analog output channels that will be written by this block. Specify a scalar number from 0 to 3 to write to one of the four analog output channels. To write more than one channel, specify a vector. For example, entering [5 2] will cause the Analog Output block to write to analog channels 5 and 2. The input to the Analog Output block must be a vector, where the first element corresponds to analog channel 5, and the second element corresponds to analog channel number 2.

In most cases, you will probably want to specify the channels in ascending order. You can do this using the convenient ':' vector shorthand notation of MATLAB. For example, to write to analog channels 2 through 4, simply enter 2:4 into the Channel(s) to Use field.

The Mode(s) parameter sets the Mode for each output channel specified in the Channel(s) to Use parameter. If Channel(s) to Use is [3,2,0] as above, then an entry of [0,1,0] for the Mode(s) will put D/A channels 3 and 0 in bipolar mode, and D/A channel 2 in unipolar mode. A scalar value applies to all channels.

The Range(s) parameter sets the range for each output channel specified in the Channel(s) to Use parameter. If Channel(s) to Use is [3,2,0] as above, then an entry of [0,1,0] for the Mode(s) will put D/A channels 3 and 0 in bipolar mode, and D/A channel 2 in unipolar mode. A scalar value applies to all channels.

The Initial Output(s) field specifies the value(s) that will be written to the analog output channels when the real-time code is initialized, before it starts running. If you specify a scalar, then all analog output channels in the Channel(s) to Use field will be initialized to the same value. To initialize each channel to a different value, specify a vector containing the initial values for each channel.

The Final Output(s) field specifies the value(s) that will be written to the analog output channels when the real-time code is stopped. If you specify a scalar, then all analog output channels in the Channel(s) to Use field will be set to the same value. To set each channel to a different value, specify a vector containing the final values for each channel.

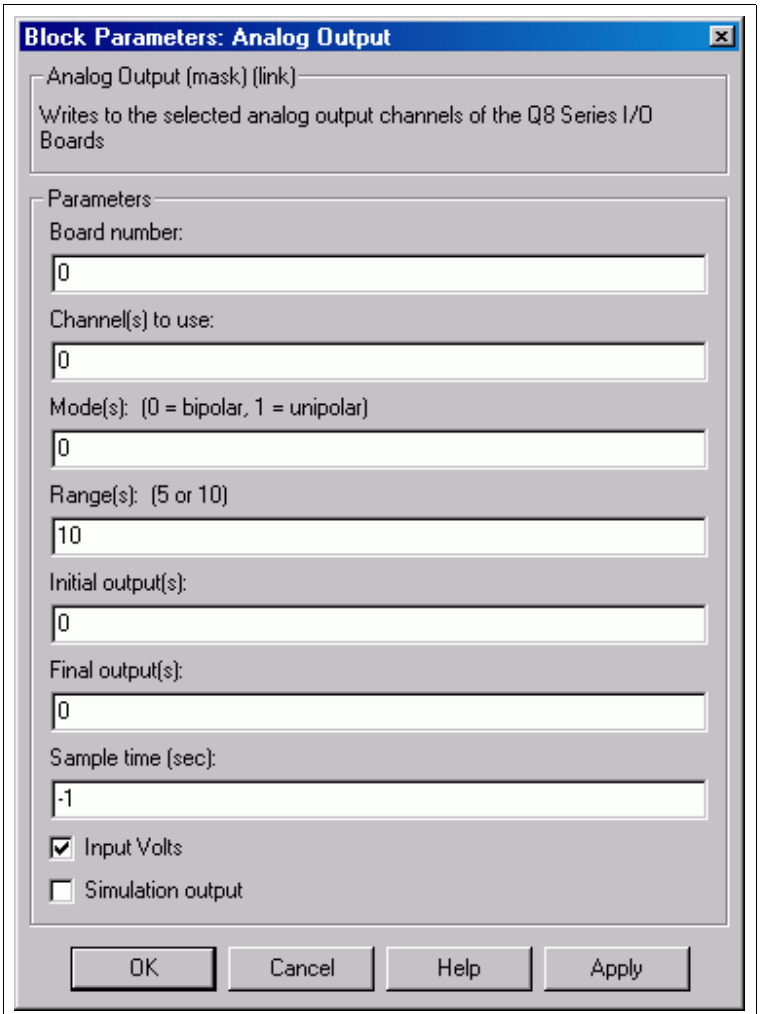

Figure 13 Q8 Analog Output parameters

All the Q8 blocks include a sample time parameter so that they may be used in discrete-time systems. The default sample time is set to -1 in the case of the Analog Output block. A sampling time of -1 indicates that the sampling rate is inherited from the input signal driving the block. Thus, you will usually not have to change this parameter, even in a multi-rate system. The sample time can also be specified as a positive scalar. The expression simget (bdroot,'FixedStep') returns the base sampling period as a real number, so you can multiply this expression by an integer to get a multiple of the base sampling period. Alternatively, you can simply enter a real number corresponding to the sampling period for this block. The sample time must be -1 or positive.

The second-last parameter is the Input Volts parameter. When this option is checked, the input signal to the block will be treated as Volts. Hence, an input value of 5 will result in 5V being produced at the output. If the option is not checked, then the input is treated as raw 12-bit D/A converter values. In unipolar mode, a value of 0 corresponds to 0V and a value of 4095 (0x0fff) corresponds to 9.998V (+10V ± 1 LSB). In 5V bipolar mode, a value of 0 corresponds to -5V and a value of 4095 corresponds to +4.998V (+5V ± 1 LSB). Similarly, in 10V bipolar mode, a value of 0 corresponds to -10V and a value of 4095 corresponds to an output voltage of +9.995V (+10V ± 1 LSB).

The Simulation Output checkbox is similar to the Simulation Input option of the Analog Input block. Checking this box puts an output port on the Analog Output block. This output port is only used during simulation. It is not used during real-time execution. During simulation, this output port produces a quantized version of the data at the input port, using the resolution of the Q8 digital-to-analog converters. Like the Simulation Input option of the Analog Input block, this feature may be used to combine a simulation of your hardware and the real-time code into a single Simulink diagram. See the discussion of the Analog Input block's Simulation Input parameter for more details.

Digital Input

The Digital Input block is used for reading the digital inputs on the Q8 card. Like the Analog Input block, there are fields for the Board Number, Sample Time and Simulation Input. See the Analog Input section for a discussion of these parameters.

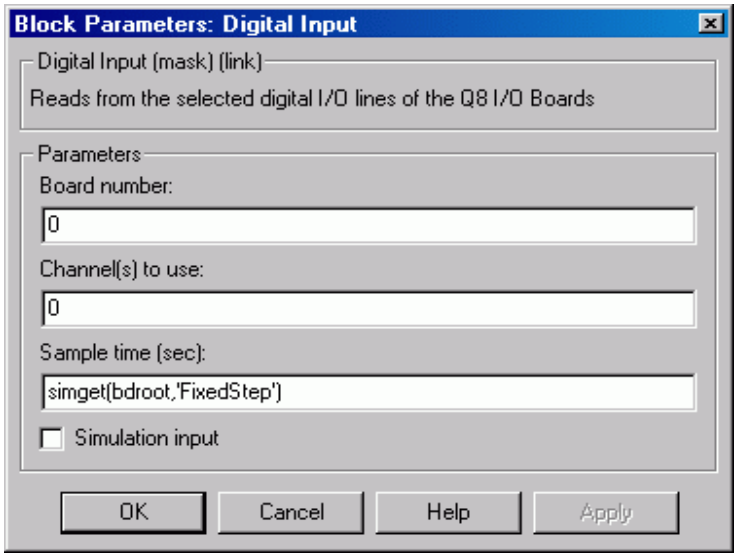

Figure 14 Q8 Digital Input parameters

The output of the Digital Input block is zero or one, not a voltage. The digital input channels to read are specified in the Channel(s) to Use parameter, much like the Analog Input block. A vector may be used to specify more than one input channel.

The Digital Input block supports up to 32 digital inputs. However, in this case, *the digital input channels are shared with the Digital Output block*. Thus, a Digital Input block and a Digital Output block cannot share the same channel numbers because the common channels would be using the same digital I/O pin. For example, if a Digital Input block was reading channel 0 then a Digital Output block could not be writing to channel 0.

Digital Output

The Digital Output block is used to write to the digital outputs of the Q8 data acquisition card. Like the Analog Output block, there are fields for the Board Number, Initial Output (s), Final Output(s), Sample Time and Simulation Input. See the Analog Output section for a discussion of these parameters. The Digital Output block takes inputs that are zero or one, not a voltage, and sends them to the digital outputs of the Q8.

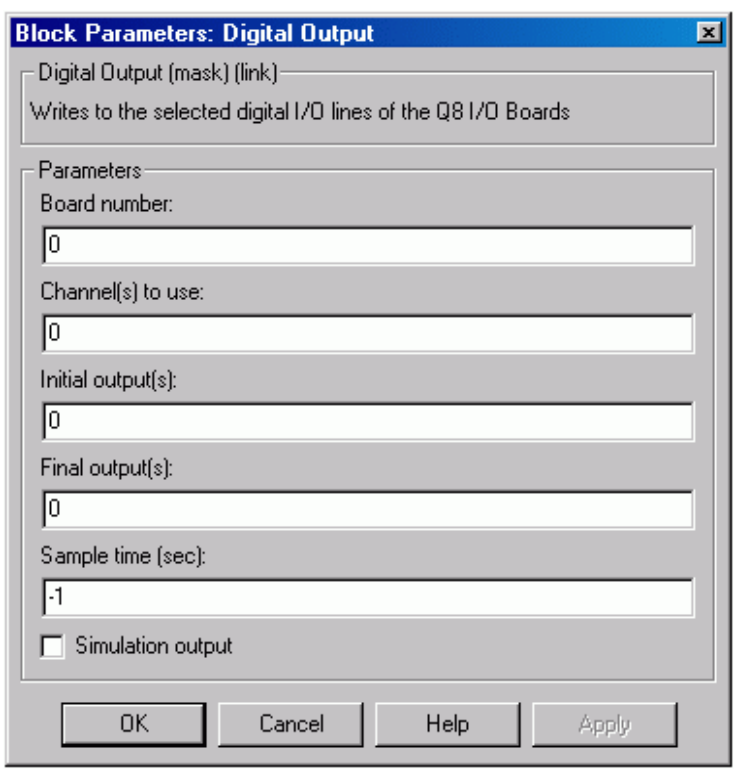

Figure 15 Q8 Digital Output parameters

The digital output channels to write are specified in the Channel(s) to Use parameter, much like the Analog Output block. A vector may be used to specify more than one output channel.

The Digital Output block supports up to 32 digital outputs. However, in this case, *the digital output channels are shared with the Digital Input block*. Thus, a Digital Output block and a Digital Input block cannot share the same channel numbers because the common channels would be using the same digital I/O pin. For example, if a Digital Output block was writing to channel 0 then a Digital Input block could not be reading from channel 0.

Encoder Input

The Encoder Input block is used to read the counter values of the Q8 encoder inputs. Up to eight encoder inputs are supported, number 0 through 7. The output of the encoder block is the 24-bit encoder count(s). More than one encoder channel may be read at the same time and more than one Encoder Input block may be present in your Simulink diagram. The parameters of the Encoder Input block are illustrated in Figure 10. Like the Analog Input block, there are fields for the Board Number, and Sample Time. See the Analog Input section for a discussion of these parameters.

The Channel(s) to Use parameter specifies which encoder channels are to be read. More than one encoder channel may be specified by entering a vector for this parameter.

The Initial Value(s) parameter is used to preload the encoder counts when the model starts. For example, setting this parameter to 0 will cause all encoders to read a value of 0 initially. The initial value for each encoder channel may be specified separately by entering a vector for this parameter. For example, if the Channel(s) to Use parameter is [2, 4] and the Initial Value(s) parameter is [32768, -8192] then encoder channel 2 will have an initial value of 32768 and channel 4 will have an initial value of -8192 when the model is run.

The final parameter, the Simulation Input checkbox, puts an input port on the Encoder Input block when it is selected. This input port may be used much like the Simulation Input on the Analog Input block. However, in this case, the input is a count value that is fed to the output (during simulation only) after restricting it to a 24-bit value using the modulus operator. The simulation input is ignored in real-time code.

For a complete description of all the other Encoder Input parameters, please refer to the Help section of the Encoder Input block under Simulink.

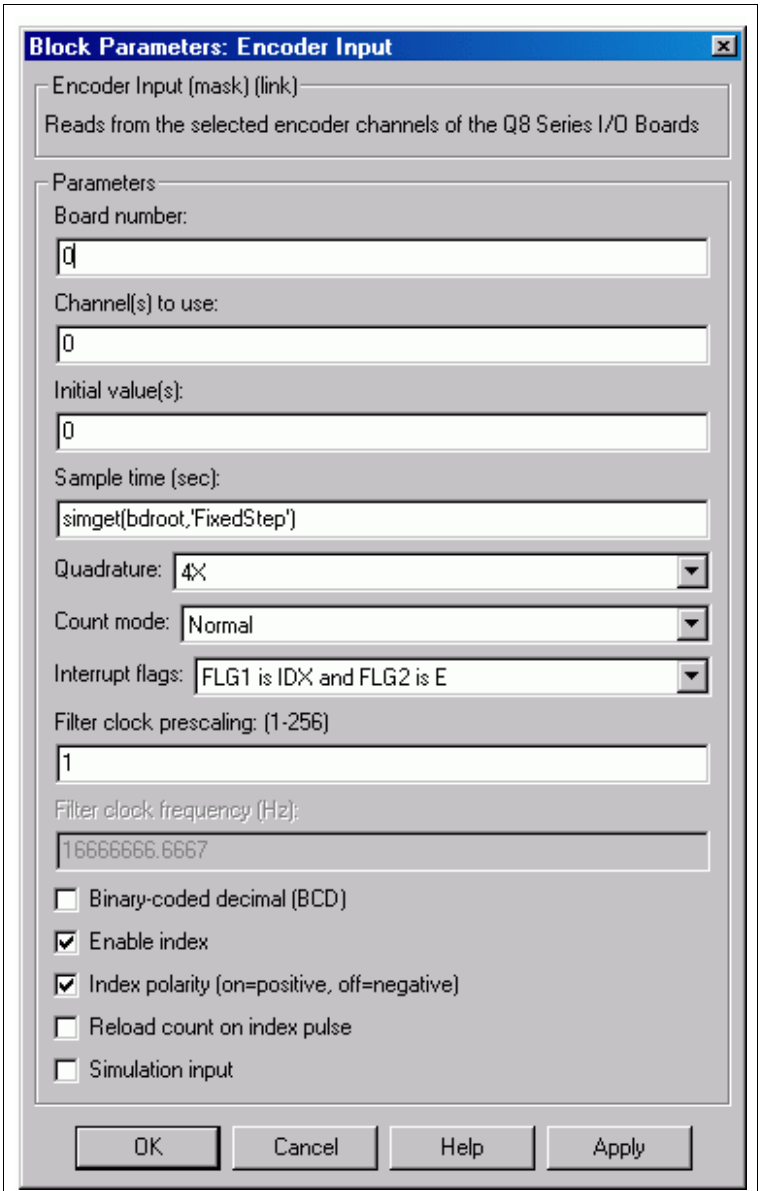

Figure 16 Q8 Encoder Input parameters

Counter Output

The Counter Output Module for the Q8 series of I/O cards programs one of the 32-bit general purpose counters to output a PWM signal on an external pin. The polarity of the CNTR_EN input may also be configured for hardware enabling/disabling of the counter.

The CNTR_EN line only affects the COUNTER clock source. Like the Analog Input block, there are fields for the Board Number, and Sample Time. See the Analog Input section for a discussion of these parameters.

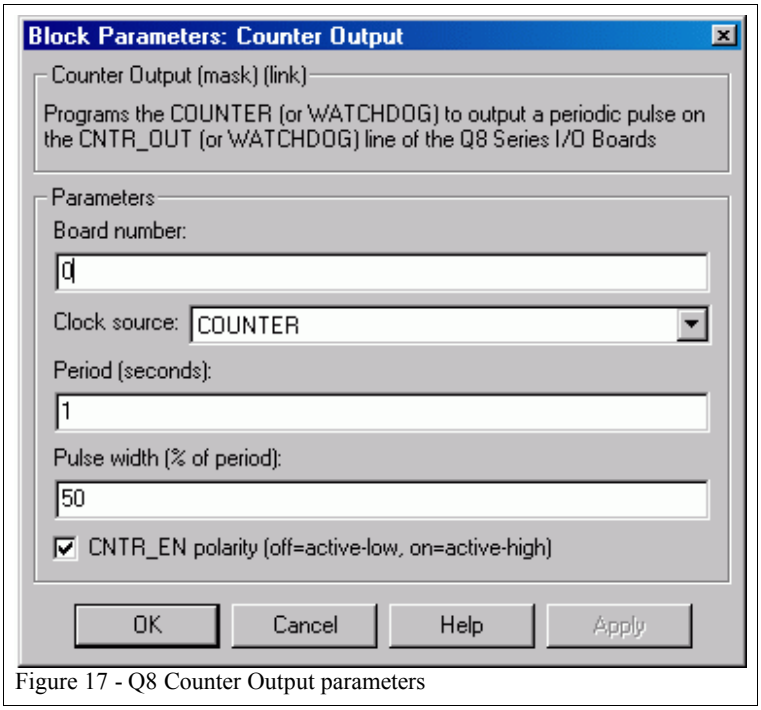

Figure 17 - Q8 Counter Output parameters

The Clock source parameter sets which of the 2 32-bit counters to use. If COUNTER is selected, then the PWM signal will appear on the CNTR_OUT pin of the CONTROL header on the terminal board. If WATCHDOG is selected, then the PWM signal will appear on the WATCHDOG pin of the CONTROL header on the terminal board.

The Period parameter sets the period in seconds of the output signal. The period may range from 60 ns to 257 seconds (4.3 minutes), in 60 ns units.

The Pulse width parameter sets the width of the high pulse as a percentage of the period. The pulse width may range from 0 to 100.

The CNTR_EN polarity checkbox sets the polarity of the CNTR_EN input. If this option is not checked then the CNTR_EN is an active-low input. Otherwise it is active-high. The CNTR_EN line has a pull up resistor to bring it high when the input is unconnected. Hence, this option is normally checked to enable the counter by default.

PWM Output

The PWM Output Module for the Q8 series of I/O cards programs one of the 32-bit general purpose counters to output a PWM signal on an external pin. The polarity of the CNTR_EN input may also be configured for hardware enabling/disabling of the counter. The CNTR_EN line only affects the COUNTER clock source. Unlike the Counter Output block, the PWM Output block reprograms the duty cycle every sampling instant, according to its input. Note that when the current duty cycle is non-zero, the new duty cycle is activated at the end of the next PWM period. Hence there are no spurious output values during transitions. Like the Analog Input block, there is a field for the Board Number. See the Analog Input section for a discussion of these parameters.

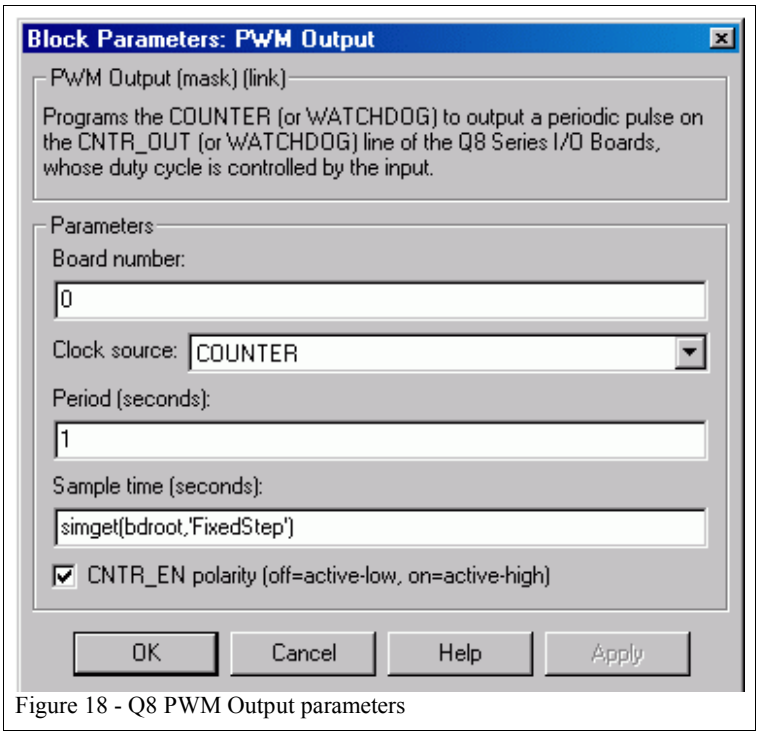

Figure 18 - Q8 PWM Output parameters

The Clock source parameter sets which of the 2 32-bit counters to use. If COUNTER is selected, then the PWM signal will appear on the CNTR_OUT pin of the CONTROL header on the terminal board. If WATCHDOG is selected, then the PWM signal will appear on the WATCHDOG pin of the CONTROL header on the terminal board.

Watchdog

The Watchdog Module for the Q8 series of I/O cards provides a mechanism for using the watchdog timer on an installed Q8 I/O card. If the watchdog expires, then the analog outputs are reset to 0V and the digital outputs are reconfigured as digital inputs and pulled high. Hence, all digital outputs go high on watchdog expiration. The output of the block goes from zero to one at the next sampling instant. To stop the model on watchdog expiration, connect a Stop Simulation or Stop With Error block to the output of the Watchdog module. Like the Analog Input block, there are fields for the Board Number, and Sample Time. See the Analog Input section for a discussion of these parameters.

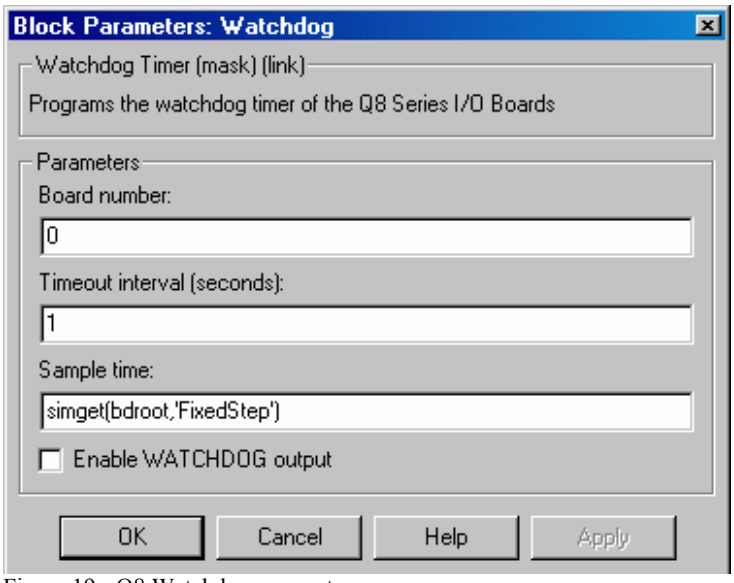

Figure 19 - Q8 Watchdog parameters

The watchdog timer is a 32-bit counter. It supports any timeout period from 60 ns to 257.69803776 seconds (4.3 minutes) in 60 ns units. The period is generally programmed to be more than the sampling period of the model, but small enough that the model stops quickly after the watchdog expires. The output of the block changes at the next sampling instant after the watchdog timer expires. The Watchdog Interrupt block may also be used to act on watchdog expiration. The Watchdog Interrupt block is an asynchronous block that is executed immediately upon expiration of the watchdog timer, independent of the sampling period of the model.

Checking the Enable WATCHDOG output checkbox enables output of the watchdog state to the WATCHDOG pin on the CONTROL header of the Q8 terminal board. The output remains high as long as the watchdog timer has not expired. If the watchdog expires then the

output goes low. This signal can be used to shut off external hardware, such as current amplifiers for robot motors.

It can also be fed to the EXT_INT pin of other Q8 cards (available on the CONTROL header of the Q8 terminal board) in order to propagate the watchdog functionality to the other cards. Use one of the External Interrupt, External Status or Poll External Interrupt blocks with the other cards and check the "EXT_INT acts as watchdog" option. When EXT_INT is configured as a watchdog, a falling-edge on the EXT_INT line causes all the analog outputs of the card to be reset to zero and all the digital I/O lines to be pulled high, just like a watchdog timer. The EXT_INT line can also be programmed to act upon a rising edge.

The Quanser Q8 blocks described above are the standard and most commonly used blocks in interfacing to the Q8. They allow the user to read and write to all the I/O channels available on the Q8 board. The Quanser Q8 board is a highly versatile and powerful multifunction I/O board. The following section will briefly describe all the additional simulink blocks available to use with the Q8. For a more thorough explanation of these blocks and their parameters, please refer to the online help that is available for each block through the Simulink environment.

Additional Q8 Blocks

The following blocks will be listed with a brief description. For a complete description of each block, refer to the online help available through Simulink.

Encoder Extras

It is often convenient to calibrate the encoders in the real-time code in response to some event, such as a limit switch being hit or at a certain time. The Encoder Extras library provides a set of Encoder Reset blocks for performing such calibration. One block resets the encoder value based on a level-sensitive input. Two others reset the encoder value based on an edge-triggered input. Since the Encoder Input block sets the initial value of the encoder count, the Encoder Reset blocks are not necessary to set the initial count value. However, they are very useful for calibration.

Encoder Reset (level-sensitive / edge-triggered / edge-triggered with enable): Resets the count value of an encoder in the Q8 data acquisition system.

Configure Encoder I/O: Configures the encoder flags and whether to reload the 24-bit encoder counter on an index pulse.

Read Encoder Flags: Reads the encoder flags for the specified channels. These flags are not the interrupt flags but the contents of the encoder status register.

Measure Average Period: Measures the average period of a digital signal applied to the CNT input of one of the encoder DIN connectors.

Measure Computation Time: Measures the computation time for the given sampling rate in terms of the signal applied to the CNT input of one of the encoder DIN connectors.

Time Bases

The Time Base blocks for the Q8 series of I/O cards provides a mechanism for using one of the hardware counters on an installed Q8 card as a time-base for the real-time code. This block is used in Windows to achieve sampling rates faster than 10kHz or to synchronize multiple computers.

The Time Base block allows a hardware timer to be used to generate the sampling rate for the system. When no Time Base block is present in the diagram, the system timer is used. The system timer under Windows NT/2000 is fast and accurate, supporting sub-millisecond sampling rates, so this block is not usually necessary, but it is available for those users wanting to use a hardware time-base on their data acquisition card.

Counter Time Base: Establishes a hardware time-base for the model using one of the two 32-bit general purpose counters on the Q8 data acquisition system.

External Time Base: Establishes a hardware time-base for the model using the EXT_INT line of the Q8 data acquisition system.

Analog Input Time Base: Establishes a hardware time-base for the model by automatically triggering A/D conversions using the COUNTER of the Q8 data acquisition system.

Analog Input External Time Base: Establishes a hardware time-base for the model by automatically triggering A/D conversions using the external CNTR_EN line of the Q8 data acquisition system.

Asynchronous Interrupts

The Interrupt blocks for the Q8 series of I/O cards provides a mechanism for handling asynchronous interrupts. Attaching a function-call subsystem to an output causes the function-call subsystem to be executed as soon as the associated interrupt occurs. The code executes asynchronously from the rest of the model. The timing of its execution is not governed by the sampling period of the system.

Counter Interrupt: Asynchronously invokes the function-call subsystem when a counter interrupt occurs on the Q8 data acquisition system.

Watchdog Interrupt: Asynchronously invokes the function-call subsystem when a watchdog interrupt occurs on the Q8 data acquisition system.

External Interrupt: Asynchronously invokes the function-call subsystem when an external interrupt occurs on the Q8 data acquisition system.

Counter Interrupt: Asynchronously invokes the function-call subsystem when a counter interrupt occurs on the Q8 data acquisition system.

Analog Interrupts: Asynchronously invokes the function-call subsystem when an analog interrupt occurs on the Q8 data acquisition system. The available interrupts are *end of conversion* and *ready*.

Fuse Interrupt: Asynchronously invokes the function-call subsystem when a fuse interrupt occurs on the Q8 data acquisition system.

Encoder Interrupts: Asynchronously invokes the function-call subsystem when an encoder interrupt occurs on the Q8 data acquisition system. The Interrupts get triggered when either of the 2 encoder flags go high. The flags can be configured when using an encoder input block as described above.

Polling Interrupts

The Poll Interrupts block for the Q8 series of I/O cards provides a mechanism for polling any of the interrupt sources of the card. It reads the Interrupt Status Register of the Q8 card. Connecting an output to another block causes that interrupt source to be polled. The output goes high when the associated interrupt occurs. The block has an option for resetting the interrupt flag after an interrupt occurs. The block polls the Interrupt Status Register each sampling instant.

Poll Counter Interrupt: Polls the counter interrupt of the Q8 data acquisition system.

Poll Watchdog Interrupt: Polls the watchdog interrupt of the Q8 data acquisition system.

Poll External Interrupt: Polls the external interrupt of the Q8 data acquisition system.

Poll Counter Interrupt: Polls the counter interrupt of the Q8 data acquisition system.

Poll Analog Interrupts: Polls the analog interrupts of the Q8 data acquisition system. The available interrupts are *end of conversion* and *ready*.

Poll Fuse Interrupt: Polls the fuse interrupt of the Q8 data acquisition system.

Poll Encoder Interrupts: Polls the encoder flag interrupts of the Q8 data acquisition system. The flags can be configured when using an encoder input block as described above.

Status

The Status blocks for the Q8 series of I/O cards provides a mechanism for checking the status flags of the card. It reads the Status Register of the Q8 card. Each output reflects the status of that particular flag. The output is high when the status flag is high and low otherwise.

The block queries the Status Register each sampling instant.

Counter Enable Status: Returns the status of the CNTR_EN input of the Q8 data acquisition system.

External Status: Returns the status of the EXT_INT input of the Q8 data acquisition system.

Fuse Status: Returns the status of the fuse on the Q8 terminal board.

Analog Status: Returns the status of the A/D converters on the Q8 terminal board.

Encoder Status: Returns the status of the encoders on the Q8 terminal board.

Q8 Registers

This chapter describes the registers on the Q8 data acquisition card. These registers can only be programmed from a device driver or a real-time kernel with PCI hardware support. The registers cannot be programmed from a Win32 application directly. Refer to the Q8 Programming chapter for information on how to program the Q8 data acquisition system using the drivers provided. This chapter is only included for those users who wish to write their own device drivers for operating systems or real-time kernels that are not currently supported by Quanser.

Card Identification

The Q8 card may be identified in the plug-and-play subsystem by its vendor ID, device ID, subvendor ID and subdevice ID, as listed in Table 5 below.

Identifier	*Value*
Vendor ID	0x11E3
Device ID	0x0010
Subvendor ID	0x5155
Subdevice ID	0x0200 (identifies Rev. 2. card, full featured)

Table 5 Card identification values

Register Map

The Q8 card is fully plug-and-play compatible. Hence, the operating system automatically assigns a base address and interrupt vector to the card during boot up. The system BIOS will do so for operating systems that are not plug-and-play capable. All registers on the Q8 are relative to this system-assigned base address. The offset of each register from this base address is given in Table 6 below. Byte offsets are given in hexadecimal. Each register is described in detail in subsequent sections.

All registers may be read or written using 32 or 64 bit accesses.

Certain registers may be read or written using 8 or 16 bit accesses as well. Refer to the corresponding section for more information.

The Q8 supports burst cycles on the PCI bus.

Hence, optimal transfer rates are achieved by accessing consecutive addresses, which may permit the PCI bridge to combine bus cycles into a burst cycle.

For example, it may be more efficient to read the encoder chips in the following sequence:

```
int evenChannels1 = pQ8->encoderData.four.enc0246;
int oddChannels1  = pQ8->encoderData.four.enc1357;
int evenChannels2 = pQ8->encoderData.four.enc0246;
int oddChannels2  = pQ8->encoderData.four.enc1357;
int evenChannels3 = pQ8->encoderData.four.enc0246;
int oddChannels3  = pQ8->encoderData.four.enc1357;
```

than it is to read in the order:

```
int evenChannels1 = pQ8->encoderData.four.enc0246;
int evenChannels2 = pQ8->encoderData.four.enc0246;
int evenChannels3 = pQ8->encoderData.four.enc0246;
int oddChannels1  = pQ8->encoderData.four.enc1357;
int oddChannels2  = pQ8->encoderData.four.enc1357;
int oddChannels3  = pQ8->encoderData.four.enc1357;
```

Byte Offset	*Register*		*Read/Write*	
0x00	Interrupt Enable Register		RW	
0x04	Interrupt Status Register		RW	
0x08	Control Register		RW	
0x0c	Status Register		R	
0x10	Counter Preload Register	Counter Preload Low Register	R	W
0x14	Counter Register	Counter Preload High Register	R	W
0x18	Watchdog Preload Register	Watchdog Preload Low Register	R	W
0x1c	Watchdog Register	Watchdog Preload High Register	R	W
0x20	Counter Control Register		RW	
0x24	Digital I/O Register		RW	
0x28	Digital Direction Register		W	
0x2c	A/D Register		RW	
0x30	Encoder Data Register A (enc0246)		RW	
0x34	Encoder Data Register B (enc1357)		RW	
0x38	Encoder Control Register A (enc0246)		RW	
0x3c	Encoder Control Register B (enc1357)		RW	
0x40	D/A Output Register A (dac04)		RW	
0x44	D/A Output Register B (dac15)		RW	

Byte Offset	*Register*	*Read/Write*
0x48	D/A Output Register C (dac26)	RW
0x4c	D/A Output Register D (dac37)	RW
0x50	D/A Update Register	W
0x54-0x68	Reserved	-
0x6c	D/A Mode Register	RW
0x70	D/A Mode Update Register	W
0x74-0x3ff	Reserved	-

Table 6 Register Map

Interrupt Sources

The Q8 provides a single interrupt line. However, interrupts may be generated from a variety of sources. For example, interrupts may be generated when an encoder index pulse is encountered or an A/D conversion is complete.

There are a total of 55 possible interrupt sources in the Q8 data acquisition system.

Interrupts may be individually enabled or disabled. Interrupts are controlled via the Interrupt Enable Register and the Interrupt Status Register. Both of these registers have the same layout, with the exception of bit 31. Table 7 below enumerates the different interrupt sources and the corresponding bit in the Interrupt Enable and Interrupt Status Registers.

Interrupts are generated on the rising edge of the interrupt source, if interrupts are enabled from that source. The Interrupt Status Register may be used by an interrupt handler to determine the source of the interrupt. The corresponding bit in the Interrupt Status Register must be cleared by the interrupt handler. Refer to the discussion of the Interrupt Enable and Interrupt Status Registers for more details on the use of interrupts and the Q8.

Bit	*Name*	*Interrupt Source*
31→24	Reserved	Currently unused. Set to zero. Bit 31 used in Interrupt Status.
23	EXT_INT	External interrupt signal
22	FUSE	Detect blown fuse on terminal board or detached cable
21	WATCHDOG	Expiration of watchdog timer
20	CNTR_OUT	Expiration of timer
19	ADC47_RDY	Ready signal for A/D channels 4-7
18	ADC03_RDY	Ready signal for A/D channels 0-3
17	ADC47_EOC	End-of-conversion output for A/D channels 4-7
16	ADC03_EOC	End-of-conversion output for A/D channels 0-3
15	ENC8_FLG2	FLG2 output for encoder channel 7
14	ENC7_FLG1	FLG1 output for encoder channel 7
13	ENC6_FLG2	FLG2 output for encoder channel 6
12	ENC6_FLG1	FLG1 output for encoder channel 6
11	ENC5_FLG2	FLG2 output for encoder channel 5
10	ENC5_FLG1	FLG1 output for encoder channel 5
9	ENC4_FLG2	FLG2 output for encoder channel 4
8	ENC4_FLG1	FLG1 output for encoder channel 4
7	ENC3_FLG2	FLG2 output for encoder channel 3
6	ENC3_FLG1	FLG1 output for encoder channel 3
5	ENC2_FLG2	FLG2 output for encoder channel 2
4	ENC2_FLG1	FLG1 output for encoder channel 2
3	ENC1_FLG2	FLG2 output for encoder channel 1
2	ENC1_FLG1	FLG1 output for encoder channel 1
1	ENC0_FLG2	FLG2 output for encoder channel 0
0	ENC0_FLG1	FLG1 output for encoder channel 0

Table 7 Interrupt Sources

The FLG*n* outputs for the encoder channels are outputs that may be configured via the Encoder Control Registers. For example, FLG1 may be configured to go high when an index

pulse occurs on the corresponding encoder channel. There are six possible interrupt sources per encoder channel, yielding a total of 55 possible interrupt sources! More information may be found concerning the encoder flags in the discussion of the Encoder Control Registers.

The end-of-conversion flags are asserted each time an A/D conversion has completed. The ready signals are asserted after all the selected channels in the group are converted. For example, if channels 0-3 have been selected for conversion, then ADC03_RDY is asserted after all four channels have been converted. Refer to the section concerning the A/D Register for more information about the end-of-conversion and ready flags.

The external interrupt signal is a special digital input available on the Q8 terminal board. It is provided in addition to the 32 general purpose digital I/O lines. The external interrupt signal may be used to trigger an interrupt. It is often used to cause an interrupt when a limit switch has been encountered or an emergency stop button has been pressed. The EXT_INT signal is actually the inversion of the external interrupt input. Hence, a falling edge on the external interrupt input causes an interrupt to occur. This inversion allows a wired-OR connection to be made to the external interrupt input, simplifying external circuitry when multiple external interrupt sources are desired. The general purpose digital I/O lines may be used to determine the external interrupt source, if desired.

The pulse on the external interrupt line must be at least 40ns wide in order to generate an interrupt.

An interrupt may also be generated when either the general purpose timer or watchdog timer expire. The watchdog timer interrupt is typically used to allow software to recover gracefully when hard timing constraints are not met by the software, or worse, the software crashes. The general purpose timer interrupt is often employed as an accurate time base that is independent of the PC clock and has higher resolution.

The FUSE interrupt is used to detect one of two situations: either the J1 cable is not connected or the fuse on the terminal board has been blown by improper connection of the encoder inputs.

The FUSE interrupt is of critical importance for safety reasons because the terminal board supplies the +5V power to the encoders. This interrupt may be used to prevent control systems from going unstable due to a blown fuse.

Consider a simple PID closed-loop position-control system that uses an encoder input to read motor position and the analog outputs to drive the motor. The controller drives the motor to the desired position by applying a voltage to the motor. However, if the fuse is blown by improper connection of an encoder input, or malfunction of the encoder, then the encoder counts will not change, even though the motor shaft is moving. If the encoder reading doesn't change then the controller will continue to increase the drive voltage due to the inte-

grator term. Hence, when the fuse is blown, the physical system will move faster and faster because the controller cannot detect this movement and continues to increase the drive voltage. Serious damage to the motor and other hardware can result. Hence, the Q8 allows the fuse to be monitored. The FUSE signal is low as long as the fuse is operating correctly. The FUSE input goes high as soon as the fuse is blown.

Furthermore, the Q8 resets the analog outputs and sets all digital outputs when this condition is detected. No software intervention is required.

Software can also react by interrupting on the FUSE interrupt, or polling the FUSE bit of the Status Register.

Interrupt Enable Register

Byte Offset	0x00
Read/Write	read or write
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	cleared

The Interrupt Enable Register is a 32-bit register. It is used to enable or disable interrupts from various sources. The contents of this register are shown in Table 8 below. The register is cleared when the PCI bus is reset. Hence, interrupts are disabled on boot up.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
								EXTINT	FUSE	WATCHDOG	CNTROUT	ADC47RDY	ADC03RDY	ADC47EOC	ADC03EOC	ENC7FLG2	ENC7FLG1	ENC6FLG2	ENC6FLG1	ENC5FLG2	ENC5FLG1	ENC4FLG2	ENC4FLG1	ENC3FLG2	ENC3FLG1	ENC2FLG2	ENC2FLG1	ENC1FLG2	ENC1FLG1	ENC0FLG2	ENC0FLG1

Table 8 Interrupt Enable Register

Each bit of the Interrupt Enable Register enables or disables the interrupt source:

```
0 = disable interrupt
1 = enable interrupt
```

Write a '1' to a bit to enable interrupts from the corresponding interrupt source. Write a '0' to disable interrupts from that source. The bits refer to the interrupt sources listed in Table 7 of section Interrupt Sources. To disable all interrupts from the Q8, write the value zero to this register. When enabling interrupts, only enable the interrupts that are required.

Reading from the Interrupt Enable Register returns the contents of the register. Reserved bits 24-31, shaded grey, will always return the value zero, regardless what value was written.

For example, in order to enable interrupts when A/D conversions are complete on any A/D channel, and disable interrupts from all other sources, write the value 0x000C0000 to the Interrupt Enable Register, which sets bits 18 and 19.

Note that the Interrupt Status Register should be cleared before enabling interrupts. Otherwise an interrupt may occur immediately due to status bits that had not been cleared.

See the discussion of the Interrupt Status Register for more information on clearing interrupt flags.

```
pQ8->interruptStatus = 0xffffffff; /* Clear all interrupts  */
pQ8->interruptEnable = 0x000c0000; /* Enable A/D interrupts */
```

Interrupt Status Register

Byte Offset	0x04
Read/Write	read or write
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	cleared

The Interrupt Status Register identifies which interrupt source(s) caused an interrupt. The contents of this register are shown in Table 9 below. The register is cleared during reset. Hence, after booting the computer the Interrupt Status Register will indicate that no interrupts occurred.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
INT-PEND								EXTINT	FUSE	WATCHDOG	CNTROUT	ADC47RDY	ADC03RDY	ADC47EOC	ADC03EOC	ENC7FLG2	ENC7FLG1	ENC6FLG2	ENC6FLG1	ENC5FLG2	ENC5FLG1	ENC4FLG2	ENC4FLG1	ENC3FLG2	ENC3FLG1	ENC2FLG2	ENC2FLG1	ENC1FLG2	ENC1FLG1	ENC0FLG2	ENC0FLG1

Table 9 Interrupt Status Register

Each bit may be interpreted as follows on a read:

```
0 = did not cause an interrupt
1 = did cause an interrupt
```

A bit will be '1' if the corresponding interrupt source caused the interrupt. Otherwise the bit will be '0'.

Note that an interrupt may be caused by multiple interrupt sources simultaneously. In this case, more than one bit will be set.

If the value of this register is zero then the Q8 card did not cause the interrupt.

Bits in the Interrupt Status Register are set by a *rising edge* of the interrupt source signal. The bits remain set until they are explicitly cleared. To clear a bit in the Interrupt Status Register, write a '1' to the bit. Writing a '0' to a bit has no effect on the bit – the bit will remain unchanged:

```
0 = leave status bit unchanged
1 = clear status bit
```

For example, if the counter has expired and caused an interrupt then bit 20 will be set. To clear this bit, write a '1' to bit 20:

```
if (pQ8->interruptStatus & 0x00100000) /* counter interrupt */
{
    pQ8->interruptStatus = 0x00100000; /* clear counter flag */
    ....
}
```

This logic is used so that the same bitmask can be used to check the source of the interrupt, to enable the interrupt and to clear the interrupt flag.

To generate an interrupt, the hardware logically ANDs the Interrupt Status Register with the Interrupt Enable Register and produces an interrupt whenever the result is non-zero. Hence, the bit(s) which caused an interrupt must be cleared before returning from the interrupt handler to prevent the same interrupt from recurring immediately upon exit from the interrupt handler.

Note that the bits in the Interrupt Status Register will be set by the interrupt sources regardless of whether the interrupt is enabled in the Interrupt Enable Register.

Thus, the Interrupt Status Register can also be used for polling the status of the interrupt sources. Once a bit is set by an interrupt source, the bit must be explicitly cleared by software before the interrupt source can set the bit again. To poll the interrupt sources without having to clear the Interrupt Status Register, use the Status Register instead.

The upper-most bit, the INT_PEND bit, indicates whether the interrupt was caused by *any* of the interrupt sources. Unlike the other bits in this register, it is set *only when interrupts are enabled and one of the interrupt sources caused the interrupt.* Hence, it provides a quick way to determine whether the Q8 card caused the interrupt. Also note that this bit cannot be cleared by writing a '1' to the upper-most bit. It is only cleared when the interrupt status bits for those interrupts that are enabled have been cleared.

The WATCHDOG bit is inverted and sent to the WATCHDOG output of the card, if configured to do so in the Counter Control Register.

Also, if the watchdog reset feature is activated, the analog outputs will be reset and digital outputs pulled high as long as this bit is set.

Control Register

Byte Offset	0x08
Read/Write	read or write
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	cleared

The Control Register controls a number of features of the card. For example, it configures the way A/D conversions are performed and whether the encoder index pulses are used. The contents of this register are shown in Table 10 below. The register is cleared during reset. Hence, after booting the computer, it will be have a known default configuration.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
			CTENCV	EXTACT	EXTPOL	DAC47TR	DAC03TR	ADC47CV	ADCSTBY	ADC47CT	ADC47HS	ADCSL7	ADCSL6	ADCSL5	ADCSL4	ADC03CV		ADC03CT	ADC03HS	ADCSL3	ADCSL2	ADCSL1	ADCSL0	ENC7IDX	ENC6IDX	ENC5IDX	ENC4IDX	ENC3IDX	ENC2IDX	ENC1IDX	ENC0IDX

Table 10 Control Register

The bits of the Control Register are described in Table 11 below. When the Control Register is read, the contents of the register are returned, with the exception of the shaded bits and the ADC03_CV and ADC47_CV bits (bits 15 and 23 respectively). These bits always return '0' on a read since they do not reflect the configuration of the card.

Bit	Name	Interpretation
31→29	Reserved	Currently unused. Set to zero.
28	CTEN_CV	0 = Use Counter to trigger automatic A/D conversions
		1 = Use CNTR_EN line to trigger automatic A/D conversions
27	EXT_ACT	0 = Deactivate the watchdog features of EXT_INT line
		1= Activate the watchdog features of the EXT_INT line
26	EXT_POL	0 = EXT_INT active low, 1 = EXT_INT active high
25	DAC47_TR	0 = Disable transparent mode for D/A channels 4-7
		1 = Ensable transparent mode for D/A channels 4-7
24	DAC03_TR	0 = Disable transparent mode for D/A channels 0-3
		1 = Enable transparent mode for D/A channels 0-3
23	ADC47_CV	0 = Do not perform A/D conversions on channels 4-7
		1 = Start A/D conversions on channels selected in 4-7 range
22	ADC_STBY	0 = Put A/D converters in full power mode
		1 = Put A/D converters in low-power standby mode
21	ADC47_CT	0 = Start A/D conversions in 4-7 range manually
		1 = Start A/D conversions in 4-7 range automatically (see bit 28)

Bit	*Name*	*Interpretation*	
20	ADC47_HS	0 = Use Control Register to select A/D channels in 4-7 range	
		1 = Use A/D Register to select A/D channels in 4-7 range	
19	ADC_SL7	ADC47_HS = 0	0 = Do not include A/D channel 7
			1 = Include A/D channel 7 in conversion
		ADC47_HS = 1	Ignored (A/D Register selects channels)
18	ADC_SL6	ADC47_HS = 0	0 = Do not include A/D channel 6
			1 = Include A/D channel 6 in conversion
		ADC47_HS = 1	Ignored (A/D Register selects channels)
17	ADC_SL5 / SCK47	ADC47_HS = 0	0 = Do not include A/D channel 5
			1 = Include A/D channel 5 in conversion
		ADC47_HS = 1	0 = Use internal clock (150 ns)
			1 = Use common clock (210 ns)
16	ADC_SL4	ADC47_HS = 0	0 = Do not include A/D channel 4
			1 = Include A/D channel 4 in conversion
		ADC47_HS = 1	Ignored (A/D Register selects channels)
15	ADC03_CV	0 = Do not perform A/D conversions on channels 0-3	
		1 = Start A/D conversions on channels selected in 0-3 range	
14	Reserved	Currently unused. Set to zero.	
13	ADC03_CT	0 = Start A/D conversions in 0-3 range manually	
		1 = Start A/D conversions in 0-3 range automatically (see bit 28)	
12	ADC03_HS	0 = Use Control Register to select A/D channels in 0-3 range	
		1 = Use A/D Register to select A/D channels in 0-3 range	
11	ADC_SL3	ADC03_HS = 0	0 = Do not include A/D channel 3
			1 = Include A/D channel 3 in conversion
		ADC03_HS = 1	Ignored (A/D Register selects channels)
10	ADC_SL2	ADC03_HS = 0	0 = Do not include A/D channel 2
			1 = Include A/D channel 2 in conversion
		ADC03_HS = 1	Ignored (A/D Register selects channels)

Bit	*Name*	*Interpretation*	
9	ADC_SL1 / SCK03	ADC03_HS = 0	0 = Do not include A/D channel 1
			1 = Include A/D channel 1 in conversion
		ADC03_HS = 1	0 = Use internal clock (150 ns)
			1 = Use common clock (210 ns)
8	ADC_SL0	ADC03_HS = 0	0 = Do not include A/D channel 0
			1 = Include A/D channel 0 in conversion
		ADC03_HS = 1	Ignored (A/D Register selects channels)
7	ENC7_IDX	0 = Disable index pulse for encoder channel 7	
		1 = Enable index pulse for encoder channel 7	
6	ENC6_IDX	0 = Disable index pulse for encoder channel 6	
		1 = Enable index pulse for encoder channel 6	
5	ENC5_IDX	0 = Disable index pulse for encoder channel 5	
		1 = Enable index pulse for encoder channel 5	
4	ENC4_IDX	0 = Disable index pulse for encoder channel 4	
		1 = Enable index pulse for encoder channel 4	
3	ENC3_IDX	0 = Disable index pulse for encoder channel 3	
		1 = Enable index pulse for encoder channel 3	
2	ENC2_IDX	0 = Disable index pulse for encoder channel 2	
		1 = Enable index pulse for encoder channel 2	
1	ENC1_IDX	0 = Disable index pulse for encoder channel 1	
		1 = Enable index pulse for encoder channel 1	
0	ENC0_IDX	0 = Disable index pulse for encoder channel 0	
		1 = Enable index pulse for encoder channel 0	

Table 11 Control Register bits

Encoder Index Pulses

The ENC0_IDX to ENC7_IDX bits control the index pulse logic for the encoder channels. Set the bit to '1' to enable the index pulse on the corresponding encoder channel. Set the bit to '0' to disable use of the index pulse on the corresponding channel. Refer to the discussion of the Encoder Control Registers for more details on using the index pulse of the encoders.

On reset, the ENC*n*_IDX bits are cleared, disabling the index pulses of all encoders.

A/D Channel and Timing Selection

There are two A/D converter chips on the Q8 data acquistion card. Each chip can simultaneously sample and hold four channels and then sequentially convert each channel at high speed into an internal FIFO. The first A/D chip, which handles channels 0 through 3, is controlled using bits 8-15 in the Control Register and bits 0-15 in the A/D Register. The second A/D chip handles channels 4 through 7. It is configured using bits 16-23 in the Control Register and bits 16-31 in the A/D Register.

The two A/D chips are called ADC03 and ADC47 respectively to reflect the analog channels that they convert. Which channels are converted is determined either by the Control Register or by programming the A/D chips themselves. The register used depends on the state of the ADC03_HS and ADC47_HS bits of the Control Register.

When ADC03_HS is set to '0' then the channels converted by ADC03 are determined by the ADC_SL0 through ADC_SL3 bits of the Control Register. If ADC_SL0 is a '1', then A/D channel 0 will be converted when A/D conversions are started for channels in the 0-3 range. If ADC_SL0 is a '0', then A/D channel 0 will not be converted. Similarly, ADC_SL1 through ADC_SL3 determine whether channels 1 through 3 are converted. Conversion values are written to the internal FIFO in ascending order. For example, if channels 0, 2 and 3 are selected for conversion, then the conversion result for channel 0 is written to the internal FIFO first, followed by channel 2 and finally channel 3.

When ADC03_HS is set to '1', then the channels converted by ADC03 are programmed by writing to the A/D Register. See the format of the A/D Register to see how channels are selected. In this case, ADC_SL0, ADC_SL2 and ADC_SL3 are ignored and the meaning of the ADC_SL1 bit changes to ADC_SCK03.

Each A/D chip can use one of two clock sources: an internal 150 ns clock or a 210 ns clock that is common to both A/D chips. An A/D conversions take 16 clock cycles to complete. The track and hold acquisition time is 0.35 µs. Each A/D chip samples all four of its input channels simultaneously. Hence, using the internal clock results in an A/D conversion every 2.4 µs or 100 kSPS per channel using both A/D chips if all 8 channels are being read:

```
150 ns/clock cycle x 16 clock cycles/channel        = 2.4 µs/ch.
1 channel / (0.35 µs + 4 channels/ADC * 2.4 µs/ch.) = 100 kSPS
```

For two channels (one per A/D chip), the conversion rate is:

```
1 channel / (0.35 µs + 1 channel/ADC * 2.4 µs/ch.) = 363 kSPS
```

Using the common clock yields a conversion result every 3.36 µs or an average of 73 kSPS per channel using both A/D chips, if all 8 channels are being read:

```
210 ns/clock cycle x 16 clock cycles/channel      = 3.36 µs/ch.
8 channels / (0.35 µs + 4 chs./ADC * 3.36 µs/ch.) = 73 kSPS
```

For two channels (one per A/D chip), the conversion rate is:

```
1 channel / (0.35 µs + 1 channel/ADC * 3.36 µs/ch.) = 270 kSPS
```

The "internal" clock is internal to the A/D chip.

While the internal clock provides faster conversion times, the two A/D chips may perform their conversions at different times, because each chip has its own internal clock.

Hence, the ADC03_EOC and ADC03_RDY flags may go high at a different time than the ADC47_EOC and ADC47_RDY flags respectively even if conversions on both chips are started at the same time. The timing difference may be as large as one internal clock period or 150 ns.

In contrast, the common clock is supplied to both A/D chips.

Thus, use of the common clock guarantees that both A/D chips yield conversion results at the same time.

Thus, the ADC03_EOC and ADC47_EOC flags are guaranteed to go high at the same time. The ADC03_RDY and ADC47_RDY flags will also go high at the same time if the two chips are converting the same number of channels. Use of the common clock makes programming simpler when more than 4 analog channels are being converted, but does not provide the maximum throughput possible with the A/D converters.

The common clock may only be used when ADC03_HS is '1'. In this case, channels are selected by programming the A/D Register instead of using the ADC_SL0-3 bits of the Control Register. In this case, the ADC_SL1 / SCK03 bit of the Control Register is used to control whether the internal or common clock is used. Set the ADC_SCK03 to '1' to select the common clock. Set ADC_SCK03 to '0' to choose the internal clock.

Programming the ADC47 chip is similar. When ADC47_HS is '0', the ADC_SL4 through ADC_SL7 bits are used to select the channels to convert. When ADC47_HS is '1', the channels are selected by writing to the A/D Register and the ADC_SL5 / SCK47 bit is used to select either the internal or common clock. Set ADC_SCK47 to '1' to select the common clock. Set it to '0' to choose the internal clock.

Starting A/D Conversions Manually

A/D conversions are started manually by writing a '1' to the ADC03_CV or ADC47_CV bits of the Control Register. Writing a '1' to the ADC03_CV bits causes the ADC03 chip to begin converting the analog input channels that were selected in the 0-3 range. Similarly, Writing a '1' to the ADC47_CV bits causes the ADC47 chip to begin converting the analog input channels that were selected in the 4-7 range. Writing a '1' to both bits causes both A/D

chips to begin conversions. When conversions are initiated, each A/D chip samples all four channels simultaneously and holds those values while it converts the requested channels.

Hence, the Q8 data acquisition card is capable of sampling all eight analog inputs simultaneously. The card can also convert two channels simultaneously, yielding twice the throughput of a data acquisition system based on a single A/D.

While the format of the Control Register implies that the A/D channels can be selected at the same time as conversions are initiated, the two operations must be performed separately.

The A/D channels cannot be changed at the same time as conversions are initiated. Write once to the Control Register to select the new channels and timing. Write a second time to initiate the A/D conversions (changing only the ADC03_CV or ADC47_CV bits).

For example, to perform A/D conversions on channels 0 and 1 using the Control Register for channel selection and the internal clock of the ADC03 chip:

```
pQ8->control = 0x00000300;   /* Select channels 0 and 1 */
pQ8->control = 0x00008300;   /* Start A/D conversions   */
```

Note that the channel selection (SL*n*) and register selection (HS) bits must be the same in the second write. If the same channels are being converted each time, then subsequent A/D conversions may be initiated by a single write to the Control Register.

It is slightly more efficient to program the channel selection using the Control Register instead of the A/D Register because other bits in the Control Register, such as the watchdog or encoder index pulse enable bits, may be configured at the same time. However, use of the Control Register to select the A/D channels precludes the use of the common clock. The common clock may only be used when channels are selected using the A/D Register.

On reset, all bits of the Control Register are set to zero. Hence, the default mode for the A/D converters is the Control Register channel selection mode with no channels selected.

Starting A/D Conversions Automatically

A/D conversions may also be initiated periodically by the hardware, or by an external signal. The CTEN_CV bit controls whether A/D conversions are triggered by expiration of the Counter or an external signal on the CNTR_EN input line.

Set CTEN_CV to zero, to allow A/D conversions to be triggered by expiration of the Counter. In this case, the ADC47_CT and ADC03_CT bits control whether the Counter is actually used, or whether manual conversions are performed. Setting the ADC03_CT bit to one causes ADC03 to perform conversions on the selected channels in the range 0-3 whenever the Counter expires. Setting the ADC47_CT bit to one causes ADC47 to perform conversions on the selected channels in the range 4-7 whenever the Counter expires.

For example, by programming the Counter to 1 ms, setting CTEN_CV to zero, and ADC03_CT to one, the Q8 will convert all selected channels in the range 0-3 every 1 ms. The converted values are stored in the ADC03 FIFO. By enabling interrupts on ADC03_RDY, which indicates that conversions have been completed on all selected channels, you can interrupt the PC every 1 ms, but only after all the A/D conversions have been completed, and the results are waiting in the ADC03 FIFO. Thus, the ISR need only read the data from the FIFO – it doesn't have to wait for the A/D conversions to complete because they are already done! This configuration is ideal for control systems because no CPU time is wasted waiting for A/D conversions to complete.

Note however that the FIFO *must be read before the Counter expires again and starts another A/D conversion sequence.* Otherwise the contents of the FIFO will be replaced with the new conversion results.

Set CTEN_CV to one to allow A/D conversions to be triggered by a falling edge on the CNTR_EN line. The CNTR_EN line is pulled high on the Q8 so that the Counter is enabled by default. By setting the CTEN_CV bit in the Control Register, this input is used instead as a means of triggering A/D conversions externally.

This mode is useful for synchronizing multiple Q8 cards. On the master Q8, program the Counter Control Register such that the output of the Counter appears on the CNTR_OUT line. Also use the Counter to trigger A/D conversions on the master Q8. Wire the CNTR_OUT line of the master Q8 to the CNTR_EN line of all the slave Q8 devices. Program each slave Q8 to trigger A/D conversions on the CNTR_EN line. Thus, the master Q8 will trigger A/D conversions on all the slaves at the same time as it is performing its own A/D conversions. Furthermore, if the master and slave(s) are in separate computers, and each computer interrupts on ADCxx_RDY, then interrupts on each PC will also be synchronized (within the constraints of interrupt latency and jitter).

The polarity of the CNTR_EN input may be changed using the CNTR_POL bit in the Counter Control Register. See the Counter Control Register for details.

A/D Standby Mode

The A/D converters have a low-power "standby" mode. To reduce the power consumption of the A/D converters, write a '1' to the ADC_STBY bit in the Control Register. This bit must be cleared before A/D conversions are begun. This feature is only useful in embedded applications where power requirements must be kept to a minimum or to support power management. Note that for the lowest power consumption, the counter and watchdog counter should be disabled as well via the Counter Control Register.

Conversion results may be still be read from the A/D chip's internal FIFO when the A/D converter is in standby mode. Hence, the A/D chips may be placed in standby mode as soon

as A/D conversions are complete (when ADCxx_RDY goes high).

Conversions cannot be performed until 1 μs after the chips have been taken out of standby mode. Take the converters out of standy mode by clearing the ADC_STBY bit of the Control Register.

Thus, to conserve the most power, put the A/D chips in standby mode just after performing the A/D conversions and then re-enable the A/D chips 1 μs before the next set of conversions.

On reset, the ADC_STBY bit is set to zero, disabling standby mode.

D/A Transparent Mode

There are two DAC chips in the Q8 data acquisition system. Each DAC chip supports four analog output channels. The first DAC chip manages analog output channels 0-3, and is thus named DAC03. The second DAC chip, called DAC47, handles analog outputs 4-7.

The DAC chips are double-buffered, allowing the D/A value to be preloaded into the DAC before being transferred to the DAC's analog outputs.

Hence, the Q8 data acquisition system is capable of updating all eight D/A channels simultaneously.

Thus, a DAC operation normally entails a series of 32-bit writes to preload the D/A values, and then a single 32-bit write to update the D/A outputs with the preloaded values.

In certain situations, however, it is desirable to preload and update the D/A outputs in a single 32-bit write. This "transparent" mode of operation is enabled by setting the DAC03_TR or DAC47_TR bits in the Control Register. Setting the DAC03_TR bit to '1' enables transparent mode for DAC03. When transparent mode is enabled, any values written to the lower 16 bits of D/A Output Registers A-D appear immediately at the DAC outputs, without the need to write to the D/A Update Register. Transparent mode is disabled by writing a '0' to the DAC03_TR bit.

Similarly, setting the DAC47_TR bit enables transparent mode for D/A channels 4-7. When transparent mode is enabled for DAC47, any values written to the upper 16 bits of the D/A Output Registers A-D appear immediately at the DAC outputs. Transparent mode is disabled for DAC47 by writing a '0' to the DAC47_TR bit.

On reset, the DAC03_TR and DAC47_TR bits are cleared, disabling transparent mode.

External Interrupt Control

The Q8 supports a separate external interrupt line called EXT_INT. This line has a pullup resistor, so it is normally high. When this line goes low, it causes the EXT_INT bit in the Interrupt Status Register to go high. Thus, it may be used to generate an interrupt by enabling the EXT_INT bit in the Interrupt Enahle Register. The polarity of the EXT_INT line can be reversed by setting the EXT_POL bit in the Control Register. However, if the EXT_INT line is configured to trigger on a rising edge, then the EXT_INT line should be held low by external hardware until an interrupt should be generated. Note that interrupt generation is edge-triggered, not level-sensitive.

The Q8 also allows the EXT_INT line to be configured as an external watchdog input. Byh setting the EXT_ACT bit in the Control Register, a falling edge on the EXT_INT line will cause the DACs to be reset to zero and the digital outputs to be pulled high (by clearing the Digital Direction Register), just like the Watchdog counter. The resetting of the outputs is actually based on the EXT_INT bit in the Interrupt Status Register. As long as the EXT_INT bit in the Interrupt Status Register is high, and EXT_ACT in the Control Register is set, the DACs will be continuously reset to zero, and the Digital Direction Register will likewise be reset (causing all digital outputs to be pulled high).

Thus, to release the "external watchdog", clear the EXT_INT bit in the Interrupt Status Register and/or clear the EXT_ACT bit in the Control Register. **Do so *before* reprogramming the DAC modes and outputs, and the Digital Direction Register**.

See the Watchdog section under the Counter Control Register for more information about watchdog functionality.

This external watchdog feature is very useful for ensuring system safety. For example, by wiring the limit switches and emergency stop button via a wired-OR configuration to the EXT_INT line and enabling the external watchdog feature, **the outputs of the Q8 will be automatically reset *in hardware, with no software intervention*, the instant a limit is reached or the emergency stop button is pressed**! Furthermore, by enabling an interrupt on the EXT_INT line, this event can also be detected immediately in software.

Status Register

Byte Offset	0x0c
Read/Write	read only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	unknown

The Status Register allows the current condition of various interrupt sources and other signals to be polled. Unlike the Interrupt Status Register, whose bits are set on the rising edge of an interrupt source, the Status Register reflects the actual signals. Hence, bits in the Status Register do not have to be explicitly cleared. The contents of this register are shown in Table 12 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
							CNTREN	EXTINT	FUSE	ADC47FST	ADC03FST	ADC47RDY	ADC03RDY	ADC47EOC	ADC03EOC	ENC7FLG2	ENC7FLG1	ENC6FLG2	ENC6FLG1	ENC5FLG2	ENC5FLG1	ENC4FLG2	ENC4FLG1	ENC3FLG2	ENC3FLG1	ENC2FLG2	ENC2FLG1	ENC1FLG2	ENC1FLG1	ENC0FLG2	ENC0FLG1

Table 12 Status Register

Encoder Flags

The ENC*n*_FLG*i* bits reflect the status of the encoder flags. These flags are programmed via the Encoder Control Registers.

When the ENC*n*_FLG*i* bits are used as a $\overline{\text{CARRY}}$, $\overline{\text{BORROW}}$ or $\overline{\text{COMPARE}}$ output, the bits are normally high and only go low for two PCI clock cycles, or 60 ns. Hence, it is generally not possible to poll these bits using the Status Register when they are configured as $\overline{\text{CARRY}}$, $\overline{\text{BORROW}}$ or $\overline{\text{COMPARE}}$. Use the Interrupt Status Register instead in these casese, remembering to clear the bits of the Interrupt Status Register in software after a $\overline{\text{CARRY}}$, $\overline{\text{BORROW}}$ or $\overline{\text{COMPARE}}$ operation occurs. Alternatively, read the Encoder status from the Encoder Control Registers.

However, these flags can be polled via the Status Register when configured as the up/down, index or error flags. Hence, the Status Register is particularly useful for polling for an index pulse of an encoder channel, for determining the direction of motion or detecting errors due to excessive noise. Refer to the Encoder Control Registers for more information about the encoder flags.

A/D Conversion Flags

The ADC03_EOC and ADC47_EOC bits reflect the end-of-conversion outputs of the A/D converter chips. These bits go high for 120 ns to 180 ns. Hence, polling must be very fast in order to catch these bits. It is probably better to use the Interrupt Status Register instead to poll for these signals. Note however that the Interrupt Status Register must be explicitly cleared by software between EOC pulses.

The ADC03_RDY and ADC47_RDY bits are the inverse of the BUSY outputs of the A/D converter chips. They are clear while the A/D chips are converting the input signals. As soon as all selected channels have been converted, these bits go high. Note that these signals go low immediately after a conversion is started by writing to the Control Register. Hence, the Status Register may be polled immediately after starting a conversion. No delays are required in software.

The ADC03_FST and ADC47_FST bits are set as long as the first conversion results are available from the A/D chips. Once the first conversion result has been read, these bits go low (provided another conversion result is available in the A/D FIFOs). Since the FIFOs are cleared when a conversion starts, these signals have limited utility.

The FUSE bit reflects the status of the +5V fuse on the Q8 terminal board. During normal operation, this bit is zero. If the fuse fails, due to improper connection of an encoder, or encoder failure, then this bit goes high and will remain high until the fuse is replaced. This input also goes high if the ribbon cable has not been connected to the J1 connector. Hence, this bit may also be used initially to test for the Q8 terminal board.

Note that the analog and digital outputs cannot be used as long as the terminal board is disconnected or the fuse is blown, because detection of fuse failure automatically resets the DACs and sets all digital outputs high.

The EXT_INT bit reflects the inverted value of the external $\overline{\text{EXT\_INT}}$ input. Hence, the EXT_INT bit is set as long as the EXT_INT input is low. The $\overline{\text{EXT\_INT}}$ input is an active low signal.The Q8 samples the EXT_INT signal on the rising edge of the 30 ns PCI clock. **The value of the EXT_POL bit in the Control Register does not affect the EXT_INT bit in the Status Register.**

The $\overline{\text{EXT\_INT}}$ input functions as an extra digital input. It has the added advantage that it can also be used to generate an interrupt on a falling edge (ie, a rising edge of EXT_INT).

The EXT_INT input is primarily useful for generating interrupts from an external interrupt source. For example, by connecting the EXT_INT input to an emergency stop system, the software can be notified immediately when the emergency stop is engaged or a limit switch is encountered.

The CNTR_EN bit reflects the value of the external CNTR_EN input. This external input is used to enable or disable the general purpose counter. The input has a pullup resistor so that the counter is enabled by default if no external signal is connected to this input. The input allows the counter to be gated by an external signal. For example, the CNTR_EN input can be used for event timing or it can be tied to an emergency stop system to prevent counter interrupts from occurring on an external condition.

Note that counter interrupts will not occur even if the counter is disabled when the count value is zero. The Q8 only generates interrupts on the rising edge of an interrupt source – it is not level-sensitive. An interrupt will occur, however, as soon as the counter is re-enabled, if the count value is not changed by software.

The external CNTR_EN signal is active high. The Q8 samples the CNTR_EN signal on the rising edge of the 30 ns PCI clock. It is important to note that **the value of the CNTR_POL bit in the Counter Control Register does not affect this state of the CNTR_EN bit in the Status Register**.

Counter Preload Registers

The Q8 contains two general purpose, 32-bit counters called Counter and Watchdog. Both counters decrement every 30 nanoseconds when enabled. When the count value reaches zero, the counter output is toggled and the count value is reloaded from a preload register.

Each counter has four preload registers, organized in two pairs, or "sets". Each preload register set consists of a Preload Low Register and a Preload High Register. Hence, the preload registers for Counter are:

1. Counter Preload Low Register – Set #0
2. Counter Preload High Register – Set #0
3. Counter Preload Low Register – Set #1
4. Counter Preload High Register – Set #1

Each counter has two modes: square wave mode and pulse-width modulated (PWM) mode. The mode of Counter is selected by the CT_MODE bit in the Counter Control Register. Which of the four preload registers is used in counting depends on the counter mode and the counter output.

The Watchdog counter has an identical set of preload registers and functions in the same way.

In square wave mode, only one preload register is used. In this mode, the count value is always reloaded from the same preload register. Hence, the output of the counter is a square wave with a 50% duty cycle. The period of the square wave, in this case, is calculated as:

```
period = (preload value + 1) * 60 ns
```

For example, a preload value of 16,666 results in a Counter period of 0.001000020 seconds, or approximately 1 ms. The minimum preload value is 0, yielding a minimum Counter period of 60 ns (16.7 MHz).

The maximum preload value is $2^{32} - 1$. The maximum preload value results in a Counter period of 257.69803776 seconds or just over 4 minutes.

The preload register used by the Counter in square wave mode is determined by two bits in the Counter Control Register: CT_RSET and CT_PRSEL. The CT_RSET bit determines the preload register set to use and the CT_PRESEL bit selects whether the Preload Low Register or the Preload High Register will be employed.

In PWM mode, both the Preload Low Register and the Preload High Register are used. However, only one preload register set is used at a time. In this mode, the counter output determines which preload register is used. When the counter output goes to zero, the counter is loaded from the Preload Low Register. When the counter output goes high, the counter is loaded from the Preload High Register. Thus, the Preload Low Register determines the duration of the low pulse of the square wave output, and the Preload High Register sets the duration of the high pulse. In this manner, the duty cycle of the counter output can be controlled with 30ns resolution. The period of the counter output in PWM mode is then:

```
period = (preload low value + preload high value + 2) * 30 ns
```

To force the output to always be high or to always be low, disable the counter and set the counter output value using the CT_ENAB, CT_LD and CT_VAL bits of the Counter Control Register.

The reason there are two sets of preload registers is to allow the duty cycle of the PWM output to be changed without affecting the period of the output. The preload registers are essentially double-buffered. It is possible to write to one set of preload registers while the counter is reading the other set.

The CT_RSET bit in the Counter Control Register controls which set of preload registers is used when reloading the Counter. It also determines which preload register is read when reading the Counter Preload Register.

The CT_WSET bit in the Counter Control Register determines which set of preload registers is represented by the Counter Preload Low Register and the Counter Preload High Register in the Q8 memory space. For example, when CT_WSET is '0', writing to the Counter Preload Low Register will set the value of Counter Preload Low Register – Set #0. When CT_WSET is '1', writing to the same register will set the value of Counter Preload Low Register – Set #1.

Counter Preload Register

Byte Offset	0x10
Read/Write	read only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Counter Preload Register is used to read a preload register. Which of the four preload registers of the Counter is read is determined by the counter mode, the counter output, and the active register set and preload register.

In square wave mode (when CT_MODE is '0'), the preload register that is returned by reading the Counter Preload Register is determined by the CT_PRSEL and CT_RSET bits of the Counter Control Register. The CT_RSET bit determines which register set will be used:

```
0 = preload register set #0 is used
1 = preload register set #1 is used
```

Note that CT_RSET also selects the preload register set currently in use by the Counter.

The CT_PRESEL bit selects whether the preload low register or the preload high register will be read in the active register set:

```
0 = preload low register is read
1 = preload high register is read
```

Note that CT_PRSEL also selects the preload register used by the Counter in square wave mode.

In PWM mode (when CT_MODE is '1'), the preload register that is returned by reading the Counter Preload Register is determined by the counter output and active register set. As in square wave mode, the CT_RSET bit of the Counter Control Register determines which register set is referenced.

Whether the preload low register or the preload high register is returned is dictated by the counter output. The preload register that is read is the one that will be loaded the next time the Counter expires. For example, if the Counter output is currently '0', then the preload high register will be returned because the next time the Counter expires, the output will change to a '1' and the preload high register will be loaded into the Counter. Likewise, when the Counter output is '1', then the preload low register will be returned.

Note that there is typically no need to read the preload registers. This feature is primarily available for debugging purposes.

To read the current Counter value, read the Counter Register instead.

Counter Preload Low Register

Byte Offset	0x10
Read/Write	write only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Counter Preload Low Register is used to set the period of the general purpose Counter in square wave mode or the duration of the low pulse in PWM mode.

Write to the Counter Preload Low Register to set the preload value. The register set used is determined by the CT_WSET bit in the Counter Control Register. Note that writing to this register does not immediately load the Counter with the new value. The Counter loads the contents of a Counter Preload Register the next time it expires. To load the Counter immediately with the new value, write to the Counter Control Register's CT_LD and CT_VAL bits after writing to the appropriate Counter preload register. Refer to section Counter Preload Registers for more information.

Counter Register

Byte Offset	0x14
Read/Write	read only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	reset to zero

The Counter Register contains the current counter value. The Counter is a 32-bit periodic counter with 30 ns resolution. It counts downward from the value in one of the Counter Preload Registers to zero, toggles the output and then loads the next preload value. The output can be enabled on the external CNTR_OUT line using the CT_OUTEN bit in the Counter Control Register.

Reading from the Counter Register returns the current counter value. Note that since the output toggles each time the counter expires, reading the count value is not enough to determine the time from the start of the period. The output value is also required. The current output value can be read from the Counter Control Register, in the CT_VAL bit.

The Counter can optionally be used to cause a periodic interrupt or to initiate A/D conversions. When the CNTR_OUT enable bit is set in the Interrupt Enable Register, an interrupt will be generated each time there is a rising edge on the counter output. Since the Counter is periodic, an interrupt will be generated each period.

When the ADC03_CT bit is set in the Control Register, and the CTEN_CV bit is zero, the Counter will initiate A/D conversions on channels selected in the range 0-3. Similarly, when the ADC47_CT bit is set in the Control Register, the Counter will initiate A/D conversions on channels selected in the range 4-7. The A/D converter will use either the internal or external clock, according to the configuration set in the Control Register and the A/D Register.

Note that the Counter triggers the A/D conversions – it does not serve as the A/D conversion clock. **Conversions occur at the internal or common clock rates (2.4 μs per channel for the internal clock or 3.36 μs per channel for the common clock).**

For example, suppose the internal A/D clock is selected in the Control Register, the ADC03_CT bit is set, CTEN_CV is zero, channels 0-2 are selected and the Counter period is 1 ms. In this case, the Counter will trigger A/D conversions every 1 ms. When the Counter triggers a conversion, the ADC03 will sample channels 0-2 simultaneously and then start converting each channel using its internal clock. Each channel will take 2.4 μs to convert.

Hence, when the 0.35 µs track and hold acquisition time is included, it will take 7.55 µs for all three channels to be converted.

If interrupts are enabled on the ADC03_RDY line then the CPU will be interrupted every 1 ms. The interrupt handler can immediately read the results of the three A/D conversions, which occurred at the fastest possible rate just prior to the invocation of the interrupt handler. This configuration is ideal for control systems because no time must be spent in the interrupt handler for performing A/D conversions and the timing of the interrupt handler is based on a hardware timer with 30 ns resolution.

Counter Preload High Register

Byte Offset	0x14
Read/Write	write only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Counter Preload High Register is used to set the duration of the high pulse of the general purpose Counter. It may also be used to set the period of the Counter in square wave mode if CT_PRSEL is '1' in the Counter Control Register.

Write to the Counter Preload High Register to set the preload value. The preload register set that is modified depends upon the value of the CT_WSET bit in the Counter Control Register. Also note that writing to this register does not immediately load the Counter with the new value. The Counter loads the contents of a Counter Preload Register the next time it expires. To load the Counter immediately with the new value, write to the Counter Control Register's CT_LD and CT_VAL bits after writing to the appropriate Counter Preload Register. Refer to the Counter Preload Registers section for more details.

The Counter Preload High Register is a write-only register. Reading from this register returns the Counter's current count value, since the read-only Counter Register is at the same register offset as the Counter Preload High Register.

Watchdog Preload Register

Byte Offset	0x18
Read/Write	read only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Watchdog Preload Register is used to read a preload register of the Watchdog counter. Which of the four preload registers of the Watchdog counter is read is determined by the counter mode, the counter output, and the active register set and preload register.

In square wave mode (when WD_MODE is '0'), the preload register that is returned by reading the Watchdog Preload Register is determined by the WD_PRSEL and WD_RSET bits of the Counter Control Register. The WD_RSET bit determines which register set will be used:

```
0 = preload register set #0 is used
1 = preload register set #1 is used
```

Note that WD_RSET also selects the preload register set currently in use by the Watchdog counter.

The WD_PRESEL bit selects whether the preload low register or the preload high register will be read in the active register set:

```
0 = preload low register is read
1 = preload high register is read
```

Note that WD_PRSEL also selects the preload register used by the Watchdog counter in square wave mode.

In PWM mode (when WD_MODE is '1'), the preload register that is returned by reading the Watchdog Preload Register is determined by the counter output and active register set. As in square wave mode, the WD_RSET bit of the Counter Control Register determines which register set is referenced.

Whether the preload low register or the preload high register is returned is dictated by the counter output. The preload register that is read is the one that will be loaded the next time the Watchdog counter expires. For example, if the Watchdog counter output is currently '0', then the preload high register will be returned because the next time the Watchdog counter expires, the output will change to a '1' and the preload high register will be loaded into the Watchdog counter. Likewise, when the Watchdog counter output is '1', then the preload low

register will be returned.

Note that there is typically no need to read the preload registers. This feature is primarily available for debugging purposes.

To read the current Watchdog counter value, read the Watchdog Register instead.

Watchdog Preload Low Register

Byte Offset	0x18
Read/Write	write only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Watchdog Preload Low Register is used to set the period of the general purpose Watchdog in square wave mode or the duration of the low pulse in PWM mode.

Write to the Watchdog Preload Low Register to set the preload value. The register set used is determined by the WD_WSET bit in the Counter Control Register. Note that writing to this register does not immediately load the Watchdog counter with the new value. The Watchdog counter loads the contents of a Watchdog Preload Register the next time it expires. To load the Watchdog counter immediately with the new value, write to the Counter Control Register's WD_LD and WD_VAL bits after writing to the appropriate Watchdog preload register. Refer to section Counter Preload Registers for more information.

Watchdog Register

Byte Offset	0x1c
Read/Write	read only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	reset to zero

The Watchdog Register contains the current counter value for the Watchdog counter. The Watchdog counter is a 32-bit periodic counter with 30 ns resolution. It counts downward from the value in one of the Watchdog Preload Registers to zero, toggles the output and then loads the next preload value. The output can be enabled on the external WATCHDOG line using the WD_OUTEN bit in the Counter Control Register.

Reading from the Watchdog Register returns the current counter value. Note that since the output toggles each time the counter expires, reading the count value is not enough to determine the time from the start of the period. The output value is also required. The current output value can be read from the Counter Control Register, in the WD_VAL bit.

The Watchdog counter can optionally be used to cause a periodic interrupt or act as a watchdog timer. When the WATCHDOG bit is set in the Interrupt Enable Register, an interrupt will be generated each time there is a rising edge on the counter output. Since the Watchdog counter is periodic, an interrupt will be generated each period.

The Watchdog Counter does not stop counting when the counter reaches zero. It is a general purpose counter that can be used as a normal periodic counter if its watchdog features are deactivated.

The external $\overline{\text{WATCHDOG}}$ output can be configured as the counter output or as a "watchdog" output via the WD_SEL bit of the Counter Control Register. To function as an output, it must be enabled using the WD_OUTEN bit.

If WD_SEL is '0 then the $\overline{\text{WATCHDOG}}$ output reflects the contents of the WATCHDOG bit in the Interrupt Status Register. This bit is set whenever there is a rising edge on the output of the Watchdog counter. It remains set until it is explicitly cleared in software, by writing a '1' to that bit in the Interrupt Status Register.

Thus, if the Watchdog counter is allowed to expire, the WATCHDOG bit in the Interrupt Status Register is set and the external $\overline{\text{WATCHDOG}}$ output goes low (being active low). The $\overline{\text{WATCHDOG}}$ output stays low until the WATCHDOG bit in the Interrupt Status Register is cleared. This output can be used to automatically engage safety brakes or disable power amplifiers, for example, when the watchdog timer expires. It can also be tied to the EXT_INT line of other Q8 cards to disable the outputs of those cards also when the Watchdog expires. Hence, this functionality is an important safety feature for industrial control sytems.

If WD_SEL is '1', then the WATCHDOG output reflects the output of the Watchdog counter. Hence, it can be used to generate a square wave output of programmable duty cycle. In this mode, it functions just like the CNTR_OUT line when it is enabled.

The Watchdog Counter can optionally be used to disable the digital and analog outputs of the Q8. When the WDOG_ACT bit is set in the Counter Control Register, the

Watchdog Counter will clear the Digital Direction Register and reset the D/A converters when it expires.

Clearing the Digital Direction Register causes all digital I/O lines to become inputs. Since the digital I/O port has pullup resistors, this action will cause all digital outputs to be pulled high. Hence, all digital outputs become '1' when the Watchdog Counter expires and the WDOG_ACT bit is set.

Resetting the D/A converters puts the D/A converters into unipolar, 10V mode and sets the preload and output values to zero. Hence, when the WDOG_ACT bit is set in the Control Register, the D/A converters will output 0V after the Watchdog Counter expires.

See the discussion of the WDOG_ACT bit in the Counter Control Register for more information on the watchdog features of this counter.

Watchdog Preload High Register

Byte Offset	0x1c
Read/Write	write only
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	indeterminate

The Watchdog Preload High Register is used to set the duration of the high pulse of the general purpose Watchdog counter. It may also be used to set the period of the Watchdog counter in square wave mode if WD_PRSEL is '1' in the Counter Control Register.

Write to the Watchdog Preload High Register to set the preload value. The preload register set that is modified depends upon the value of the WD_WSET bit in the Counter Control Register. Also note that writing to this register does not immediately load the Watchdog counter with the new value. The Watchdog counter loads the contents of a Watchdog Preload Register the next time it expires. To load the Watchdog counter immediately with the new value, write to the Counter Control Register's WD_LD and WD_VAL bits after writing to the appropriate Watchdog Preload Register. Refer to the Counter Preload Registers section for more details.

The Watchdog Preload High Register is a write-only register. Reading from this register returns the Watchdog's current count value, since the read-only Watchdog Register is at the same register offset as the Watchdog Preload High Register.

Counter Control Register

Byte Offset	0x20
Read/Write	read or write
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	cleared

The Counter Control Register controls the two 32-bit general purpose counters on the card: Counter and Watchdog. Both counters may be used to generate periodic interrupts. However, the Watchdog counter may also be activated as a watchdog timer, and Counter may be used to trigger periodic A/D conversions. The Counter Control Register is used to control these two counters and is divided into two sections. The upper 16 bits control the Watchdog counter and the lower 16 bits control Counter. For example, it configures whether the counters output a square wave or pulse-width modulated (PWM) signal. The contents of this register are shown in Table 13 below. The register is cleared during reset. Hence, after reset the Q8 will be have a known default configuration.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
						WDLD	WDVAL	WDACT	WDSEL	WDOUTEN	WDPRSEL	WDWSET	WDRSET	WDMODE	WDENAB							CTLD	CTVAL		CTENPOL	CTOUTEN	CTPRSEL	CTWSET	CTRSET	CTMODE	CTENAB

Table 13 Counter Control Register

The bits of the Counter Control Register are described in Table 14 below. When the Countner Control Register is read, the contents of the register are returned, with the exception of the shaded bits and the CT_LD and WD_LD bits (bits 9 and 25 respectively). These bits always return '0' on a read since they do not reflect the configuration of the card.

Bit	*Name*	*Interpretation*
31→26	Reserved	Currently unused. Set to zero.
25	WD_LD	0 = No load operation (WD_VAL also ignored)
		1 = Load watchdog counter from active preload register and set counter output equal to value of WD_VAL bit.

Counter Control Register

Bit	*Name*	*Interpretation*	
24	WD_VAL	Value of watchdog counter output (ignored on writes if WD_LD bit is zero)	
23	WD_ACT	0 = Deactivate the watchdog features	
		1= Activate the watchdog features of the watchdog counter	
22	WD_SEL	0 = WATCHDOG output reflects watchdog state (active low)	
		1 = WATCHDOG output is output of Watchdog counter	
21	WD_OUTEN	0 = Disable WATCHDOG output. Output always high.	
		1 = Enable WATCHDOG output. Value determined by WD_SEL	
20	WD_PRSEL	WD_MODE = 0	0 = Use Watchdog Preload Low Register
			1 = Use Watchdog Preload High Register
		WD_MODE = 1	Ignored
19	WD_WSET	0 = Select watchdog register set #0 for writes to preload registers	
		1 = Select watchdog register set #1 for writes to preload registers	
18	WD_RSET	0 = Select watchdog register set #0 as active set and for reads	
		1 = Select watchdog register set #1 as active set and for reads	
17	WD_MODE	0 = Square wave mode (WD_PRSEL selects Preload Register)	
		1 = PWM mode (both Preload Low and High Registers used)	
16	WD_ENAB	0 = Disable the watchdog counter (disables counting)	
		1 = Enable the watchdog counter (enables counting)	
15→10	Reserved	Currently unused. Set to zero.	
9	CT_LD	0 = No load operation (CT_VAL also ignored)	
		1 = Load counter from active preload register and set counter output equal to value of CT_VAL bit	
8	CT_VAL	Value of counter output (ignored on writes if CT_LD = 0)	
7	Reserved	Currently unused. Set to zero.	
6	CTEN_POL	CTEN_CV = 0	0 = CNTREN is active high
			1 = CNTREN is active low
		CTEN_CV = 1	0 = CNTREN triggers on falling edge
			1 = CNTREN triggers on rising edge

Bit	*Name*	*Interpretation*	
5	CT_OUTEN	0 = Disable CNTR_OUT output. Output always high.	
		1 = Enable CNTR_OUT output. Value is output of Counter.	
4	CT_PRSEL	CT_MODE=0	0 = Use Preload Low Register only
			1 = Use Preload High Register only
		CT_MODE=1	Ignored
3	CT_WSET	0 = Select counter register set #0 for writes to preload registers	
		1 = Select counter register set #1 for writes to preload registers	
2	CT_RSET	0 = Select counter register set #0 as active set and for reads	
		1 = Select counter register set #1 as active set and for reads	
1	CT_MODE	0 = Square wave mode (CT_PRSEL selects Preload Register)	
		1 = PWM mode (both Preload Low and High Registers used)	
0	CT_ENAB	0 = Disable the counter (disables counting)	
		1 = Enable the counter (enables counting)	

Table 14 Counter Control Register bits

Watchdog Timer Support

The watchdog functionality of the Watchdog counter is activated by setting the WD_ACT bit of the Counter Control Register to '1'.

When used as a watchdog timer, the Watchdog counter resets the D/A outputs to zero and pulls all digital outputs high when the watchdog counter expires.

This feature is useful for detecting software failures and automatically disabling power amplifiers, motor drives and other systems in the event of a software failure. Since the outputs are changed in hardware, with no software intervention, the watchdog counter is an important feature for implementing safety systems in any environment.

The Watchdog counter pulls the digital outputs high by resetting the Digital Direction Register. Resetting this register causes all the digital I/O channels to become inputs. Since the digital I/O channels have pullup resistors, the channels that were used as digital outputs will be pulled high. The Digital I/O Register is left unchanged. Hence, reprogramming the Digital Direction Register is enough to restore the original digital output values.

The D/A outputs are set to zero by resetting the DAC chips themselves. In this case, the D/A preload registers and output latches are reset to zero and the DAC mode is changed to unipolar, 10V mode.

The watchdog continuously resets the DACs and Digital Direction Register when active. The reset action is actually based upon the WATCHDOG bit in the Interrupt Status Register. As long as the WATCHDOG bit in the Interrupt Status Register is high, and the watchdog reset feature is activated, the DACs and the Digital Direction Register will be reset.

Hence, after the watchdog counter expires, the WATCHDOG bit in the Interrupt Status Register must be cleared, and then the Digital Direction Register and the DACs reprogrammed to restore the outputs to their original state.

Reload the Watchdog counter and reset the WATCHDOG bit in the Interrupt Status Register *before* reprogramming the outputs.

For maximum safety, the software should be able to recover from the expiration of the watchdog timer, in addition to notifying the user of the event.

An interrupt may be generated when the Watchdog counter expires, allowing software to detect the watchdog expiration as well.

The status of the Watchdog counter may be monitored externally using the $\overline{\text{WATCHDOG}}$ output of the card, when the WD_OUTEN and WD_SEL bits of the Counter Control Register are programmed appropriately (WD_OUTEN=1, WD_SEL=0). In this case, the external $\overline{\text{WATCHDOG}}$ output is the inverse of the WATCHDOG status bit in the Interrupt Status Register. The Watchdog counter sets this bit when it expires. The status bit must be cleared explicitly in software by writing a '1' to the WATCHDOG bit in the Interrupt Status Register. Hence, the external $\overline{\text{WATCHDOG}}$ output is high as long as the watchdog counter has not expired.

When the Watchdog counter expires, the $\overline{\text{WATCHDOG}}$ output goes low, and remains low, until the WATCHDOG bit of the Interrupt Status Register is cleared.

The external output is designed to be used by power amplifiers and other systems as an emergency stop signal. Note that it only has this functionality when enabled in the Counter Control Register.

Thus, the watchdog counter can be used to automatically disable external devices, such as motor drive systems, that are connected to the Q8.

The Watchdog counter is typically reloaded every time the software enters its interrupt handler. Hence, under normal operating conditions, the Watchdog counter never expires. However, if the software fails, it will no longer reset the Watchdog counter and the counter will expire, causing all D/A outputs to be driven to zero and all digital outputs to go high and external devices to be disabled.

The Watchdog counter may be activated as a watchdog timer by setting the WDOG_ACT of the Counter Control Register to '1'. If the WDOG_ACT is set to '0', then the Watchdog counter functions as a normal 32-bit periodic counter and does not reset the DACs or set the

digital outputs when it expires.

When WD_SEL = 0 and WD_OUTEN = 1, the $\overline{\text{WATCHDOG}}$ output reflects the (inverted) status of the WATCHDOG bit in the Interrupt Status Register regardless of the state of the WDOG_ACT bit.

The Watchdog counter may be disabled in software by writing a '0' to the WDOG_EN bit in the Counter Control Register. When the Watchdog counter is disabled, it does not count. The counter holds its previous value. When the Watchdog counter is enabled, it counts down. When the counter reaches zero, it reloads its preload value and continues counting.

The Watchdog counter continues to count whenever it is enabled. Hence, if the WDOG_ACT bit is set and the counter is enabled, it will reset the D/A outputs and set the digital outputs every time the counter expires.

To stop it from resetting the outputs, deactivate the watchdog counter by clearing the WDOG_ACT bit or disable the counter by clearing the WDOG_EN bit or start loading the count value periodically before it can expire again.

When choosing to reload the count value, do so *before* reprogramming the D/A outputs and the Digital Direction Register, so that the Watchdog counter does not expire in the middle of configuration the outputs.

Watchdog Counter Support

Bits WD_LD and WD_VAL are used to set the current Watchdog counter value. When WD_LD is '1' on a write to the Counter Control Register, the contents of the currently active preload register are transferred to the Watchdog counter, without waiting for the counter to expire. Normally, the counter reads the contents of a preload register only when it expires.

The WD_VAL bit sets the current output value of the counter, and also determines which preload register is used. For example, if WD_VAL is '0' when WD_LD is '1', then the contents of the Watchdog preload low register will be transferred to the Watchdog counter. If WD_VAL is '1', then the contents of the Watchdog preload high register will be used. The WD_RSET bit controls which register set is used. If the WD_LD bit is '0', then the Watchdog counter contents are left unchanged, and the WD_VAL bit is ignored.

When WD_LD is '1' during a write, all other bits in the Counter Control Register, with the exception of the WD_VAL, CT_LD and CT_VAL should be left unchanged. Failure to do so can lead to unexpected behaviour.

This feature allows the count value to be set without waiting for the counter to expire. This functionality is useful for resetting the Watchdog counter when using it as a watchdog timer. The counter can be reloaded from the preload register without having to set the

preload register again simply by writing WD_LD = '1' and WD_VAL = '1' to the Counter Control Register.

Furthermore, since the Counter may also be loaded in this way, it allows both counter's values to be set at exactly the same time, via a single 32-bit write to the Counter Control Register. Thus, the two counters can be synchronized.

Note that if the Watchdog counter is disabled (by setting the WD_ENAB bit to '0'), the WD_LD bit still loads the counter value and sets the counter output via the WD_VAL bit. Hence, it is possible to use the external WATCHDOG output as a digital output.

Watchdog Output Support

The external WATCHDOG output is also controlled by the Counter Control Register. Writing to the WD_OUTEN bit enables or disables this output:

```
0 = disable WATCHDOG output. Output will be high.
1 = enable WATCHDOG output. Output controlled by WD_SEL bit.
```

When the WATCHDOG output is disabled, it is set high by the hardware, since it is an active-low signal.

When it is enabled, its value depends upon the WD_SEL bit. If WD_SEL is '0', then the WATCHDOG output will be the inverse of the WATCHDOG bit in the Interrupt Status Register. Since the WATCHDOG bit in the Interrupt Status Register is set by a rising edge on the Watchdog counter output, this output selection reflects the status of the "watchdog timer". It is generally used when the Watchdog timer features are activated (WD_ACT = 1). The WATCHDOG output in this case can be used to disable power amplifiers, engage brakes, etc. when a Watchdog timer event occurs.

If WD_SEL is '1', then the WATCHDOG output reflects the output of the Watchdog counter. In this mode, the Watchdog counter can be used to generate a square wave output of programmable duty cycle.

Watchdog Preload Control

The next four bits: WD_PRSEL, WD_WSET, WD_RSET, and WD_MODE, control the manner in which the Watchdog preload registers are used. The Watchdog counter has four 32-bit preload registers organized into two sets: set #0 and set #1. Each set contains a preload low register and a preload high register.

Only one of the register sets is used at a time by the Watchdog counter. If WD_RSET is '0', then register set #0 is used. Otherwise register set #1 is used. The WD_RSET bit only controls the register set that is used when the Watchdog counter expires and reloads its value from one of the preload registers.

The WD_WSET, on the other hand, controls which register set is accessed via the Watchdog Preload Low Register and the Watchdog Preload High Register. If WD_WSET is '0', then writing to the Watchdog Preload Low Register will set the value of the preload low register in set #0. Writing to the Watchdog Preload High Register will set the value of the preload high register in set #0.

Likewise, if WD_WSET is '1', then writing to the Watchdog Preload Low Register will change the value of the preload low register in set #1, and writing to the Watchdog Preload High Register will modify the contents of the preload high register in set #1.

Two different bits are used to control the register set read on counter expiration and the register set accessed by the Watchdog Preload Low/High Registers so that the preload registers can be programmed when not in use by the counter. In other words, the preload registers may be double-buffered. For example, if WD_RSET is '0' and WD_WSET is '1', then the Watchdog counter will use register set #0 while the software sets the values of the preload registers in set #1. Then by changing WD_RSET to '1' and WD_SET to '0' in a single 32-bit write to the Counter Control Register, the Watchdog counter will begin using register set #1 the next time it expires and register set #0 may now be programmed. This functionality is useful for making smooth transitions from one duty cycle to the next in PWM mode. It can also be used to change the period of the square wave output, without unexpected glitches.

The WD_MODE bit controls whether the Watchdog counter output is used in determining the next preload register. It selects between a square wave output of 50% duty cycle or a PWM output. If WD_MODE is '0', then only one preload register in the current register set (selected by WD_RSET) is used when reloading the counter value. Hence, the output is a square wave of 50% duty cycle. This mode makes it easier (and faster) to program the Watchdog counter for squave wave output, since only one preload register is required. The preload low register in the active set is used if WD_PRSEL is '0'. Otherwise, the preload high register is used.

If WD_MODE is '1', then the preload register is determined by the counter output. If the counter output goes low, then the preload low register in the active register set is used to reload the count value. Otherwise, the preload high register is used. This mode produces a square wave output of programmable duty cycle, since the preload low register and preload high register may be set to different values.

Counter Support

Bits CT_LD and CT_VAL are used to set the current Counter value. When CT_LD is '1' on a write to the Counter Control Register, the contents of the currently active preload register are transferred to the Counter, without waiting for the counter to expire. Normally, the counter reads the contents of a preload register only when it expires.

The CT_VAL bit sets the current output value of the counter, and also determines which preload register is used. For example, if CT_VAL is '0' when CT_LD is '1', then the contents of the Counter Preload Low Register will be transferred to the Counter. If CT_VAL is '1', then the contents of the Counter Preload High Register will be used. The CT_RSET bit controls which register set is used. If the CT_LD bit is '0', then the Counter contents are left unchanged, and the CT_VAL bit is ignored.

When CT_LD is '1' during a write, all other bits in the Counter Control Register, with the exception of the CT_VAL, WD_LD and WD_VAL should be left unchanged. Failure to do so can lead to unexpected behaviour.

This feature allows the count value to be set without waiting for the counter to expire. This functionality is useful for resetting the Counter when using it as a measurement tool. The counter can be reloaded from the preload register without having to set the preload register again simply by writing CT_LD = '1' and CT_VAL = '1' to the Counter Control Register.

Furthermore, since the Watchdog counter may also be loaded in this way, it allows both counter's values to be set at exactly the same time, via a single 32-bit write to the Counter Control Register. Thus, the two counters can be synchronized.

Note that if the Counter is disabled (by setting the CT_ENAB bit to '0'), the CT_LD bit still loads the counter value and sets the counter output via the CT_VAL bit. Hence, it is possible to use the external CNTR_OUT output as a digital output.

Counter Gating

The Counter on the Q8 may be enabled or disabled by external hardware as well using the CNTR_EN input. This input is normally pulled high by an on-board pullup resister. However, if it is pulled low by external hardware, then the Counter will be disabled until it CNTR_EN goes high again (assuming CTEN_CV = 0 in the Control Register and CTEN_POL is '0').

The polarity of the CNTR_EN input may be reversed by setting the CTEN_POL bit to '1'. In this case, the CNTR_EN input will disable the Counter when high, and the Counter will be enabled when CNTR_EN is low. This mode is useful for measuring the duration of a pulse, since the Counter will only count when the pulse is high.The change in count value before and after the pulse will indicate the duration of the pulse, with 30 ns accuracy. If the input signal is periodic, and the period is known, this feature can also be used to measure the average duty cycle of the incoming signal, particularly if the Watchdog counter is used to measure the time between measurements and the two counters are synchronized.

If the CTEN_CV bit is '1' in the Control Register, then the CNTR_EN input functions instead as an external trigger for A/D conversions. In this case, the CTEN_POL bit controls whether a rising edge or falling edge on the CNTR_EN input triggers A/D conversions. If

CTEN_POL is '0', then A/D conversions are triggered on the falling edge of CNTR_EN. If CTEN_POL is '1', then A/D conversions are triggered by a rising edge on CNTR_EN.

Counter Output Support

The external $\overline{\text{CNTR\_OUT}}$ output is also controlled by the Counter Control Register. Writing to the CT_OUTEN bit enables or disables this output:

```
0 = disable CNTR_OUT output. Output will be high.
1 = enable CNTR_OUT output. Value is output of Counter.
```

When the $\overline{\text{CNTR\_OUT}}$ output is disabled, it is set high by the hardware, since it is an active-low signal.

When it is enabled, the $\overline{\text{CNTR\_OUT}}$ output reflects the output of the Counter. In this mode, the Counter can be used to generate a square wave output of programmable duty cycle.

Counter Preload Control

The next four bits: CT_PRSEL, CT_WSET, CT_RSET, and CT_MODE, control the manner in which the Counter preload registers are used. The Counter has four 32-bit preload registers organized into two sets: set #0 and set #1. Each set contains a preload low register and a preload high register.

Only one of the register sets is used at a time by the Counter. If CT_RSET is '0', then register set #0 is used. Otherwise register set #1 is used. The CT_RSET bit only controls the register set that is used when the Counter expires and reloads its value from one of the preload registers.

The CT_WSET, on the other hand, controls which register set is accessed via the Counter Preload Low Register and the Counter Preload High Register. If CT_WSET is '0', then writing to the Counter Preload Low Register will set the value of the preload low register in set #0. Writing to the Counter Preload High Register will set the value of the preload high register in set #0.

Likewise, if CT_WSET is '1', then writing to the Counter Preload Low Register will change the value of the preload low register in set #1, and writing to the Counter Preload High Register will modify the contents of the preload high register in set #1.

Two different bits are used to control the register set read on counter expiration and the register set accessed by the Counter Preload Low/High Registers so that the preload registers can be programmed when not in use by the counter. In other words, the preload registers may be double-buffered. For example, if CT_RSET is '0' and CT_WSET is '1', then the Counter will use register set #0 while the software sets the values of the preload registers in set #1. Then by changing CT_RSET to '1' and CT_SET to '0' in a single 32-bit write to the

Counter Control Register, the Counter will begin using register set #1 the next time it expires and register set #0 may now be programmed. This functionality is useful for making smooth transitions from one duty cycle to the next in PWM mode. It can also be used to change the period of the square wave output, without unexpected glitches.

The CT_MODE bit controls whether the Counter output is used in determining the next preload register. It selects between a square wave output of 50% duty cycle or a PWM output. If CT_MODE is '0', then only one preload register in the current register set (selected by CT_RSET) is used when reloading the counter value. Hence, the output is a square wave of 50% duty cycle. This mode makes it easier (and faster) to program the Counter for squave wave output, since only one preload register is required. The preload low register in the active set is used if CT_PRSEL is '0'. Otherwise, the preload high register is used.

If CT_MODE is '1', then the preload register is determined by the counter output. If the counter output goes low, then the preload low register in the active register set is used to reload the count value. Otherwise, the preload high register is used. This mode produces a square wave output of programmable duty cycle, since the preload low register and preload high register may be set to different values.

The Counter may be enabled or disabled using the CT_ENAB bit. If CT_ENAB is '0', then the Counter does not decrement. If CT_ENAB is '1', then counting is enabled. The CT_ENAB bit has not affect on the CT_LD and CT_VAL bits and their functionality.

Digital I/O Register

Byte Offset	0x24
Read/Write	read or write
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	unknown

The Digital I/O Register is used to read from the digital inputs or write to the digital outputs of the Q8 data acquisition system. The contents of this register are shown in Table 15 below. Bit *n* corresponds to digital I/O channel *n*.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
DIGIO31	DIGIO30	DIGIO29	DIGIO28	DIGIO27	DIGIO26	DIGIO25	DIGIO24	DIGIO23	DIGIO22	DIGIO21	DIGIO20	DIGIO19	DIGIO18	DIGIO17	DIGIO16	DIGIO15	DIGIO14	DIGIO13	DIGIO12	DIGIO11	DIGIO10	DIGIO09	DIGIO08	DIGIO07	DIGIO06	DIGIO05	DIGIO04	DIGIO03	DIGIO02	DIGIO01	DIGIO00

Table 15 Digital I/O Register

Reading from this register returns the values of the digital I/O lines. For any channels configured as outputs, the read will return the value of the output. For any channels configured as inputs, it will return the value of the input:

```
0 = digital input (or output) channel is low (0V)
1 = digital input (or output) channel is high (5V)
```

Writing to this register sets the values of the digital I/O lines configured as outputs:

```
0 = digital output low (0V)
1 = digital output high (5V)
```

Bits configured as digital inputs are not changed by writing to this register.

However, these output values are stored internally. If any digital inputs are reconfigured as outputs, the output values stored will appear immediately on the digital I/O lines.

For example, suppose all digital I/O lines are configured as inputs. Writing the value 0x12345678 to the Digital I/O Register will not change the value of any of the digital I/O lines. However, if the Digital Direction Register is then used to convert all the digital I/O lines to outputs, then the value 0x12345678 will appear immediately on the digital I/O lines, without rewriting the value in the Digital I/O Register.

This feature can simplify the task of using the digital I/O lines as a bidirectional data bus. It may also be used to restore the digital output values after the Watchdog Counter expires, since the Watchdog Counter only changes the Digital Direction Register.

Digital Direction Register

Byte Offset	0x28
Read/Write	**write-only**
Access Width	32 bit only
Write Access	180 ns
Read Access	210 ns
At Reset	cleared

The Digital Direction Register is used to configure the 32 digital I/O channels of the Q8 data acquisition system as either inputs or outputs. Each digital I/O line may be independently configured. The contents of this register are shown in Table 16 below. Bit *n* corresponds to digital I/O channel *n*.

31	***30***	***29***	***28***	***27***	***26***	***25***	***24***	***23***	***22***	***21***	***20***	***19***	***18***	***17***	***16***	***15***	***14***	***13***	***12***	***11***	***10***	***09***	***08***	***07***	***06***	***05***	***04***	***03***	***02***	***01***	***00***
DIGIO31	DIGIO30	DIGIO29	DIGIO28	DIGIO27	DIGIO26	DIGIO25	DIGIO24	DIGIO23	DIGIO22	DIGIO21	DIGIO20	DIGIO19	DIGIO18	DIGIO17	DIGIO16	DIGIO15	DIGIO14	DIGIO13	DIGIO12	DIGIO11	DIGIO10	DIGIO09	DIGIO08	DIGIO07	DIGIO06	DIGIO05	DIGIO04	DIGIO03	DIGIO02	DIGIO01	DIGIO00

Table 16 Digital Direction Register

Write a '1' to the bit to configure the corresponding digital I/O line as an output. Write a '0' to configure the digital channel as an input:

```
0 = configure channel as a digital input
1 = configure channel as a digital output
```

On reset, or when the Watchdog Counter expires and is activated, this register is cleared. Clearing this register causes all digital I/O lines to be configured as inputs. Since the digital I/O lines have pullup resistors, this action causes all digital I/O lines not driven by external devices to be pulled high.

A/D Register

Byte Offset	0x2c
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	180 ns
Read Access	270 ns
At Reset	unknown

The A/D Register is used to select the analog input channels to be converted and to read the results of the A/D conversions. The contents of this register for a write operation are shown in Table 17 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
												ADCSL7	ADCSL6	ADCSL5	ADCSL4													ADCSL3	ADCSL2	ADCSL1	ADCSL0

Table 17 A/D Register on Write

These bits have the same functionality as the similarly named bits in the Control Register. However, the A/D Register bits are used when the ADCxx_HS bit in the Control Register is '1', as enumerated in the Table 18 below.

Bit	*Name*	*Interpretation*	
31→20	Reserved	Currently unused. Set to zero.	
19	ADC_SL7	ADC47_HS = 0	Ignored (Control Register selects channels)
		ADC47_HS = 1	0 = Do not include A/D channel 7
			1 = Include A/D channel 7 in conversion
18	ADC_SL6	ADC47_HS = 0	Ignored (Control Register selects channels)
		ADC47_HS = 1	0 = Do not include A/D channel 6
			1 = Include A/D channel 6 in conversion

Bit	*Name*	*Interpretation*	
17	ADC_SL5	ADC47_HS = 0	Ignored (Control Register selects channels)
		ADC47_HS = 1	0 = Do not include A/D channel 5
			1 = Include A/D channel 5 in conversion
16	ADC_SL4	ADC47_HS = 0	Ignored (Control Register selects channels)
		ADC47_HS = 1	0 = Do not include A/D channel 4
			1 = Include A/D channel 4 in conversion
15→4	Reserved	Currently unused. Set to zero.	
3	ADC_SL3	ADC03_HS = 0	Ignored (Control Register selects channels)
		ADC03_HS = 1	0 = Do not include A/D channel 3
			1 = Include A/D channel 3 in conversion
2	ADC_SL2	ADC03_HS = 0	Ignored (Control Register selects channels)
		ADC03_HS = 1	0 = Do not include A/D channel 2
			1 = Include A/D channel 2 in conversion
1	ADC_SL1	ADC03_HS = 0	Ignored (Control Register selects channels)
		ADC03_HS = 1	0 = Do not include A/D channel 1
			1 = Include A/D channel 1 in conversion
0	ADC_SL0	ADC03_HS = 0	Ignored (Control Register selects channels)
		ADC03_HS = 1	0 = Do not include A/D channel 0
			1 = Include A/D channel 0 in conversion

Table 18 A/D Register bits

Setting an SL*n* bit to '1' selects the corresponding A/D channel for conversion. Setting the bit to '0' indicates that the corresponding A/D channel should not be converted. Bits 0-3 are ignored when the ADC03_HS bit in the Control Register is set to '0'. Similarly, bits 16-19 are ignored when the ADC47_HS bit in the Control Register is set to '1'.

The channels selected in the A/D Register are latched. It is not necessary to write to this register for every A/D conversion.

The contents of the A/D register on a read operation are shown in Table 19 below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
ADC47 conversion result																ADC03 conversion result															

Table 19 A/D Register on Read

There are two A/D converter chips on the Q8 card. The A/D converters have 14-bit resolution, with a ±10V input range. Each converter handles four analog channels. The ADC03 chip handles analog inputs 0 through 3. The ADC47 chip handles analog inputs 4 through 7. When a conversion is initiated, either manually or via the Counter, the A/D converters sample all selected channels simultaneously and then converts each selected channel in ascending sequence. The conversion results are stored in an internal FIFO in each A/D converter.

Reading the A/D Register reads these internal FIFOs. After the A/D conversion is complete, as signalled by the ADCxx_RDY flags in the Status Register or Interrupt Status Register, the conversion results may be read using the A/D Register. Each time the A/D Register is read, the next conversion result is returned until the last result in the FIFO is reached. Continuing to read from the A/D Register will start at the beginning of the FIFO again. Note that the 14-bit conversion results are sign-extended to 16 bits by the Q8 hardware.

When the first channel is available to be read, the ADCxx_FST bit will be set in the Status Register. The FIFO pointer is reset when conversions are initiated, so the first value read from the A/D Register is always the first conversion result.

Reading Conversion Results After All Conversions

For example, suppose channels 0, 2 and 3 are selected for conversion. Starting an A/D conversion causes all three channels to be sampled simultaneously by the track and hold circuitry of A/D converter ADC03. The ADC03_RDY flag goes low as soon as the A/D conversion is initiated. The ADC03_EOC also goes low. The state of the ADC03_FST flag is unknown at this point (it depends on the previous conversion).

The ADC03 converter converts channel 0 first and stores the result in its internal FIFO. When the conversion is complete, the ADC03_EOC flag goes high for about 120 ns. At the same time, the ADC03_FST flag goes high indicating that the first conversion result is available in the FIFO.

The ADC03 then converts channel 2 and stores the result in its internal FIFO. When the conversion is complete, the ADC03_EOC flag goes high again for about 120 ns. The ADC03_FST flag stays high because we have not read the first conversion result from the FIFO yet.

Finally, the last channel is converted and the result stored in the FIFO. The ADC03_EOC goes high for about 120 ns at the end of the conversion. The ADC03_RDY flag also goes high because the last conversion has been completed. The ADC03_FST flag remains high because the first conversion result has not yet been read from the FIFO.

Now the software reads the lower 16 bits of the A/D Register using a 16-bit read. The value returned is the 14-bit conversion result for channel 0, **sign-extended to 16 bits**. To make it easier for the programmer, the Q8 automatically sign-extends the A/D conversion results to 16 bits. As soon as the A/D Register has been read, the ADC03_FST flag goes low, indicating that the next value available in the FIFO is not the first channel converted.

Reading from the lower 16 bits of the A/D Register a second time returns the 14-bit conversion result for channel 2, sign-extended to 16 bits. A third read from the A/D Register returns the conversion result for channel 3. At this point, the ADC03_FST flag goes high because the conversion result for channel 0 is now available to be read again.

If the A/D Register is read again, then the conversion result for channel 0 is returned and the ADC03_FST flag goes low. Note that the same value is returned as before. The FIFO acts as a circular buffer until a new A/D conversion is initiated.

The ADC47 converter works analogously. Its results are also sign-extended to 16 bits. However, its results are returned in the upper 16-bits of the A/D Register.

Note that the results from both A/D converters may be read simultaneously by performing a 32-bit read on the A/D Register instead of 16-bit reads.

However, in this case, the software must ensure that **both** A/D converters have finished their conversions.

When the internal clocks are used, the A/D converters are not guaranteed to finish their conversions at the same time, even when conversions are started at the same time via the Control Register or Counter.

Thus, to ensure that the results are available for both A/D converters (in order to do a 32-bit read), both the ADC03_RDY and ADC47_RDY flags must be checked by software. Alternatively, the common clock may be used instead. If the same number of channels are being converted by each A/D chip then using the common clock ensures that the two converters will finish their conversions at exactly the same time. If not, the converter with the most channels selected is guaranteed to take longer and its RDY flag may be used.

Note that the maximum throughput is achieved by balancing the channels between ADC03 and ADC47.

For example, it is faster to read four A/D inputs by connecting two analog signals to ADC03 and two signals to ADC47 than it is to connect all four signals to ADC03. The reason is simple: while the A/D chips sample all their inputs simultaneously, the conversions are

performed sequentially by each chip. The two chips, however, operate in parallel. Hence, channel 0 can be converted at the same time as channel 4 because the two channels are connected to different A/D converters. On the other hand, channel 1 cannot be converted at the same time as channel 0 because both channels are connected to the same A/D chip. Thus, much better performance is achieved by connecting two analog inputs to channels 0 and 4, than by connecting the two inputs to channels 0 and 1.

Furthermore, if the common clock is used, the results from both A/D converters may be read simultaneously from the A/D Register by performing a 32-bit read instead of two 16-bit reads.

Reading the results from both A/D converters simultaneously using a 32-bit read of the A/D Register is much faster than two 16-bit reads.

Hence, code that is designed for optimal performance should make use of the ability to read the results from both A/D converters simultaneously in a single 32-bit read.

Reading from the 33MHz PCI bus is much slower than operating in the internal cache of a fast CPU (eg. 1 GHz) so it make sense to minimize the number of PCI bus accesses that occur, even at the cost of more CPU instructions.

Thus, it is faster to perform a 32-bit read and extract the two 16-bit results by a shift-and-mask operation, than it is to perform two 16-bit reads.

Reading Conversion Results During Conversion

It is not necessary to wait for all conversions to complete before starting to read the conversion results. Consider the same example, in which channels 0, 2 and 3 are being read. Starting an A/D conversion causes all three channels to be sampled simultaneously by the track and hold circuitry of A/D converter ADC03. The ADC03_RDY flag goes low as soon as the A/D conversion is initiated. The ADC03_EOC also goes low. The state of the ADC03_FST flag is unknown at this point (it depends on the previous conversion).

The ADC03 converter converts channel 0 first and stores the result in its internal FIFO. When the conversion is complete, the ADC03_EOC flag goes high for about 120 ns. At the same time, the ADC03_FST flag goes high indicating that the first conversion result is available in the FIFO.

If the software polls the Interrupt Status Register or enables interrupts on the ADC03_EOC pulse then it can read the first conversion result as soon as the ADC03_EOC flag goes high. Reading the lower 16 bits of the A/D Register returns the first conversion result and clears the ADC03_FST flag of the Status Register.

The same 16 bits of the A/D Register should not be read again until the EOC flag indicates that the next conversion result is available. Attempting to read it early will cause the next conversion result to be skipped.

If the software interrupted on the ADC03_EOC signal then the interrupt handler must clear the ADC03_EOC status bit in the Interrupt Status Register. It does so by writing a '1' to the ADC03_EOC bit in the Interrupt Status Register.

If the software is polling the Interrupt Status Register then it must clear the ADC03_EOC status bit as soon as it detects it high. It does so by writing a '1' to the ADC03_EOC bit in the Interrupt Status Register.

Using the Status Register to poll the ADCxx_EOC bit is not recommended because the pulse is so short it could easily be missed.

The ADC03_EOC bit in the Interrupt Status Register, on the other hand, is set by a rising edge of the ADC03_EOC signal. Hence, the end-of-conversion flag cannot be missed. Since conversions take at least 2.4 µs, device driver software should have no problem clearing the status bit and polling for the next end-of-conversion before it takes place.

However, software should not depend on catching all the ADCxx_EOC pulses. Instead, check the ADCxx_RDY flag to know when all conversions are complete.

In other words, don't count the number of ADC03_EOC pulses to know when all selected A/D channels have been converted. Instead, use the ADC03_RDY flag to know when all the conversion results are available and read any remaining conversion results when this flag goes high.

Once channel 0 has been converted, the ADC03 then converts channel 2 and stores the result in its internal FIFO. When the conversion is complete, the ADC03_EOC flag goes high again for about 120 ns. The ADC03_FST flag is low because the result for channel 0 has already been read.

The software can now read the conversion result for channel 2, while the A/D converter starts converting channel 3. When channel 3 has been converted, the result is stored in the FIFO. The ADC03_EOC goes high for about 120 ns at the end of the conversion. The ADC03_RDY flag also goes high because the last conversion has been completed.

Now the software reads the final conversion result by reading the lower 16-bits of the A/D Register. As soon as the A/D Register has been read, the ADC03_FST flag goes high, indicating that the next value available in the FIFO is again the first channel converted.

The ADC47 converter works analogously, but its results are available in the upper 16 bits of the A/D Register.

If both A/D converters are being used, it is more efficient to perform a 32-bit read of the A/D Register than two 16-bit reads.

Note however, that performing a 32-bit read of the A/D Register will update the position of the internal FIFOs of both A/D chips.

Thus, the software must be designed accordingly. For example, since the internal clocks of the two A/D chips have a 150 ns period, the conversion results are not likely to be ready more than 150 ns apart. Hence, software that is polling could check for both EOC flags being set and then perform a 32-bit read of the A/D Register to get both results. This technique is likely to yield the best polling performance, since conversion results are read while the next conversion is taking place in both chips.

If interrupts on EOC are used instead of polling, the interrupt handler could read the Interrupt Status Register to determine which A/D chip caused the interrupt. If both results are available, it could read the two results immediately. Otherwise, it could set a flag indicating that one result was available, clear the corresponding bit in the Interrupt Status Register and exit. When the second A/D chip finished its conversion, the interrupt handler could then read both A/D conversion results from the A/D Register.

Encoders

There are four encoder chips in the Q8 data acquisition system. Each encoder chip handles two single-ended encoder channels. The first encoder chip, dubbed ENC01, handles encoder channels 0 and 1. Similarly, ENC23 controls channels 2 and 3, ENC45 controls channels 4 and 5 and ENC67 handles channels 6 and 7.

The encoders have a byte-wide data bus. Each encoder chip is mapped to a different byte within the Encoder Data Registers and the Encoder Control Registers. ENC01 is mapped to the least significant byte (bits 0-7), ENC23 is mapped to bits 8-15, ENC45 is mapped to bits 16-23 and ENC67 is mapped to bits 24-31. These bytes may be accessed individually using 8-bit accesses, in pairs as a 16-bit access or all at once, using a 32-bit access.

Encoder Data Register A and Encoder Control Register A reference the even-numbered channels: 0, 2, 4 and 6. Encoder Data Register B and Encoder Control Register B reference the odd-numbered encoder channels: 1, 3, 5 and 7. The least significant byte of all four of these registers accesses the same encoder chip: ENC01. Similarly, the next significant byte (bits 8-15) of each of these registers accesses ENC23. ENC45 and ENC67 are likewise common to all of these registers. The address simply determines whether data or control information is being accessed and whether the even or odd-numbered channel is being accessed. Since each byte within an Encoder Data Register or Encoder Control Register accesses a different encoder chip, each byte will be referred to as an Encoder Data Byte or En-

coder Control Byte respectively. Since each encoder chip is identical, the functionality of all the Encoder Data Bytes and all the Encoder Control Bytes are the same, except that they operate on different encoder channels.

Each encoder chip maintains two 24-bit counters: one for the even-numbered channel and one for the odd-numbered channel. Two 24-bit preload registers (PR) are also available in each encoder chip, one per channel. Various status flags and internal configuration registers are also maintained for each of the two channels. Each encoder chip maintains two of the following registers, one for each channel: a Status FLAG Register, a Reset and Load Signal Decoders Register (RLD), a Counter Mode Register (CMR), an Input/Output Control Register (IOR) and an Index Control Register (IDR). They are accessed using the Encoder Control Registers. The Status FLAG Register is 8 bits wide. The other registers are 5 bits wide, with bits 5 and 6 in the byte used to select between the RLD, CMR, IOR and IDR registers, and bit 7 indicating whether the registers for both channels should be modified simultaneously. These encoder configuration registers are discussed in more detail in the Encoder Control Register sections.

Note that the contents of the encoder registers are indeterminate on power-up. Hence, all the encoder registers (RLD, CMR, IOR and IDR) must be programmed prior to using the encoders for the first time after power-up.

Encoder Data Register A

Byte Offset	0x30
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers or as four 8-bit registers
Write Access	240 ns
Read Access	300 ns
At Reset	unknown

Encoder Data Register A is used to read the output latches of the even-numbered encoder channels or to set the count value of the even-numbered encoder channels. The contents of this register are shown in Table 20 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
ENC6_DATA								ENC4_DATA								ENC2_DATA								ENC0_DATA							

Table 20 Encoder Register A

As discussed in section Encoders above, there are four encoder chips in the Q8 data acquisition system, with each encoder chip handling two single-ended encoder channels. The first encoder chip, dubbed ENC01, handles encoder channels 0 and 1. Similarly, ENC23 controls channels 2 and 3, ENC45 controls channels 4 and 5 and ENC67 handles channels 6 and 7. The encoders have a byte-wide data bus. Each encoder chip is mapped to a different byte within the Encoder Data Register. Each byte is called an Encoder Data Byte. These Encoder Data Bytes have been labelled ENC0_DATA, ENC2_DATA, ENC4_DATA and ENC6_DATA within Encoder Data Register A, since Encoder Register A only refers to the even-numbered channels of these four encoder chips.

All four encoder chips may be read and written simultaneously via Encoder Data Register A., as shown in the above table. Encoder Data Register A is used for the first, even-numbered, channel of each encoder chip. Encoder Data Register B accesses the second, odd-numbered, channel of each encoder chip.

The encoder chips contain two 24-bit counter, one per channel. Each counter increments or decrements according to the direction of motion. In order to read the 24-bit count values, the counter value must be transferred to an output latch within the encoder chip. This operation is performed for the even-numbered channels using Encoder Control Register A. The 24-bit latched value may then be read by performing three consecutive reads of the Encoder Data Byte. Each read returns the next byte in the 24-bit value. A byte pointer is maintained within the encoder chip for each channel to keep track of which byte of the 24-bit value has been read. The least significant byte is returned first.

This byte pointer must be reset before reading the first byte of the result. Resetting the byte pointer may be done using Encoder Control Register A. Refer to Encoder Control Register A for more details.

Writing to Encoder Data Register A changes the 24-bit preset register for each even-numbered encoder channel. The preset register is used to set the count value or the filter clock prescaling. The 24-bit value is written to the encoder preset register by three consecutive writes to Encoder Register A. The least significant byte should be written first.

The byte pointer must be reset before writing the first byte of the preset value. Resetting the byte pointer may be done using Encoder Control Register A.

Encoder Register A may be accessed as one 32-bit register, two 16-bit registers or four 8-bit registers. Accessing the registers as four 8-bit registers is easier for programming but does not yield the maximum performance. Using 32-bit accesses is much more efficient.

However, because the byte pointers are incremented each time an encoder is accessed, performing a 32-bit access increments the byte pointers of all four encoder chips at the same time.

Hence, it is generally best to always use the same access width when reading or writing to Encoder Register A. For example, reading the least significant byte alone and then reading the entire 32-bit register results in the byte pointer for encoder channel 0 being incremented twice, while the byte pointers of encoder channels 1 through 3 are incremented once. Such mixed accesses are not recommended unless the byte pointers are reset in-between changing access widths.

Encoder Data Register B

Byte Offset	0x34
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers or as four 8-bit registers
Write Access	240 ns
Read Access	300 ns
At Reset	unknown

Encoder Data Register B is used to read the output latches of the odd-numbered encoder channels or to set the count value of the odd-numbered encoder channels. The contents of this register are shown in Table 21 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
ENC7_DATA								ENC5_DATA								ENC3_DATA								ENC1_DATA							

Table 21 Encoder Register B

Encoder Data Register B has similar functionality to Encoder Data Register A. Refer to the discussion of Encoder Data Register A for details.

Encoder Control Register A

Byte Offset	0x38
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers or as four 8-bit registers
Write Access	240 ns
Read Access	300 ns
At Reset	unknown

Encoder Control Register A is used to configure the even-numbered encoder channels. It may also be used to configure both even and odd channels simultaneously. The contents of this register are shown in Table 22 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
ENC6_CONTROL								ENC4_CONTROL								ENC2_CONTROL								ENC0_CONTROL							

Table 22 Encoder Control A

Each encoder chip has a variety of internal configuration registers for setting up the counter modes, filter clock prescaling, flag outputs, index control and other features of the encoders. Encoder Control Register A allows these internal registers to be configured for all the even-numbered encoder channels simultaneously when accessed as a 32-bit register. It can also be used to configure all channels – both even and odd – simultaneously, or a combination thereof (eg. Channels 0 and 1, 2, 4, 6 and 7).

Consider each byte within Encoder Control Register A. Each byte represents the internal control registers of an encoder chip. For example, bits 0-7, labelled ENC0_CONTROL, represent the internal control registers for encoder ENC01. Let each byte be called an Encoder Control Byte. Thus, ENC0_CONTROL is the Encoder Control Byte for ENC01, the encoder chip that handles channels 0 and 1. Similarly, ENC2_CONTROL is the Encoder Control Byte for ENC23, ENC4_CONTROL is the Encoder Control Byte for ENC45 and ENC6_CONTROL is the Encoder Control Byte for ENC67. Since each Encoder Control Byte has exactly the same format, the following subsections discuss a single Encoder Control Byte.

Reading from an Encoder Control Byte returns the 8-bit Status FLAG Register for the encoder as discussed below.

Writing to the Encoder Control Byte configures one of the 5-bit internal encoder control registers. The different internal registers are selected using the upper three bits of the Encoder Control Byte, as shown in Table 23 below.

Bit	*Name*	*Interpretation*
7	XY_SEL	0 = even or odd channel only
		1 = both even and odd channel simultaneously
6→5	REG_SEL	00 = Reset and Load Signal Decoders Register (RLD)
		01 = Counter Mode Register (CMR)
		10 = Input/Output Control Register (IOR)
		11 = Index Control Register (IDR)

Table 23 Register Selection Bits of an Encoder Control Byte

When the most significant bit of the Encoder Control Byte is zero, the write operation configures the internal registers for the even-numbered channel only. If Encoder Control Register B is being accessed, then a zero in the most-significant bit means that the write operation configures the odd-numbered channel only.

If the most significant bit is one, then the write operation configures the internal registers for both the even and odd numbered channel at the same time.

For example, writing the value 0x00 to the ENC2_CONTROL byte in Encoder Control Register A will access the RLD register for encoder channel 2 only. Writing the value 0x80 to ENC2_CONTROL will access the RLD registers for encoder channels 2 and 3 simultaneously.

Writing 0x00 to the ENC3_CONTROL byte in Encoder Control Register B accesses the RLD register for encoder channel 3 only. Writing the value 0x80 to ENC3_CONTROL accesses the RLD registers of encoder channels 2 and 3 at the same time. Hence, writing 0x80 to ENC3_CONTROL has the same effect as writing 0x80 to ENC2_CONTROL – the RLD registers for encoder channels 2 and 3 are accessed simultaneously.

Note that the REG_SEL bits in Table 23 are listed with bit 6 first, followed by bit 5. Hence, writing 0xc0 to ENC0_CONTROL accesses the Input/Output Control Registers for channel 0 and 1 simultaneously.

Status FLAG Register

Reading from an Encoder Control Byte returns the 8-bit Status FLAG Register for the encoder. The bits of the Status FLAG Registers are enumerated in Table 24 below.

Bit	*Name*	*Interpretation*
7	Reserved	Currently unused. Returns zero.

Bit	*Name*	*Interpretation*
6	Index Flag (IDX)	0 = not at index position
		1 = at index position
5	Up/Down Flag (U/D)	0 = counting down
		1 = counting up
4	Error Flag (E)	0 = no errors
		1 = excessive noise present at count inputs
3	Sign Flag (S)	Reset to zero when the counter (CNTR) overflows
2	Compare Toggle (CPT)	Toggles every time the counter equals the preload register value (CNTR = PR)
1	Carry Toggle (CT)	Toggles every time the counter (CNTR) overflows
0	Borrow Toggle (BT)	Toggles every time the counter (CNTR) underflows

Table 24 Status FLAG Register bits

Reset and Load Signal Decoders Register (RLD)

The RLD Register is used to reset status flags, reset the byte pointer, and to transfer values to the counter, output latch or filter clock prescaler. The contents of this register are shown .

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
XY_SEL	0	0	TRANSFER		RST_FLAGS		RST_BP

Table 25 Reset and Load Signal Decoders Register (RLD)

The bits of the RLD register are defined in Table 26 below. Note that paired bits are listed with the higher bit first. Hence, writing the value 0x14 transfers the counter to the output latch while resetting the borrow, carry, compare and sign flags in the Status FLAG Register.

Bit	*Name*	*Interpretation*
7	XY_SEL	0 = Even or odd channel only
		1 = Both even and odd channel simultaneously
6→5	REG_SEL	00 = Reset and Load Signal Decoders Register (RLD)

Bit	*Name*	*Interpretation*
4→3	TRANSFER	00 = Do not perform any transfers
		01 = Transfer preload register (PR) to counter (CNTR)
		10 = Transfer counter (CNTR) to output latch (OL)
		11 = Transfer preload byte 0 (PR0) to prescaler (PSC)
2→1	RST_FLAGS	00 = Do not reset flags or counter
		01 = Reset counter (CNTR)
		10 = Reset borrow (BT), carry (CT), compare (CPT) and sign (S) flags
		11 = Reset error flag (E)
0	RST_BP	0 = Do not reset byte pointer (BP)
		1 = Reset byte pointer (BP)

Table 26 Reset and Load Signal Decoder Register (RLD) bits

The TRANSFER bits are used to set the encoder count value via the preload register, to latch the current count value into the output latch, or to set the filter prescaler value.

The encoder count value cannot be read directly. It must be latched first. The contents of the output latch may be read by reading the corresponding Encoder Data Byte.

The RST_FLAGS are used to reset the encoder count value and clear status flags. The status flags may be read from the Status FLAG Register, described above.

The byte pointer is used when reading from or writing to an Encoder Data Byte in one of the Encoder Data Registers. The byte pointer indicates which byte of the 24-bit value is currently being read or written. The byte pointer must be reset prior to reading or writing a 24-bit value from the Encoder Data Registers.

Counter Mode Register (CMR)

The CMR Register is used to set the operational mode of the encoder counters. The contents of this register are shown in Table 27 below.

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
XY_SEL	0	1	QUADRATURE		MODE		BCD

Table 27 Counter Mode Register (CMR)

The bits of the CMR register are defined in Table 28 below. Note that paired bits are listed with the higher bit first. Hence, writing the value 0x38 puts the counter in normal 4X quadrature, binary counting, mode.

Bit	*Name*	*Interpretation*
7	XY_SEL	0 = Even or odd channel only
		1 = Both even and odd channel simultaneously
6→5	REG_SEL	01 = Counter Mode Register (CMR)
4→3	QUADRATURE	00 = Non-quadrature
		01 = 1X
		10 = 2X
		11 = 4X
2→1	MODE	00 = Normal mode
		01 = Range limit
		10 = Non-recycle
		11 = Modulo-N
0	BCD	0 = Count in binary
		1 = Count in BCD

Table 28 Counter Mode Register (CMR) bits

The QUADRATURE bits select whether the A and B inputs will be treated as count (CNT) and direction (DIR) inputs, or as quadrature inputs. The non-quadrature mode selects the inputs to the CNT and DIR. The quadrature modes select the quadrature multiplying factor. The 4X quadrature mode is typically used to get the maximum resolution out of the encoder at four times the encoder line count.

Note that the index function in the Index Control Register *must* be disabled in non-quadrature count mode.

The MODE bits select the counter mode. In normal mode, the counter wraps from 0 around to 0x0ffffff when decrementing and wraps from 0x0ffffff to 0 when incrementing. In other words, it acts like a normal 24-bit counter, wrapping around on overflow or underflow.

In range-limit mode, the counter freezes at the limits instead of wrapping around. It does not resume counting until the counter direction is reversed. The counter will not decrement below zero, and counting will only resume once the count direction is upward again. Similarly, the counter will not increment past the preload register (PR) value. It will remain at the PR value until the count direction is downward again.

In non-recycle mode, the counter is disabled when a counter overflow (count increments to 0x0ffffff) or underflow (count decrements to 0) would take place. Unlike the range-limit mode, the upper limit is not defined by the preload register and changing the count direction will not resume counting. Instead, a reset or load operation must be performed on the

counter to resume operation. Note that a Carry is generated when the counter would overflow, and a Borrow is produced when the counter would underflow.

In module-N mode, the counter acts just like it does in normal mode except that it wraps between 0 and the value of the preload register (PR) instead of 0x0ffffff. The module-N mode may be used for frequency division, since the divide-by-N output frequency is available at the Carry or Borrow flag outputs.

The BCD bit determines whether the counter is treated as a binary counter or a BCD counter. BCD counting is generally not used in control applications, but may be useful in other situations, particularly when floating-point operations are not used.

Input/Output Control Register (IOR)

The IOR Register configures the output flags from the encoder chips, as well as the action taken in response to a change in the I/LD input. The contents of this register are shown in Table 29 below.

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
XY_SEL	1	0	FLAGS		0	LOAD	AB_EN

Table 29 Input/Output Control Register (IOR)

The bits of the IOR register are defined in Table 30 below. Note that paired bits are listed with the higher bit first. Hence, writing the value 0x4B sets FLG1 to COMPARE and FLG2 to BORROW, and latches the counter value when an index pulse occurs.

Bit	*Name*	*Interpretation*
7	XY_SEL	0 = Even or odd channel only
		1 = Both even and odd channel simultaneously
6→5	REG_SEL	01 = Input/Output Control Register (IOR)
4→3	FLAGS	00 = FLG1 is CARRY output, FLG2 is BORROW output
		01 = FLG1 is COMPARE output, FLG2 is BORROW output
		10 = FLG1 is CARRY/BORROW output, FLG2 is U/D output
		11 = FLG1 is IDX, FLG2 is E (IDX and E are bits in FLAG register)
2	0	Not used
1	LOAD	0 = I/LD input loads counter value (CNTR) from preload register (PR)
		1 = I/LD input transfers counter value (CNTR) to output latch (OL)

Bit	*Name*	*Interpretation*
0	AB_EN	0 = Disable A and B inputs
		1 = Enable A and B inputs

Table 30 Input/Output Control Register (IOR) bits

The FLAGS bits select the meaning of the FLG1 and FLG2 outputs for the encoder channel. These output flags (FLG1 and FLG2) may be read in the Q8 Status Register and may also be used as interrupt sources (see the Q8 Interrupt Status Register). Hence, it is possible to generate an interrupt when the counter overflows or underflows, or when the index pulse is detected, for example. An interrupt may also be generated when an error occurs due to excessive noise on the A and B inputs.

Note that the CARRY, BORROW, CARRY/BORROW, COMPARE and IDX flags are only asserted for a brief amount of time. Hence, polling the Q8 Status Register may not be fast enough to catch these signals.

However, these same flags will set bits in the Interrupt Status Register on their rising edges, so the Interrupt Status Register may be used to poll these flags without risk of missing a transition. The encoder Status FLAG register for the channel of interest may also be read to catch these transitions, because the encoder's FLAG register bits toggle on carry, borrow and compare.

The LOAD bit configures the action of the index/load (I/LD) input. The index input can be used to load the count value (CNTR) from the preload register (PR) or to latch the count value into the output latch (OL).

If the index function is disabled in the encoder's Index Control Register (IDR), then the index input is asynchronous and level-sensitive. A low value at the I/LD input will perform the specified action as long as it is low.

If the index function is enabled in the encoder's Index Control Register (IDR), then the index input is synchronous with the quadrature clocks. The I/LD input will perform the requested action according to the setting of the index polarity bit in the IDR.

To have the hardware ignore the index input altogether, disable the index in the Q8 Control Register. When disabled in the Control Register, the index input to the encoder chip will always be high. See the Control Register description on page 47 for details. The ability to disable the index altogether is useful after a calibration sequence has been performed to home the device being controlled, for example.

The AB_EN bit is used to enable or disable the A and B inputs of the encoder. Disabling the A and B inputs prevents the counter from counting. However, it is still possible to set the count value via software.

Index Control Register (IDR)

The IDR Register configure the index input (I/LD) for each encoder. The contents of this register are shown in Table 31 below.

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
XY_SEL	1	1	0	0	0	POL	IDX_EN

Table 31 Index Control Register (IDR)

The bits of the IDR register are defined in Table 32 below.

Bit	*Name*	*Interpretation*
7	XY_SEL	0 = Even or odd channel only
		1 = Both even and odd channel simultaneously
6→5	REG_SEL	01 = Index Control Register (IDR)
4→2	0	Not used
1	POL	0 = Negative index polarity (active-low)
		1 = Positive index polarity (active-high)
0	IDX_EN	0 = Disable index mode ie, I/LD input is asynchronous
		1 = Enable index mode ie, I/LD input is synchronous

Table 32 Index Control Register (IDR) bits

The POL bit sets the polarity of the index input. The IDX_EN input enables or disables the index mode for the I/LD input. If IDX_EN is '0', then the I/LD input is treated as a level-sensitive, asynchronous input. If IDX_EN is '1', then the I/LD input functions as a level-sensitive input synchronous with the quadrature clocks.

Disabling the index mode does not disable the I/LD input. To disable the I/LD input, use the Q8 Control Register. Also, in non-quadrature count mode, the index mode *must* be disabled.

Encoder Control Register B

Byte Offset	0x3c
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers or as four 8-bit registers
Write Access	240 ns
Read Access	300 ns
At Reset	unknown

Encoder Control Register B is used to configure the odd-numbered encoder channels. It may also be used to configure both even and odd channels simultaneously. The contents of this register are shown in Table 33 below.

31 30 29 28 27 26 25 24	*23 22 21 20 19 18 17 16*	*15 14 13 12 11 10 09 08*	*07 06 05 04 03 02 01 00*
ENC6_CONTROL	ENC4_CONTROL	ENC2_CONTROL	ENC0_CONTROL

Table 33 Encoder Control B

Each encoder chip has a variety of internal configuration registers for setting up the counter modes, filter clock prescaling, flag outputs, index control and other features of the encoders. Encoder Control Register B allows these internal registers to be configured for all the odd-numbered encoder channels simultaneously when accessed as a 32-bit register. It can also be used to configure all channels – both even and odd – simultaneously, or a combination thereof (eg. Channels 0 and 1, 2, 4, 6 and 7). Refer to the description of Encoder Control Register A on page 90 for details.

D/A Output Register A

Byte Offset	0x40
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Output Register A controls D/A channels 0 and 4. The contents of this register for a read or write operation are shown in Table 34 below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
				DAC4_DATA																DAC0_DATA											

Table 34 D/.A Output Register A

This register may be accessed as a single 32-bit register, or as two 16-bit registers. The DAC*i*_DATA bits are 12-bit values representing the analog output voltage. The meaning of the 12-bit value depends upon the analog output mode. The analog outputs support three different modes, per channel:

1. Bipolar, 10V mode, in which the output range is -10V to +10V–1LSB (9.9951171875V).
2. Bipolar, 5V mode, in which the output range is -5V to +5V-1LSB (4.99755859375V).
3. Unipolar, 10V mode, in which the output range is 0V to 10V.

The relationship between the 12-bit DAC*i*_DATA value and the output voltage is enumerated in Table 35 below.

Mode	*Relationship*
-10V → +10V	000000000000 = -10V
	100000000000 = 0V
	111111111111 = +10V-1LSB (9.9951171875V)
-5V → +5V	000000000000 = -5V
	100000000000 = 0V
	111111111111 = +5V-1LSB (4.99755859375V)
0V → +10V	000000000000 = 0V
	100000000000 = 5V
	111111111111 = +10V-1LSB (9.9951171875V)

Table 35 Analog Output Modes

Note that the 12-bit values are not twos-complement binary numbers. On reset, all the DAC registers are initialized to zero. **Hence, the DAC comes out of reset in unipolar, 10V mode and the analog output is zero volts.**

The same is true when the watchdog feature is activated. The (possible) change in mode must be taken into account when recovering from a watchdog event.

The D/A Output A Register may be read or written. On a read, it simply returns the current contents of the DAC output latches. After a reset or watchdog event, the value read will be zero. The shaded bits in Table 34 will be zero, since they are unused.

Writing to the D/A Output A Register sets the D/A output latch for channels 0 and 4. However, this value does not appear immediately at the analog outputs unless transparent mode is enabled. Instead, a second write to the D/A Update Register is required to transfer the contents of the output latch to the analog outputs.

This two-stage process allows all the analog outputs to be updated simultaneously, since a single write to the D/A Update Register can update all eight analog outputs at the same time.

If simultaneous updating of the outputs is not required, transparent mode may be enabled in the Q8 Control Register. When transparent mode is enabled, the analog outputs are updated as soon as a write to the D/A Output Register occurs.

D/A Output Register B

Byte Offset	0x44
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Output Register B controls D/A channels 1 and 5. The contents of this register for a read or write operation are shown in Table 36 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				DAC5_DATA																DAC1_DATA											

Table 36 D/.A Output Register B

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Refer to the description of D/A Output Register A for details.

D/A Output Register C

Byte Offset	0x48
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Output Register C controls D/A channels 2 and 6. The contents of this register for a read or write operation are shown in Table 37 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				DAC6_DATA																DAC2_DATA											

Table 37 D/.A Output Register C

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Refer to the description of D/A Output Register A for details.

D/A Output Register D

Byte Offset	0x4c
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Output Register D controls D/A channels 3 and 7. The contents of this register for a read or write operation are shown in Table 38 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				DAC7_DATA																DAC3_DATA											

Table 38 D/.A Output Register D

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Refer to the description of D/A Output Register A for details.

D/A Update Register

Byte Offset	0x50
Read/Write	write-only
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Update Register is used to update the analog outputs with the values written to the D/A output latches via the D/A Output A, B, C and D Registers. It is a write-only register. The contents of this register are shown in Table 39 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				DAC47_UPDATE																DAC03_UPDATE											

Table 39 D/.A Update Register

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Writing any value to the DAC03_UPDATE bits causes analog output channels 0 through 3 to be updated with the values in their respective output latches. Similarly, writing any value to the DAC47_UPDATE bits causes analog output channels 4 through 7 to be updated.

If the contents of the D/A output latches have not been changed since the last update then the corresponding analog outputs will show no change either. Hence, in most cases a full 32-bit write to the D/A Update Register is used to update the analog outputs.

Note that if transparent mode is enabled, then writing to the D/A Update Register is unnecessary, since the analog outputs are updated automatically as soon as the D/A Output Register is changed.

D/A Mode Register

Byte Offset	0x6c
Read/Write	read or write
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Mode Register is used to set the analog output modes for each of the eight channel. The contents of this register are shown in Table 40 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				GAIN4	GAIN5	GAIN6	GAIN7	MODE4	MODE5	MODE6	MODE7									GAIN0	GAIN1	GAIN2	GAIN3	MODE0	MODE1	MODE2	MODE3				

Table 40 D/A Mode Register

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Each analog output channel has two bits defining its mode: GAIN*i* and MODE*i*, where *i=0..7*.. The meaning of these bits is defined in Table 41 below.

GAINi	*MODEi*	*Interpretation*	*Range*
0	0	Unipolar, 10V mode	0V to 10V
0	1	Bipolar, 5V mode	-5V to +5V
1	0	Undefined, do not use	
1	1	Bipolar, 10V mode	-10V to +10V

Table 41 D/A modes

Note the location of these bits carefully. The order of the channels does not increase with the bit number, as might be expected.

Writing to the D/A Mode Register sets the mode latches, but does not update the actual mode used for the analog outputs. Unless transparent mode is enabled, a separate write to the D/A Mode Update Register is required to update the actual analog output modes.

Transparent mode may be enabled via the Q8 Control Register. Note, however, that transparent mode affects the operation of the D/A Output A, B, C and D Registers as well.

D/A Mode Update Register

Byte Offset	0x70
Read/Write	write-only
Access Width	32 bits or as two 16-bit registers
Write Access	300 ns
Read Access	420 ns
At Reset	Reset to zero

The D/A Mode Update Register is used to update the mode of the analog outputs with the values written to the D/A mode latches via the D/A Mode Register. It is a write-only register. The contents of this register are shown in Table 42 below.

31	*30*	*29*	*28*	*27*	*26*	*25*	*24*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*
				DAC47_MODE_UPDATE																DAC03_MODE_UPDATE											

Table 42 D/.A Mode Update Register

This register may be accessed as a single 32-bit register, or as two 16-bit registers. Writing any value to the DAC03_MODE_UPDATE bits causes the modes of analog output channels 0 through 3 to be updated with the values in their respective mode latches. Similarly, writing any value to the DAC47_MODE_UPDATE bits causes the modes of analog output channels 4 through 7 to be updated.

If the contents of the D/A mode latches have not been changed since the last update then the corresponding analog outputs will show no change either. Hence, in most cases a full 32-bit write to the D/A Mode Update Register is used to update the analog output modes.

Note that if transparent mode is enabled, then writing to the D/A Mode Update Register is unnecessary, since the analog output modes are updated automatically as soon as the D/A Mode Register is changed.

Register-Level Programming Examples

This chapter provides examples of programming the Q8 card at the register level. Register-level programming can only be performed from a device driver or a real-time kernel with PCI hardware support. The registers cannot be programmed from a Win32 application directly. Refer to the Q8 Programming chapter for information on how to program the Q8 data acquisition system using the drivers provided. This chapter is only included for those users who wish to write their own device drivers for operating systems or real-time kernels that are not currently supported by Quanser.

The Q8 is a memory-mapped, 32-bit PCI card. That means that the card is assigned a range of consecutive addresses in the computer's physical memory. The first address in this range is called the "base address". The register offsets given in the previous chaper are all relative to this base address. For example, if the base address is 0xEC000000 then the card's registers will occupy the address range 0xEC000000 to 0xEC000400.

In a plug-and-play operating system, the operating system itself finds a suitable range of addresses for the card, and programs the card with that address. If the operating system is not plug-and-play, then the BIOS usually performs this address assignment.

Before using the card, the base address assigned to the Q8 card must be determined, and then mapped into the virtual memory space of the driver or real-time application. To locate the card, the identification information (vendor ID, device ID, etc.) listed in section Card Identification on page 39 is required. All of the examples assume that these operations have been done, and that the *pQ8* variable contains the virtual memory address of the card, where *pQ8* is defined as:

```
PQ8Registers pQ8;
```

where `PQ8Registers` is defined in the header file Q8.h. The header files inserted in the Appendix are assumed to be included in all the examples.

Configuring the Digital I/O

On reset, all the digital I/O channels are configured as inputs. Since each channel has a pullup resistor, the digital I/O channels are always high after a reset. To configure whether a digital I/O channel is an input or an output, write to the Digital Direction Register. Each bit corresponds to a digital I/O channel. A '0' bit in this register programs a digital channel as an input. A '1' bit makes the channel an output. Once a channel has been configured using the Digital Direction Register, it need not be configured again unless a active watchdog event has occurred (WD_ACT bit set in Counter Control Register and watchdog counter period expires).

Configuring All Channels as Inputs

For example, to configure all channels as inputs:

```
WriteDWordToHwMem(&pQ8->digitalDirection, 0x00000000L);
```

Configuring All Channels as Outputs

To configure all channels as outputs:

```
WriteDWordToHwMem(&pQ8->digitalDirection, 0xffffffffL);
```

Configuring 0-7 as Outputs, 8-15 as Inputs

To configure channels 0-7 as outputs and channels 8-15 as inputs:

```
WriteDWordToHwMem(&pQ8->digitalDirection, 0x0000000fL);
```

Performing Digital I/O

To read the digital inputs, simply read the Digital I/O Register. Bits corresponding to digital output channels will simply return the current digital output value. To write to the digital outputs, simply write to the Digital I/O Register.

Reading channels 8 and 10

For example, to read digital inputs 8 and 10:

```
uint32_T  channels   = ReadDWordFromHwMem(&pQ8->digitalIO);
boolean_T channel_8  = ((channels & 0x0100) != 0);
boolean_T channel_10 = ((channels & 0x0400) != 0);
```

Note that all 32 digital channels are read simultaneously by the first line. The last two lines simply extract the value of bits 8 and 10 – the bits corresponding to channels 8 and 10. This example assumes that channels 8 and 10 have been configured as digital inputs, by writing to the Digital Direction Register.

Writing to channels 0 and 2

To write a value of '0' to digital output 0 and a value of '1' to digital output 2:

```
WriteDwordToHwMem(&pQ8->digitalIO, 0x00000004L);
```

This example assumes that all other digital I/O channels are configured as inputs – in which case the value of the corresponding bit is ignored. If there are other digital outputs, then this example would set them to '0', since all 32 digital I/O channels are written simultaneously.

If you only wish to write to digital outputs 0 and 2 without modifying any other outputs, the following code may be used:

```
uint32_T outputs = ReadDWordFromHwMem(&pQ8->digitalIO) & 0xfffffffaL;
WriteDWordToHwMem(&pQ8->digitalIO, outputs | 0x00000004L);
```

The first line reads the current output values (and inputs) and masks out bits 0 and 2 – the two channels we wish to write. The second line then writes to the digital outputs, adding the desired values for channels 0 and 2.

While this second example works, it is not the most efficient because the 33 MHz PCI bus is not as fast as the system bus. Hence, it would be more efficient to maintain the current output values in a global variable instead. In this case, the code might look like:

```
current_outputs = (current_outputs & 0xfffffffaL) | 0x00000004L;
WriteDWordToHwMem(&pQ8->digitalIO, current_outputs);
```

Configuring the Analog Outputs

On reset, the analog outputs are in unipolar, 10V mode and the analog outputs are 0V. To configure the analog outputs to be bipolar, the analog output mode must be changed. Note that when the mode is changed, it is also a good idea to reprogram the analog output value as well, since changing the mode typically changes the output voltage at the same time. For example, if the analog output latch contains 0x000 and the mode is unipolar, 10V, then the analog output will be zero. Changing the mode to bipolar, 10V mode causes the analog output to jump to -10V, because a code of 0x000 corresponds to -10V in bipolar, 10V mode. Hence, in all these configuration examples, the analog outputs will be programmed to be 0V as well.

Configuring all channels as bipolar, 10V

To configure all analog output channels as bipolar, 10V, change and update the analog mode. Also set the analog output value to ensure it is at the desired value – do not assume that the output will be 0V after changing the mode.

```
/* Program analog modes to bipolar, 10V */
WriteDWordToHwMem(&pQ8->analogMode.all, 0x0ff00ff0L);

/* Program analog outputs to bipolar zero */
for (i=0; i < 4; i++)
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i], 0x08000800L);

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The constants defined in Q8.h may also be used, for greater readability:

```
/* Program analog modes to bipolar, 10V */
WriteDWordToHwMem(&pQ8->analogMode.all, DAC0_BIPOLAR_10V
          | DAC1_BIPOLAR_10V | DAC2_BIPOLAR_10V | DAC3_BIPOLAR_10V
          | DAC4_BIPOLAR_10V | DAC5_BIPOLAR_10V | DAC6_BIPOLAR_10V
          | DAC7_BIPOLAR_10V);

/* Program analog outputs to bipolar zero */
for (i=0; i < 4; i++) {
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i],
          DAC_BIPOLAR_ZERO | (DAC_BIPOLAR_ZERO << 16));
}

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The analog modes and outputs do not actually change until the write to the Analog Update Register and Analog Mode Update Register. All 8 analog output channels are changed at the same time.

Note that the outputs are updated first, to reduce the magnitude of the potential change in output between updating the output and the mode. For example, if the original mode was unipolar, 10V mode, and the output values were 0x0000, then changing the mode first would cause the output to start to go toward -10V. By updating the analog outputs first, the output only starts to go toward +5V before the mode update brings it back to 0V.

Configuring all channels as bipolar, 5V

To configure all analog output channels as bipolar, 5V, change and update the analog mode. Also set the analog output value to ensure it is at the desired value – do not assume that the output will be 0V after changing the mode.

```
/* Program analog modes to bipolar, 5V */
WriteDWordToHwMem(&pQ8->analogMode.all, 0x00f000f0L);

/* Program analog outputs to bipolar zero */
for (i=0; i < 4; i++)
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i], 0x08000800L);

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The constants defined in Q8.h may also be used, for greater readability:

```
/* Program analog modes to bipolar, 10V */
WriteDWordToHwMem(&pQ8->analogMode.all, DAC0_BIPOLAR_5V
```

```
          | DAC1_BIPOLAR_5V | DAC2_BIPOLAR_5V | DAC3_BIPOLAR_5V
          | DAC4_BIPOLAR_5V | DAC5_BIPOLAR_5V | DAC6_BIPOLAR_5V
          | DAC7_BIPOLAR_5V);

/* Program analog outputs to bipolar zero */
for (i=0; i < 4; i++) {
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i], 0);
         DAC_BIPOLAR_ZERO | (DAC_BIPOLAR_ZERO << 16));
}

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The analog modes and outputs do not actually change until the write to the Analog Update Register and Analog Mode Update Register. All 8 analog output channels are changed at the same time.

Configuring all channels as unipolar, 10V

To configure all analog output channels as unipolar, 10V, change and update the analog mode. Also set the analog output value to ensure it is at the desired value – do not assume that the output will be 0V after changing the mode.

```
/* Program analog modes to unipolar, 10V */
WriteDWordToHwMem(&pQ8->analogMode.all, 0x00000000L);

/* Program analog outputs to unipolar zero */
for (i=0; i < 4; i++)
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i], 0x00000000L);

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The constants defined in Q8.h may also be used, for greater readability:

```
/* Program analog modes to unipolar, 10V */
WriteDWordToHwMem(&pQ8->analogMode.all, DAC0_UNIPOLAR_10V
          | DAC1_UNIPOLAR_10V | DAC2_UNIPOLAR_10V | DAC3_UNIPOLAR_10V
          | DAC4_UNIPOLAR_10V | DAC5_UNIPOLAR_10V | DAC6_UNIPOLAR_10V
          | DAC7_UNIPOLAR_10V);

/* Program analog outputs to unipolar zero */
for (i=0; i < 4; i++)
    WriteDWordToHwMem(&pQ8->analogOutput.pairs[i], 0x0000000L);

/* Transfer modes and output values to analog outputs */
WriteDWordToHwMem(&pQ8->analogUpdate.all, 0);
WriteDWordToHwMem(&pQ8->analogModeUpdate.all, 0);
```

The analog modes and outputs do not actually change until the write to the Analog Update Register and Analog Mode Update Register. All 8 analog output channels are changed at the same time.

Configuring channels 0 and 2 as bipolar, 10V

To configure channels 0 and 2 as bipolar, 10V, while configuring channels 1 and 3 as unipolar, 10V, change and update the analog mode. Also set the analog output value to ensure it is at the desired value – do not assume that the output will be 0V after changing the mode.

This example uses 16-bit memory accesses since channels 0-3 may be accessed as the lower 16-bits of each register. If all 8 analog outputs will be configured, use 32-bit accesses to configure multiple channels at the same time, as in the preceding examples.

```
/* Program analog modes */
WriteWordToHwMem(&pQ8->analogMode.four.dac03, 0x0550L);

/* Program analog outputs to zero volts */
WriteWordToHwMem(&pQ8->analogOutput.one.dac0, 0x0800);
WriteWordToHwMem(&pQ8->analogOutput.one.dac1, 0x0000);
WriteWordToHwMem(&pQ8->analogOutput.one.dac2, 0x0800);
WriteWordToHwMem(&pQ8->analogOutput.one.dac3, 0x0000);

/* Transfer modes and output values to analog outputs */
WriteWordToHwMem(&pQ8->analogUpdate.four.dac03, 0);
WriteWordToHwMem(&pQ8->analogModeUpdate.four.dac03, 0);
```

The constants defined in Q8.h may also be used, for greater readability:

```
/* Program analog modes to bipolar, 10V */
WriteWordToHwMem(&pQ8->analogMode.four.dac03, DAC0_BIPOLAR_10V
          | DAC1_UNIPOLAR_10V | DAC2_BIPOLAR_10V | DAC3_UNIPOLAR_10V);

/* Program analog outputs to zero volts */
WriteWordToHwMem(&pQ8->analogOutput.one.dac0, DAC_BIPOLAR_ZERO);
WriteWordToHwMem(&pQ8->analogOutput.one.dac1, 0);
WriteWordToHwMem(&pQ8->analogOutput.one.dac2, DAC_BIPOLAR_ZERO);
WriteWordToHwMem(&pQ8->analogOutput.one.dac3, 0);

/* Transfer modes and output values to analog outputs */
WriteWordToHwMem(&pQ8->analogUpdate.four.dac03, 0);
WriteWordToHwMem(&pQ8->analogModeUpdate.four.dac03, 0);
```

The analog modes and outputs do not actually change until the write to the Analog Update Register and Analog Mode Update Register. All 8 analog output channels are changed at the same time.

Configuring transparent mode for channels 0-3

To configure analog output channels 0-3 to operate in transparent mode, set the DAC03_TR bit in the Control Register:

```
WriteDWordToHwMem(&pQ8->control, control | 0x01000000);
```

where `control` is the contents of the other control register bits. Using the constants defined in the Q8.h header file, the same line of code becomes:

```
WriteDWordToHwMem(&pQ8->control, control | CTRL_DAC03TR);
```

Subsequent writes to the D/A Output Registers (A, B, C or D) will update the analog outputs for channels 0-3 immediately, without waiting for a write to the D/A Update Register. Note that transparent mode results in slightly faster performance, since the write to the D/A Update Register is eliminated, but the analog outputs are no longer updated simultaneously.

Configuring transparent mode for all channels

To configure all the analog output channels to operate in transparent mode, set the DAC03_TR and DAC47_TR bits in the Control Register:

```
WriteDWordToHwMem(&pQ8->control, cntrl | 0x03000000);
```

where `cntrl` is the contents of the other control register bits. Using the constants defined in the Q8.h header file:

```
WriteDWordToHwMem(&pQ8->control, cntrl | CTRL_DAC03TR | CTRL_DAC47TR);
```

Subsequent writes to the D/A Output Registers (A, B, C or D) will update the analog outputs for channels 0-3 immediately, without waiting for a write to the D/A Update Register. Note that transparent mode results in slightly faster performance, since the write to the D/A Update Register is eliminated, but the analog outputs are no longer updated simultaneously.

Writing to the Analog Outputs

Writing to the analog outputs is performed by writing to the analog output latches via the D/A Output Registers (A, B, C and D), and then updating the analog outputs from the latches via the D/A Update Register. The update is unnecessary if transparent mode is enabled for the analog outputs. However, the advantage of a separate update command is that all the analog outputs may be changed simultaneously.

Writing to Analog Output Channel 0

Suppose analog output channel 0 is configured as bipolar, 10V. To write 5V to analog output channel 0, perform the following operations:

```
const uint16_T five_volts = DAC_BIPOLAR_ZERO + (uint16_T)(5 * 4096/20);

WriteWordToHwMem(&pQ8->analogOutput.one.dac0, five_volts);
WriteWordToHwMem(&pQ8->analogUpdate, 0);
```

The last line is unnecessary if transparent mode is enabled for channels 0-3.

Writing to Analog Output Channels 0 and 4

Suppose analog output channels 0 and 4 are configured as bipolar, 10V. To write 5V to analog output channel 0 and -2V to channel 4, perform the following operations:

```
const uint16_T five_volts = DAC_BIPOLAR_ZERO + (uint16_T)(5 * 4096/20);
const uint32_T two_volts  = DAC_BIPOLAR_ZERO - (uint16_T)(2 * 4096/20);

WriteDWordToHwMem(&pQ8->analogOutput.two.dac04,
                  five_volts | (two_volts << 16));
WriteDWordToHwMem(&pQ8->analogUpdate, 0);
```

The last line is unnecessary if transparent mode is enabled for all channels.

The above code could be implemented using separate 16-bit writes for each analog output as follows:

```
WriteWordToHwMem(&pQ8->analogOutput.one.dac0, five_volts);
WriteWordToHwMem(&pQ8->analogOutput.one.dac4, two_volts);
WriteDWordToHwMem(&pQ8->analogUpdate, 0);
```

While this code is more readable, using 16-bit writes is less efficient. Because channels 0 and 4 may be accessed in the same 32-bit word, the output latches for both channels may be set using a single 32-bit write as in the first instance. Thus, the Q8 makes optimum use of the 32-bit PCI bus, minimizing the overhead of programming the card.

Writing to Analog Output Channels 0, 4 and 5

Suppose analog output channels 0 and 4 are configured as bipolar, 5V, while channel 5 is configured as unipolar, 10V. To write 2V to analog output channels 0 and 5 and -2V to channel 4, perform the following operations:

```
const uint16_T two_pos = DAC_BIPOLAR_ZERO + (uint16_T)(2 * 4096/10);
const uint32_T two_neg = DAC_BIPOLAR_ZERO - (uint16_T)(2 * 4096/10);
const uint16_T two_uni = (uint16_T)(2 * 4096/10);

WriteDWordToHwMem(&pQ8->analogOutput.two.dac04,
                  two_pos | (two_neg << 16));
WriteWordToHwMem(&pQ8->analogOutput.one.dac5, two_uni);
WriteDWordToHwMem(&pQ8->analogUpdate, 0);
```

The last line is unnecessary if transparent mode is enabled for all channels. Note that because channels 0 and 4 may be accessed in the same 32-bit word, only a single write is re-

quired to set the output latches for both channels. Also note that when transparent mode is disabled, all three analog outputs are updated simultaneously by the last line of code.

Configuring the Analog Inputs

The Q8 has eight single-ended analog input channels. Each channel accepts inputs in the -10V to +10V range, and converts the analog signal to a 14-bit, binary, twos-complement result. This 14-bit result is sign-extended by the Q8 hardware to 16 bits to make programming the Q8 easier.

There are two A/D converters onboard: ADC03 and ADC47. Each converter handles four inputs. ADC03 handles channels 0-3 and ADC47 handles channels 4-7. Each chip can sample all four channels simultaneously, and then hold the sampled values while it converts each channel, sequentially, to a digital result. The converted results are stored in an onboard FIFO, which may then be read by the device driver or real-time application software.

Thus, when performing A/D conversions, the first step is to select which of the four channels will be converted by each A/D converter. The A/D converter(s) are then commanded to sample their inputs and convert each selected channel. The status of the operation may be polled or may be programmed to generate an interrupt upon completion of each conversion, or when all the selected channels have been converted. The results are then read from the A/D FIFO.

The channels are converted using one of two clocks: the internal clock of the A/D converter (2.4 usecs/conversion), or a clock common to both A/D converters (3.36 usec/conversion). The common clock is supplied so that the two A/D converters may be synchronized and all eight channels may be sampled simultaneously.

Note that it is not necessary to use the common clock to get simultaneous sampling when only one A/D converter is being used. Each A/D converter already samples its input channels simultaneously. The common clock is only required when simultaneous sampling is required across *both* converters (for example, when all 8 channels are being read).

Configuring the analog inputs thus consists of selecting which channels to convert, and which clock will be used to drive the conversions. Once the analog inputs are configured, multiple conversions may be performed, without reconfiguring the inputs each time.

Selecting Analog Inputs 0 and 2 for Conversion

There are two ways to select analog input channels 0 and 2 for conversion. Through the Control Register or the A/D Register. Which method is used depends on the clock used to drive the conversions. If the common clock is used, then the A/D Register must be used to select the channels for conversion. If the internal clocks are used, then either method may

be employed. Using the Control Register is slightly more efficient, since other operations may be configured at the same time.

Using the Control Register

Assume the global variable `cntrl` contains the current contents of the Control Register. To select analog channels 0 and 2 for conversion, do:

```
cntrl = (cntrl & 0xffff00ffL) | 0x00000500L);
WriteDWordToHwMem(&pQ8->control, cntrl);
```

or using the constants defined in Q8.h:

```
cntrl = (cntrl & 0xffff00ffL) | CTRL_ADCSL0 | CTRL_ADCSL2);
WriteDWordToHwMem(&pQ8->control, cntrl);
```

This single write to the Control Register configures the A/D to use the Control Register contents for channel selection, as well as selecting channels 0 and 2 for conversion. It also programs the A/D for manual conversions. The conversion does not actually take place at this time. A *second* write to the Control Register is used to trigger the actual conversion.

Using the A/D Register

Assume the global variable `cntrl` contains the current contents of the Control Register. To select analog channels 0 and 2 for conversion, do:

```
/* Program Control Register so that A/D Register used for channels */
cntrl = (cntrl & 0xffff00ffL) | 0x00001000);
WriteDWordToHwMem(&pQ8->control, cntrl);

/* Program selected channels to A/D Register */
WriteDWordToHwMem(&pQ8->analogInput.select, 0x00000005L);
```

or using the constants defined in Q8.h:

```
/* Program Control Register so that A/D Register used for channels */
cntrl = (cntrl & 0xffff00ffL) | CTRL_ADC03HS);
WriteDWordToHwMem(&pQ8->control, cntrl);

/* Program selected channels to A/D Register */
WriteDWordToHwMem(&pQ8->analogInput.select, 0x00000005L);
```

The write to the Control Register configures the A/D to use the A/D Register contents for channel selection. It also programs the A/D for manual conversions and to use the internal A/D clock. Other Control Register options (potentially unrelated to the A/D) can be programmed at the same time.

The conversion does not actually take place at this time. A *second* write to the Control Register is used to trigger the actual conversion.

The channel selection is programmed by writing to the A/D Register. Once again, the channel selection only needs to be configured once. After that, A/D conversions may be performed any number of times without reconfiguring the channels converted (unless, of course, you wish to convert a different set of channels). To change which channels are converted, only the A/D Register contents need be changed.

Selecting Channels 0 and 4 for Simultaneous Sampling

Channels 0 and 4 are converted by separate A/D converters: ADC03 and ADC47. In order to guarantee that both channels are sampled simultaneously, the common clock must be used. The common clock can only be used when the A/D Register is used to select the channels for conversion. Note that it is still faster to convert channels 0 and 4 than it is to convert channels 0 and 1, because channels 0 and 4 are converted in parallel, while channels 0 and 1 are converted sequentially.

Assume the global variable `cntrl` contains the current contents of the Control Register. To select analog channels 0 and 4 for conversion, do:

```
/* Program Control Register so that A/D Register used for channels */
cntrl = (cntrl & 0xffff00ffL) | 0x00121200L);
WriteDWordToHwMem(&pQ8->control, cntrl);

/* Program selected channels to A/D Register */
WriteDWordToHwMem(&pQ8->analogInput.select, 0x00010001L);
```

or using the constants defined in Q8.h:

```
/* Program Control Register so that A/D Register used for channels */
cntrl = (cntrl & 0xff0000ffL) | CTRL_ADC03HS  | CTRL_ADC47HS
                              | CTRL_ADC47SCK | CTRL_ADC03SCK);
WriteDWordToHwMem(&pQ8->control, cntrl);

/* Program selected channels to A/D Register */
WriteDWordToHwMem(&pQ8->analogInput.select, 0x00010001L);
```

The write to the Control Register configures both A/D converters to use the A/D Register contents for channel selection. It also programs both A/Ds for manual conversions and to use the common A/D clock. Other Control Register options (potentially unrelated to the A/D) can be programmed at the same time. The conversion does not actually take place at this time. A *second* write to the Control Register is used to trigger the actual conversion.

The channel selection is programmed by writing to the A/D Register. Note that both A/D converters are programmed by a single 32-bit write. Once again, the channel selection only needs to be configured once. After that, A/D conversions may be performed any number of times without reconfiguring the channels converted (unless, of course, you wish to convert a different set of channels). To change which channels are converted, only the A/D Register contents need be changed.

Reading the Analog Inputs

After configuring the channels that will be converted, the analog inputs may be read. The selected channels may be read any number of times, without reconfiguring the channel selection. The channel selection only needs to be reconfigured if a different set of channels will be read.

Reading Channels 0 and 2

Suppose channels 0 and 2 have been selected for conversion. Since channels 0 and 2 are handled by the same A/D converter (ADC03), they will be converted sequentially (but sampled simultaneously!). There are two ways the results may be read: after each channel is converted, or after both channels have been converted. Reading after both channels have been converted yields the best noise performance. Reading results immediately, as they become available, may be slightly faster, but not necessarily so.

In the following examples, assume that the variable `cntrl` is a global variable that contains the current contents of the Control Register.

Reading the Results After Conversions Complete

To read the conversion results after all the conversions are complete, do the following:

```
int16_T value[2]; /* array of conversion results */
int i;

/* Start A/D conversions on ADC03 */
WriteDWordToHwMem(&pQ8->control, cntrl | 0x00008000L);

/* Wait for all the results to become available */
while (!(ReadDWordFromHwMem(&pQ8->status) & 0x00040000L));

/* Read the results from the FIFO */
for (i=0; i < 2; i++)
    value[i] = ReadWordFromHwMem(&pQ8->analogInput.one.adc03);
```

or, using the constants defined in Q8.h:

```
int16_T value[2]; /* array of conversion results */
int i;

/* Start A/D conversions on ADC03 */
WriteDWordToHwMem(&pQ8->control, cntrl | CTRL_ADC03CV);

/* Wait for all the results to become available */
while (!(ReadDWordFromHwMem(&pQ8->status) & STAT_ADC03RDY));

/* Read the results from the FIFO */
for (i=0; i < 2; i++)
```

```
value[i] = ReadWordFromHwMem(&pQ8->analogInput.one.adc03);
```

Performing an A/D conversion is relatively simple in this case. Simply start the conversion sequence by writing to the Control Register. Wait for all the channels to be converted by polling the ADC03_RDY bit in the Status Register. Then read all the results from the A/D FIFO by reading the A/D Register.

The channels are always converted in ascending order. Hence, after the above code has executed:

```
value[0] = channel 0 conversion result (16-bit twos-complement binary)
value[1] = channel 2 conversion result (16-bit twos-complement binary)
```

Reading the Results As Soon As Possible

To start the conversions and read the conversion results as they become available, perform the following operations:

```
int16_T value[2]; /* array of conversion results */
int i;

/* Clear interrupt status flags for ADC03 */
WriteDWordToHwMem(&pQ8->interruptStatus, 0x00050000L);

/* Start A/D conversions on ADC03 */
WriteDWordToHwMem(&pQ8->control, cntrl | 0x00008000L);
for (i=0; i < 2; i++) {
    /* Wait for the next result to become available */
    while (!(ReadDWordFromHwMem(&pQ8->interruptStatus) & 0x00050000L));

    /* Read the next result from the FIFO */
    value[i] = ReadWordFromHwMem(&pQ8->analogInput.one.adc03);

    /* Reset ADC03 end-of-conversion flag for next poll */
    WriteDWordToHwMem(&pQ8->interruptStatus, 0x00010000L);
}
```

or, using the constants defined in Q8.h:

```
int16_T value[2]; /* array of conversion results */
int i;
const uint32_T intMask = INT_ADC03RDY | INT_ADC03EOC;

/* Clear interrupt status flags for ADC03 */
WriteDWordToHwMem(&pQ8->interruptStatus, intMask);

/* Start A/D conversions on ADC03 */
WriteDWordToHwMem(&pQ8->control, cntrl | CTRL_ADC03CV);
for (i=0; i < 2; i++) {
    /* Wait for the next result to become available */
    while (!(ReadDWordFromHwMem(&pQ8->interruptStatus) & intMask));
```

```
    /* Read the next result from the FIFO */
    value[i] = ReadWordFromHwMem(&pQ8->analogInput.one.adc03);

    /* Reset ADC03 end-of-conversion flag for next poll */
    WriteDwordToHwMem(&pQ8->interruptStatus, INT_ADC03EOC);
}
```

This code uses the Interrupt Status Register, rather than the Status Register, to poll the end-of-conversion flag because the flag is asserted so briefly that it could be missed polling the Status Register. The Status Register can still be used, however, in which case the code would look like:

```
int16_T value[2]; /* array of conversion results */
int i;
const uint32_T statMask = STAT_ADC03RDY | STAT_ADC03EOC;

/* Start A/D conversions on ADC03 */
WriteDWordToHwMem(&pQ8->control, cntrl | CTRL_ADC03CV);
for (i=0; i < 2; i++) {
    /* Wait for the next result to become available */
    while (!(ReadDWordFromHwMem(&pQ8->status) & statMask));

    /* Read the next result from the FIFO */
    value[i] = ReadWordFromHwMem(&pQ8->analogInput.one.adc03);
}
```

which is much simpler. The only risk in this technique is that one of the end-of-conversion flags will be missed, and the conversion results will not be read until the next conversion is detected. The correct information will still be returned – it will just be marginally slower.

The above code polls both the end-of-conversion bit *and the ready bit* for the A/D. Polling the ready bit is essential to correct operation of the code. The ready bit is asserted when all the A/D conversions are complete. Thus, polling the ready bit ensures that even if the end-of-conversion flag is missed, all channels will be read by the time all the conversions are complete. Note that the ready bit stays asserted until the next sequence of A/D conversions is initiated by writing to the ADC03_CV bit in the Control Register

The channels are always converted in ascending order. Hence, after the above code has executed:

```
value[0] = channel 0 conversion result (16-bit twos-complement binary)
value[1] = channel 2 conversion result (16-bit twos-complement binary)
```

Reading Channels 0 and 4

Suppose channels 0 and 4 have been selected for conversion. Since channels 0 and 4 are handled by different A/D converters (ADC03 and ADC47), they will be converted in parallel. There are two ways the results may be read: after each channel is converted, or after

both channels have been converted. Since these two cases are discussed in the preceding section, only one case will be considered here – reading the results after all the conversions are complete:

```
const uint32_T statMask = STAT_ADC03RDY | STAT_ADC47RDY;
int32_T values;
int16_T value[2];
int i;

/* Start conversions on ADC03 and ADC47 */
WriteDWordToHwMem(&pQ8->control, CTRL_ADC03CV | CTRL_ADC47CV);

/* Wait for all conversions to complete (on both A/D's) */
while ((ReadDWordFromHwMem(&pQ8->status) & statMask) != statMask);

/* Read conversion results */
values = (int32_T)ReadDWordFromHwMem(&pQ8->analogInput.two);
value[0] = (int16_T)(values);
value[1] = (int16_T)(values >> 16);
```

After the above code has executed:

```
value[0] = channel 0 conversion result (16-bit twos-complement binary)
value[1] = channel 4 conversion result (16-bit twos-complement binary)
```

Note that conversions are initiated on both A/D converters at the same time by a single 32-bit write to the Control Register.

Similarly, the while loop waits for both A/D converters to finish their conversions before proceeding. The same while loop will work regardless how many channels are being converted by each A/D. However, it is preferable to have each A/D converting the same number of channels in this case, or time will be wasted that could be used to read the conversion results of the A/D that is finished.

Finally, both A/D results are read at the same time, by doing a 32-bit read from the A/D Register. The result from channel 0 will be in the lower 16 bits of this double-word, while the result from channel 4 is in the upper 16 bits. Since the 14-bit conversion results are sign-extended to 16 bits, the software does not have to sign-extend the results. Note that it is more efficient to do the single 32-bit read and extract the results for the two channels than it is to do two 16-bit reads, since the 33 MHz PCI bus is slower than the system bus (typically 133MHz or faster) and CPU cache.

The results may be converted to a voltage by multiplying by ADC_FACTOR.

Reading All Eight Channels

Suppose all channels have been selected for conversion. Since channels 0-3 and 4-7 are handled by different A/D converters (ADC03 and ADC47), they will be converted in paral-

lel. This example reads the conversion results after all 8 channels have been read. It also converts the results to a floating-point voltage value.

```
const uint32_T statMask = STAT_ADC03RDY | STAT_ADC47RDY;
int32_T values;
int16_T value[8];
real_T voltage[8];
int i;

/* Start conversions on ADC03 and ADC47 */
WriteDWordToHwMem(&pQ8->control, CTRL_ADC03CV | CTRL_ADC47CV);

/* Wait for all conversions to complete (on both A/D's) */
while ((ReadDWordFromHwMem(&pQ8->status) & statMask) != statMask);

/* Read conversion results for all eight channels */
for (i=0; i < 4; i++) {
    values     = (int32_T)ReadDWordFromHwMem(&pQ8->analogInput.two);
    value[i]   = (int16_T)(values);        /* channels 0, 1, 2 and 3 */
    value[i+4] = (int16_T)(values >> 16);  /* channels 4, 5, 6 and 7 */
}

/* Convert integer results to a floating-point voltage */
for (i=0; i < 8; i++)
    voltage[i] = value[i] * ADC_FACTOR;
```

After the above code has executed:

```
value[i] = channel i conversion result (16-bit twos-complement binary)
```

where *i*=0..7.

Note that conversions are initiated on both A/D converters at the same time by a single 32-bit write to the Control Register. Also, each A/D converter converts four channels before setting the ready flag in the Status Register. The results are stored in the converter's FIFO. The results from each A/D are read from their respective FIFOs at the same time using a 32-bit read.

Reading of the conversion results could be done using 16-bit reads as follows:

```
/* Read conversion results for all eight channels */
for (i=0; i < 4; i++)
    value[i] = (int16_T)ReadWordFromHwMem(&pQ8->analogInput.one.adc03);
for (; i < 8; i++)
    value[i] = (int16_T)ReadWordFromHwMem(&pQ8->analogInput.one.adc47);
```

While this code is simpler to understand, it executes more slowly because the CPU can execute instructions in its cache much faster than it can read from the PCI bus (33MHz compared to a CPU speed > 1 Ghz!). In the 16-bit version, 8 reads are performed from the PCI bus to read the conversion results from the A/D FIFOs. In the previous 32-bit version, only 4 PCI reads are required to get the same information.

Configuring the Encoders

The Q8 has four encoder chips, each handling two channels: an even channel and an odd channel. Each channel has a multitude of options that may be programmed on an individual basis, including the counting mode, how the index pulse is used, etc.

On power-up, the encoder chips are in an indeterminate state. Hence, all the encoder registers must be programmed before using the encoder.

The Initializing All Encoder Channels example below demonstrates how to initialize all eight encoder channels after power-up. Rather than repeat the full initialization, subsequent examples show how to change that configuration to suit individual needs.

Initializing All Encoder Channels

The following example shows how to fully initialize all eight encoder channels as quadrature encoder inputs, with no index pulse, after power-up. A different configuration could be used, but it is important to fully initialize each encoder chip before use to ensure correct operation. This example assumes that the global variable `cntrl` contains the current contents of the Control Register.

```
static const uint32_T cmrInit  = ENC_BOTH_CHANNELS | ENC_CMR_REGISTER
    | ENC_CMR_BINARY | ENC_CMR_NORMAL | ENC_CMR_QUADRATURE_4X;
static const uint32_T rldInit1 = ENC_BOTH_CHANNELS | ENC_RLD_REGISTER
    | ENC_RLD_RESET_BP | ENC_RLD_RESET_E;
static const uint32_T rldInit2 = ENC_BOTH_CHANNELS | ENC_RLD_REGISTER
    | ENC_RLD_RESET_FLAGS | ENC_RLD_SET_PSC;
static const uint32_T rldInit3 = ENC_BOTH_CHANNELS | ENC_RLD_REGISTER
    | ENC_RLD_RESET_BP | ENC_RLD_RESET_CNTR;
static const uint32_T iorInit  = ENC_BOTH_CHANNELS | ENC_IOR_REGISTER
    | ENC_IOR_ENABLE_AB | ENC_IOR_LCNTR_LATCH | ENC_IOR_INDEX_ERROR;
static const uint32_T idrInit  = ENC_BOTH_CHANNELS | ENC_IDR_REGISTER
    | ENC_IDR_DISABLE_INDEX | ENC_IDR_POS_INDEX | ENC_IDR_LCNTR_INDEX;

/* Initialize the counter modes to 4X quadrature, normal counting */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (cmrInit << 24) | (cmrInit << 16) | (cmrInit << 8) | cmrInit);

/* Reset the BP and E flags of each encoder */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (rldInit1 << 24) | (rldInit1 << 16) | (rldInit1 << 8) | rldInit1);

/* Set the max. filter clock freq. and reset encoder status flags */
/* Also clear the preload registers for all channels */
WriteDWordToHwMem(&pQ8->encoderData.four.enc0246, 0); /* LSB */
WriteDWordToHwMem(&pQ8->encoderData.four.enc1357, 0);
WriteDWordToHwMem(&pQ8->encoderData.four.enc0246, 0); /* ISB */
WriteDWordToHwMem(&pQ8->encoderData.four.enc1357, 0);
WriteDWordToHwMem(&pQ8->encoderData.four.enc0246, 0); /* MSB */
```

```
WriteDWordToHwMem(&pQ8->encoderData.four.enc1357, 0);
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (rldInit2 << 24) | (rldInit2 << 16) | (rldInit2 << 8) | rldInit2);

/* Reset the encoder count values */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (rldInit3 << 24) | (rldInit3 << 16) | (rldInit3 << 8) | rldInit3);

/* Enable A,B inputs, set index to load counter, assign output flags */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (iorInit << 24) | (iorInit << 16) | (iorInit << 8) | iorInit);

/* Disable the index feature (make I/LD asynchronous) */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    (idrInit << 24) | (idrInit << 16) | (idrInit << 8) | idrInit);

/* Disable the index inputs */
WriteDWordToHwMem(&pQ8->control, cntrl & 0xffffff00L);
```

Note that all eight channels are programmed simultaneously. The four encoder chips are programmed at the same time using 32-bit writes, and both the even and odd channels for each chip are programmed simultaneously by setting the most significant bit (ENC_BOTH_CHANNELS) of each byte written.

Three separate writes to the RLD registers are required to fully initialize the status flags of the encoder. The other encoder registers can be are initialized by a single write. Take careful note of when the byte pointer (BP) is reset. Each encoder chip has an 8-bit interface, so writing 24-bit values, or the filter prescale value, requires resetting this byte pointer before the 24-bit count value or 8-bit prescale value is written. The byte pointer is also used when the encoder output latch is read.

The last line disables the index inputs in the Control Register. Since the index inputs cannot be disabled by the encoder chips themselves, this functionality is provided through the Control Register. Disabling an index input forces the index input to the encoder chip to be always high, as if there is never an index pulse.

Configuring Channel 0 as a Non-Quadrature Input

To configure encoder channel 0 as a non-quadrature input, perform the following operations:

```
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_ONE_CHANNEL
    | ENC_CMR_REGISTER | ENC_CMR_BINARY | ENC_CMR_NORMAL
    | ENC_CMR_NONQUADRATURE);
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_ONE_CHANNEL
    | ENC_IDR_REGISTER | ENC_IDR_DISABLE_INDEX | ENC_IDR_POS_INDEX
    | ENC_IDR_LCNTR_INDEX);
```

The first line sets the counter mode for channel 0 to non-quadrature, normal binary counting. In this mode, the 24-bit counter wraps around on overflow or underflow. The A and B inputs are treated as CNT (counter clock) and DIR (direction) inputs respectively instead of quadrature inputs.

The second write ensures that the index feature is not enabled for channel 0, since the index pulse cannot be synchronized to quadrature clocks in non-quadrature mode (the index input must be asynchronous, like the asynchronous preset or clear input of a D flip-flop). If the index feature is already disabled, then the second write is unnecessary.

Configuring Channels 0 and 1 as Non-Quadrature Inputs

Encoder channels 0 and 1 are handled by the same encoder chip. Hence, configuring encoder channels 0 and 1 as non-quadrature inputs involves the same set of operations as the preceding example except the `ENC_BOTH_CHANNELS` bit is set for each write instead of `ENC_ONE_CHANNEL`:

```
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_BOTH_CHANNELS
    | ENC_CMR_REGISTER | ENC_CMR_BINARY | ENC_CMR_NORMAL
    | ENC_CMR_NONQUADRATURE);
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_BOTH_CHANNELS
    | ENC_IDR_REGISTER | ENC_IDR_DISABLE_INDEX | ENC_IDR_POS_INDEX
    | ENC_IDR_LCNTR_INDEX);
```

The same operation could be performed using separate writes, as follows, but it is clearly less efficient to do so:

```
/* Configure channel 0 */
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_ONE_CHANNEL
    | ENC_CMR_REGISTER | ENC_CMR_BINARY | ENC_CMR_NORMAL
    | ENC_CMR_NONQUADRATURE);
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_ONE_CHANNEL
    | ENC_IDR_REGISTER | ENC_IDR_DISABLE_INDEX | ENC_IDR_POS_INDEX
    | ENC_IDR_LCNTR_INDEX);

/* Configure channel 1 */
WriteByteToHwMem(&pQ8->encoderControl.one.enc1, ENC_ONE_CHANNEL
    | ENC_CMR_REGISTER | ENC_CMR_BINARY | ENC_CMR_NORMAL
    | ENC_CMR_NONQUADRATURE);
WriteByteToHwMem(&pQ8->encoderControl.one.enc1, ENC_ONE_CHANNEL
    | ENC_IDR_REGISTER | ENC_IDR_DISABLE_INDEX | ENC_IDR_POS_INDEX
    | ENC_IDR_LCNTR_INDEX);
```

Configuring the channels separately takes twice as many write operations!

Configuring Channels 0, 1 and 2

This example is a combination of the preceding examples. It illustrates the advantages of the parallelism built into the Q8 multifunction I/O card. Since encoder channels 0 and 1 are on the same encoder chip, they can be initialized in one operation. Channel 2 is on a separate encoder chip, but can be initialized at the same time as channel 0 by using a 16-bit access to write to the two encoder chips together.

```
#define CMR (ENC_CMR_REGISTER | ENC_CMR_BINARY \
             | ENC_CMR_NORMAL | ENC_CMR_NONQUADRATURE)
#define IDR (ENC_IDR_REGISTER | ENC_IDR_DISABLE_INDEX \
             | ENC_IDR_POS_INDEX | ENC_IDR_LCNTR_INDEX)

static const uint8_T  cmr01 = ENC_BOTH_CHANNELS | CMR;
static const uint16_T cmr23 = ENC_ONE_CHANNEL   | CMR;
static const uint8_T  idr01 = ENC_BOTH_CHANNELS | IDR;
static const uint16_T idr23 = ENC_ONE_CHANNEL   | IDR;

WriteWordToHwMem(&pQ8->encoderControl.two.enc02, cmr01 | (cmr23 << 8));
WriteWordToHwMem(&pQ8->encoderControl.two.enc02, idr01 | (idr23 << 8));
```

The same operation could be performed using separate writes, as follows, but it is clearly less efficient to do so:

```
/* Configure channel 0 */
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, CMR);
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, IDR);

/* Configure channel 1 */
WriteByteToHwMem(&pQ8->encoderControl.one.enc1, CMR);
WriteByteToHwMem(&pQ8->encoderControl.one.enc1, IDR);

/* Configure channel 2 */
WriteByteToHwMem(&pQ8->encoderControl.one.enc2, CMR);
WriteByteToHwMem(&pQ8->encoderControl.one.enc2, IDR);
```

Configuring the channels separately takes six write operations, while configuring all three channels at once takes only two writes.

Configuring the Index Input to Reset the Count

The index pulse of an encoder is often used when homing the mechanical system. The system is moved slowly around its workspace in a predefined sequence and when the index pulse is seen, the encoder count value is recorded or reset so that a value of '0' indicates the home position. This example shows how to program the index input of the encoder to perform a reset when the index pulse is detected on channel 0:

```
/* Configure the index input as synchronous, and to load counter */
WriteWordToHwMem(&pQ8->encoderControl.one.enc0, ENC_IDR_REGISTER
```

```
    | ENC_IDR_ENABLE_INDEX | ENC_IDR_POS_INDEX | ENC_IDR_LCNTR_INDEX);
WriteWordToHwMem(&pQ8->encoderControl.one.one.enc0, ENC_IOR_REGISTER
    | ENC_IOR_ENABLE_AB | ENC_IOR_LCNTR_LOAD | ENC_IOR_INDEX_ERROR);

/* Set the preload register to zero so value loaded on index is zero */
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_RLD_REGISTER
    | ENC_RLD_RESET_BP);                    /* reset byte pointer */
WriteByteToHwMem(&pQ8->encoderData.one.enc0, 0);  /* write LSB */
WriteByteToHwMem(&pQ8->encoderData.one.enc0, 0);
WriteByteToHwMem(&pQ8->encoderData.one.enc0, 0);  /* write MSB */
```

The first two lines enable the I/LD input as a proper encoder index input that is synchronous to the quadrature clocks. The index input is then configured so that the 24-bit encoder counter is loaded from the preload register whenever an index pulse occurs. The preload register is then set to 0. Note that the flags of the encoder are configured so that the index pulse appears on ENC0FLG1 (ENC_IOR_INDEX_ERROR bitmask). Hence, the index will be available for interrupts, if enabled, so that the software can detect immediately when the index event occurs.

Reading the Encoders

To read the encoder count value, the count value (CNTR) must first be transferred to the output latch (OL). The output latch may then be read to retrieve the count value at the time it was latched.

Reading Encoder Input 0

This example illustrates the operations required to read just a single encoder input. See the subsequent examples for how to read multiple channels more efficiently than simply repeating this code for each channel.

```
int8_T  nMSB;
uint8_T nISB;
uint8_T nLSB;
int32_T nValue;

/* Transfer counter (CNTR) to output latch (OL) and reset byte ptr. */
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_ONE_CHANNEL
    | ENC_RLD_REGISTER | ENC_RLD_GET_CNTR | ENC_RLD_RESET_BP);

/* Read the count value from the output latch (OL) */
nLSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read LSB */
nISB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read ISB */
nMSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read MSB */
nValue = (nMSB << 16) | (nISB << 8) | nLSB;
```

The variable `nValue` will contain a 32-bit signed integer representing the sign-extended 24-bit count value.

Reading Encoder Inputs 0 and 1

This example illustrates the operations required to read encoder inputs 0 and 1. In this case, both channels are handled by the same encoder chip. Hence, it is possible to latch the count value for both channels at the same time by setting the ENC_BOTH_CHANNELS bit.

```
int8_T  nMSB;
uint8_T nISB;
uint8_T nLSB;
int32_T nValue[2];

/* Transfer counters to output latches and reset byte ptrs. */
WriteByteToHwMem(&pQ8->encoderControl.one.enc0, ENC_BOTH_CHANNELS
    | ENC_RLD_REGISTER | ENC_RLD_GET_CNTR | ENC_RLD_RESET_BP);

/* Read the count value from the output latch (OL) for channel 0 */
nLSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read LSB */
nISB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read ISB */
nMSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc0); /* read MSB */
nValue[0] = (nMSB << 16) | (nISB << 8) | nLSB;

/* Read the count value from the output latch (OL) for channel 1 */
nLSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc1); /* read LSB */
nISB = ReadByteFromHwMem(&pQ8->encoderData.one.enc1); /* read ISB */
nMSB = ReadByteFromHwMem(&pQ8->encoderData.one.enc1); /* read MSB */
nValue[1] = (nMSB << 16) | (nISB << 8) | nLSB;
```

The array `nValue` will contain 32-bit signed integers representing the sign-extended 24-bit count values:

```
nValue[0] = sign-extended count value for channel 0
nValue[1] = sign-extended count value for channel 1
```

This code is only slightly more efficient than reading the two channels separately. However, the code does ensure that both channels are sampled *simultaneously*. Also greater optimisations are possible as more channels are read.

Reading Encoder Inputs 0 and 2

This example demonstrates how to read encoder channels 0 and 2 efficiently. While the two channels are handled by different chips, the two encoder chips can be accessed at the same time using 16-bit operations.

```
static const uint8_T rld = ENC_ONE_CHANNEL
        | ENC_RLD_REGISTER | ENC_RLD_GET_CNTR | ENC_RLD_RESET_BP;
```

```
uint16_T nMSBs;
uint16_T nISBs;
uint16_T nLSBs;
int32_T nValue[2];

/* Transfer counters to output latches and reset byte ptrs. */
WriteWordToHwMem(&pQ8->encoderControl.two.enc02, rld | (rld << 8));

/* Read the count values from the output latches for channels 0 & 2 */
nLSBs = ReadWordFromHwMem(&pQ8->encoderData.two.enc02); /* read LSBs */
nISBs = ReadWordFromHwMem(&pQ8->encoderData.two.enc02); /* read ISBs */
nMSBs = ReadWordFromHwMem(&pQ8->encoderData.two.enc02); /* read MSBs */
nValue[0] = ((int8_T)(nMSBs & 0xff) << 16) | ((nISBs & 0xff) << 8)
            | (nLSB & 0xff);
nValue[1] = ((int16_T)(nMSBs & 0xff00) << 8) | (nISBs & 0xff00)
            | ((nLSB & 0xff00) >> 8);
```

Like the preceding example, this code reads two channels simultaneously. However, since both channels are even-numbered, it can read from both channels at the same time, using 16-bit reads. Thus, in four PCI bus operations, it has read both encoder channels.

Reading All Eight Encoder Inputs

To read all eight encoder inputs, the techniques used in the two preceding examples may be used to read the eight channels as efficiently as possible, while guaranteeing that all eight channels are sampled simultaneously.

```
static const uint8_T rld = ENC_BOTH_CHANNELS
        | ENC_RLD_REGISTER | ENC_RLD_GET_CNTR | ENC_RLD_RESET_BP;

uint32_T nEvenMSBs, nOddMSBs;
uint32_T nEvenISBs, nOddISBs;
uint32_T nEvenLSBs, nOddLSBs;
int32_T  nValue[8];

/* Transfer counters to output latches and reset byte ptrs. */
WriteDWordToHwMem(&pQ8->encoderControl.four.enc0246,
    rld | (rld << 8) | (rld << 16) | (rld << 24));

/* Read the count values from the output latches for even channels */
nEvenLSBs = ReadDWordFromHwMem(&pQ8->encoderData.four.enc0246);
nOddLSBs  = ReadDWordFromHwMem(&pQ8->encoderData.four.enc1357);
nEvenISBs = ReadDWordFromHwMem(&pQ8->encoderData.four.enc0246);
nOddISBs  = ReadDWordFromHwMem(&pQ8->encoderData.four.enc1357);
nEvenMSBs = ReadDWordFromHwMem(&pQ8->encoderData.four.enc0246);
nOddMSBs  = ReadDWordFromHwMem(&pQ8->encoderData.four.enc1357);

nValue[0] = ((int8_T)(nEvenMSBs & 0xff) << 16)
            | ((nEvenISBs & 0xff) << 8) | (nEvenLSB & 0xff);
nValue[1] = ((int8_T)(nOddMSBs & 0xff) << 16)
```

```
            | ((nOddISBs & 0xff) << 8) | (nOddLSB & 0xff);
nValue[2] = ((int16_T)(nEvenMSBs & 0xff00) << 8)
            | (nEvenISBs & 0xff00) | ((nEvenLSB & 0xff00) >> 8);
nValue[3] = ((int16_T)(nOddMSBs & 0xff00) << 8)
            | (nOddISBs & 0xff00) | ((nOddLSB & 0xff00) >> 8);
nValue[4] = (nEvenMSBs & 0xff0000) | ((nEvenISBs & 0xff0000) >> 8)
            | ((nEvenLSBs & 0xff0000) >> 16);
if (nEvenMSBs & 0x800000)
    nValue[4] |= 0xff000000;
nValue[5] = (nOddMSBs & 0xff0000) | ((nOddISBs & 0xff0000) >> 8)
            | ((nOddLSBs & 0xff0000) >> 16);
if (nOddMSBs & 0x800000)
    nValue[5] |= 0xff000000;
nValue[6] = ((int32_T)(nEvenMSBs & 0xff000000) >> 8)
            | ((nEvenISBs & 0xff000000) >> 16)
            | ((nEvenLSBs & 0xff000000) >> 24);
nValue[7] = ((int32_T)(nOddMSBs & 0xff000000) >> 8)
            | ((nOddISBs & 0xff000000) >> 16)
            | ((nOddLSBs & 0xff000000) >> 24);
```

The result of executing this code is an array containing 32-bit, sign-extended count values, whereby `nValue[`*i*`]` is the count value for channel *i*. While this code is less obvious than reading each encoder channel separately, it is much more efficient. In only seven PCI bus operations, all eight 24-bit encoder inputs have been read. Furthermore, all eight encoder channels are sampled simultaneously!

While there are more logic operations required to extract the count values using full 32-bit accesses, these operations can execute in the CPU cache and be performed significantly faster than the PCI bus accesses.

Configuring the 32-Bit Counters

The Q8 contains two 32-bit counters with 30ns resolution. Both counters are capable of producing a square wave or PWM output, and of generating periodic interrupts. Each counter then has its own special functions that it can perform. This section describes some of the ways in which these counters can be configured and used.

Initializing the Counters

On reset, the counters are disabled, as are the counter outputs. The counter values are initialized to zero. However, the preload registers, of which each counter has four, are indeterminate after a reset. While initialization of the counters is not strictly necessary, the following code initializes the counters by disabling them, and writing zeros to all the counter preload registers.

```
/* Disable both counters and outputs. Place in square wave mode. */
```

```
WriteDWordToHwMem(&pQ8->counterControl,
    CCTRL_CNTRWSET | CCTRL_WDOGWSET); /* select write sets #1 */
WriteDWordToHwMem(&pQ8->counter.pwm.preloadLow,   0);
WriteDWordToHwMem(&pQ8->counter.pwm.preloadHigh,  0);
WriteDWordToHwMem(&pQ8->watchdog.pwm.preloadLow,  0);
WriteDWordToHwMem(&pQ8->watchdog.pwm.preloadHigh, 0);

WriteDWordToHwMem(&pQ8->counterControl, 0); /* select write sets #0 */
WriteDWordToHwMem(&pQ8->counter.pwm.preloadLow,   0);
WriteDWordToHwMem(&pQ8->counter.pwm.preloadHigh,  0);
WriteDWordToHwMem(&pQ8->watchdog.pwm.preloadLow,  0);
WriteDWordToHwMem(&pQ8->watchdog.pwm.preloadHigh, 0);
```

Generating a Square Wave Output

Each counter can generate a square wave (or PWM) output, with a frequency of up to 16.7 MHz (60ns period). To simplify programming, each counter has two modes: square wave mode and PWM mode. In square wave mode, a single write determines the output frequency. The following example programs the Counter to produce a 1 MHz output frequency. It enables the CNTR_OUT output so that the square wave appears on the Control header of the Q8 terminal board.

```
const uint32_T period = (uint32_T)(1e-6 / 60e-9) - 1;

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->counter.sq.preload, period);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN
    | CCNTR_CNTROUTEN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN
    | CCNTR_CNTROUTEN | CCTRL_CNTRLD | CCTRL_CNTRVAL);
```

Note that the counter mode is set prior to loading the Counter because the counter cannot be loaded at the same time as its mode is being changed.

In square wave mode (the default), the counter may be considered as a cyclic counter that decrements every 60ns. It also wraps at zero (not one). Hence, the number of counts to get a one microsecond (1 MHz) period is:

```
counts = (1 usec / 60 ns) - 1
```

This logic may be seen in the definition of the constant variable `period` in the above example. The period is programmed into the counter preload register and then the counter is loaded immediately from the preload register. Note that the counter is also enabled, as well as the counter output to the Q8 terminal board Control header CNTR_OUT pin. Also note that the initial counter output value is set to be high (CCTRL_CNTRVAL bit is set).

To program the Watchdog counter is virtually identical:

```
const uint32_T period = (uint32_T)(1e-6 / 60e-9) - 1;

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->watchdog.sq.preload, period);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN
    | CCNTR_WDOGOUTEN | CCTRL_WDOGSEL);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN
    | CCNTR_WDOGOUTEN | CCTRL_WDOGSEL | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

Note that the counter mode is set prior to loading the Watchdog counter because the counter cannot be loaded at the same time as its mode is being changed.

Aside from the Watchdog registers and bits being used, note that the CCTRL_WDOGSEL bit is also set in the Counter Control Register. This bit selects the Watchdog counter output as the source of the WATCHDOG output pin on the Q8 terminal board Control header, so that the square wave output appears at the terminal board instead of the Watchdog bit of the Interrupt Status Register.

Note that both of these examples overwrite the other bits in the Counter Control Register, thereby reconfiguring the other counter. To avoid this situation, simply keep the contents of the Counter Control Register in a global variable and use logic operations to set or clear the bits of interest. (The Counter Control Register can also be read, but it is faster to use a global variable than to read from the PCI bus). For example, suppose the global variable `cntctrl` contained the value of the Counter Control Register (set during initialization). Then the Watchdog code above would look like:

```
const uint32_T period = (uint32_T)(1e-6 / 60e-9) - 1;

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->watchdog.sq.preload, period);

/* Enable the counter and force the counter to load immediately */
cntctrl = (cntctrl & 0xffff0000L)
          | CCTRL_WDOGEN | CCNTR_WDOGOUTEN | CCTRL_WDOGSEL;
WriteDWordToHwMem(&pQ8->counterControl, cntctrl);
WriteDWordToHwMem(&pQ8->counterControl,
    cntctrl | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

The CCTRL_WDOGLD and CCTRL_WDOGVAL bits are not stored in the global variable because they cause the Watchdog counter to load immediately from the active preload register. It would typically be undesirable for the counter to be reloaded every time the Counter Control Register was accessed. Note that the counter mode is set prior to loading the Counter because the counter mode cannot be changed at the same time the counter is being loaded.

Generating a PWM Output

The two 32-bit counters on the Q8 can be used to generate PWM outputs. In PWM mode, the two counters allow both the low pulse duration and the high pulse duration to be programmed independently. In this example, the Counter is programmed to produce a 10 kHz square wave with a 10% duty cycle. In other words, the low pulse will be 90 usecs and the high pulse will be 10 usecs.

```
const uint32_T low  = (uint32_T)(90e-6 / 30e-9) - 1;
const uint32_T high = (uint32_T)(10e-6 / 30e-9) - 1;

/* Set counter mode to PWM and read & write preload sets to #0 */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTRMODE);

/* Set the counter preload values */ WriteDWordToHwMem(&pQ8->counter.p-
wm.preloadLow,  low);
WriteDWordToHwMem(&pQ8->counter.pwm.preloadHigh, high);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTRMODE | CCTRL_CNTREN
    | CCNTR_CNTROUTEN | CCTRL_CNTRLD | CCTRL_CNTRVAL);
```

In this case, the Counter mode and preload register sets are configured before writing to the preload registers, to ensure that the Counter is reading from the same preload register set as that to which we are writing (read/active set = write set). It is also important not to set the CCTRL_CNTRLD bit at the same time as the counter mode is being changed, so a separate write to the Counter Control Register is required to configure the counter mode as PWM.

Programming the Watchdog counter to produce the same PWM output is virtually identical:

```
const uint32_T low  = (uint32_T)(90e-6 / 30e-9) - 1;
const uint32_T high = (uint32_T)(10e-6 / 30e-9) - 1;

/* Set counter mode to PWM and read & write preload sets to #0 */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGMODE);

/* Set the counter preload values */ WriteDWordToHwMem(&pQ8->watch-
dog.pwm.preloadLow,  low);
WriteDWordToHwMem(&pQ8->watchdog.pwm.preloadHigh, high);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGMODE | CCTRL_WDOGEN
    | CCTRL_WDOGOUTEN | CCTRL_WDOGSEL | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

In may be noted that the techique used to program the PWM output does not allow for an output that is always high or always low, since a preload value of 0 corresponds to a 30 ns pulse – not no pulse at all. To generate a constant output (high or low), refer to the next example.

Generating a Constant Output

The two counter outputs may be used as digital outputs and programmed to generate a high or low value. This operation is simply a matter of disabling the counter and setting the output value. Note that the counter mode is set prior to forcing the counter to load because the mode cannot be changed at the same time the counter is loading.

Setting CNTR_OUT to '1'

This example programs the CNTR_OUT pin to be high. The Counter is disabled so that the output does not change.

```
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTROUTEN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTROUTEN
    | CCTRL_CNTRLD | CCTRL_CNTRVAL);
```

Setting CNTR_OUT to '0'

This example programs the CNTR_OUT pin to be low. The Counter is disabled so that the output does not change.

```
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTROUTEN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTROUTEN
    | CCTRL_CNTRLD);
```

Setting WATCHDOG to '1'

This example programs the WATCHDOG pin to be high. The Watchdog counter is disabled so that the output does not change.

```
WriteDWordToHwMem(&pQ8->counterControl,
    CCTRL_WDOGSEL | CCTRL_WDOGOUTEN);
WriteDWordToHwMem(&pQ8->counterControl,
    CCTRL_WDOGSEL | CCTRL_WDOGOUTEN | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

Setting WATCHDOG to '0'

This example programs the WATCHDOG pin to be low. The Watchdog counter is disabled so that the output does not change.

```
WriteDWordToHwMem(&pQ8->counterControl,
    CCTRL_WDOGSEL | CCTRL_WDOGOUTEN);
WriteDWordToHwMem(&pQ8->counterControl,
    CCTRL_WDOGSEL | CCTRL_WDOGOUTEN | CCTRL_WDOGLD);
```

Generating Periodic Interrupts

A common use of the 32-bit counters is to generate a periodic interrupt for driving a control loop. Either counter may be used to produce a periodic interrupt. Other sources of periodic interrupts that may be used are the encoder counters or external interrupt line with an exter-

nal clock source. The encoder counters would be configured in the modulo-N counting mode in this case, with an interrupt on the CARRY flag.

This example, however, shows how to use the Counter to generate periodic interrupts. This example presumes that an interrupt service routine has been attached to the IRQ used by the Q8 card via the appropriate operating system interface (eg. RtAttachInterruptVector in VenturCom's RTX real-time kernel).

Suppose a 0.1 ms (10 kHz) sampling rate is desired. Then the Counter is programmed to generate interrupts every 0.1 ms as follows. This example presumes that the global variable `intEnable` contains the previous contents of the Interrupt Enable Register (Note that it is also possible to simply read this register).

```
const uint32_T period = (uint32_T)(0.1e-3 / 60e-9) - 1;

/* Disable interrupts from the counter (may already be disabled) */
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable & INT_CNTROUT);

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->counter.sq.preload, period);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN
    | CCTRL_CNTRLD | CCTRL_CNTRVAL);

/* Clear the interrupt status bit for the counter */
WriteDWordToHwMem(&pQ8->interruptStatus, INT_CNTROUT);

/* Enable interrupts from the counter */
intEnable |= INT_CNTROUT;
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable);
```

The code first disables interrupts from the counter just to be sure, before reprogramming the counter. This line can be omitted when interrupts from the counter are known to be disabled (the state after a hardware reset).

The counter is then programmed to generate a square wave output of the desired interrupt frequency. The Interrupt Status Register bit for the Counter is cleared so that an interrupt is not generated until the period expires, and then interrupts from the Counter are enabled.

Programming the Watchdog counter to generate periodic interrupts is virtually identical:

```
const uint32_T period = (uint32_T)(0.1e-3 / 60e-9) - 1;

/* Disable interrupts from the counter (may already be disabled) */
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable & INT_WATCHDOG);

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->watchdog.sq.preload, period);
```

```
/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN
    | CCTRL_WDOGLD | CCTRL_WDOGVAL);

/* Clear the interrupt status bit for the counter */
WriteDWordToHwMem(&pQ8->interruptStatus, INT_WATCHDOG);

/* Enable interrupts from the counter */
intEnable |= INT_WATCHDOG;
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable);
```

The Watchdog counter mode is set prior to loading the counter using CCTRL_WDOGLD because the counter cannot be loaded and the mode changed at the same time.

Note that in both examples, the counter outputs (CNTR_OUT and WATCHDOG) are not enabled because they are not used in these examples. However, there may be instances in which it is desirable to enable these outputs at the same time, such as situations where the Counter is used to trigger interrupts via the EXT_INT line on another Q8 card.

Using the Watchdog Feature

The Watchdog counter on the Q8 card may be used as a watchdog timer. In this mode, the Watchdog counter resets all the analog outputs and sets all the digital outputs on a rising edge at its output. For example, if the Watchdog counter is programmed to produce a square wave of 1 ms period, then it will reset the analog and digital outputs after 1 ms, if the Watchdog counter period is allowed to expire. Thus, applications generally reload the Watchdog counter in their interrupt service routines to prevent it from expiring. If the software crashes, and the Watchdog is not reloaded, then it will expire and reset all the Q8 outputs – preventing the system being controlled from going unstable (assuming resetting of the outputs is enough to stop the system being controlled, or at the very least, does not drive it). It is generally advisable to connect one of the digital outputs to an active-low enable input on any motor drives in the system, since expiration of the Watchdog period will cause this enable input to go high and disable the motor drive.

To program the period of the Watchdog counter, and activate its watchdog features, do the following steps. This example programs the Watchdog counter with a period of 10 ms, so that the system will shutdown, via the Q8 hardware, within 10 ms if a software failure occurs.

```
const uint32_T period = (uint32_T)(10e-3 / 60e-9) - 1;

/* Set the Watchdog preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->watchdog.sq.preload, period);

/* Activate the Watchdog and force the counter to load immediately */
```

```
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN | CCTRL_WDOGACT
    CCTRL_WDOGOUTEN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN | CCTRL_WDOGACT
    CCTRL_WDOGOUTEN | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

The Watchdog counter mode is set prior to loading the counter using CCTRL_WDOGLD because the counter cannot be loaded and the mode changed at the same time.

Note that this example also enables the external WATCHDOG output. Since the CCTRL_WDOGSEL bit is not set, the output will reflect the inverse of the WATCHDOG bit in the Interrupt Status Register. Thus, if the Watchdog period ever expires, the WATCHDOG output will go low. This WATCHDOG output can then be used to disable external hardware if necessary. For example, it can be tied to the EXT_INT line of another Q8 card to activate a watchdog event in the other card at the same time.

If an active-high output is more appropriate then one of the digital outputs can be used because all digital outputs are pulled high if the Watchdog period expires.

In the interrupt service routine, reload the Watchdog counter with a single write, provided the contents of the preload register have not been changed:

```
/* Force the Watchdog counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_WDOGEN | CCTRL_WDOGACT
    CCTRL_WDOGOUTEN | CCTRL_WDOGLD | CCTRL_WDOGVAL);
```

In order to generate an interrupt if the watchdog expires, simply enable the WATCHDOG interrupt in the Interrupt Enable Register (after programming the Watchdog counter and clearing the WATCHDOG bit in the Interrupt Status Register).

Performing Periodic A/D Conversions

The Counter is not only useful as a timebase – it can also trigger A/D conversions periodically. More importantly, it can trigger a burst of conversions periodically. For example, suppose channels 0-7 are selected for conversion in the A/D Register, and the Counter is programmed to trigger conversions every 1 ms.

When 1 ms expires, the A/D converters will simultaneously sample all eight channels (if requested) and then convert those sampled signals to digital codes at the fastest possible rate (2.4 usec / conversion or 3.36 usec / conversion depending on the A/D conversion clock selected). The results will be stored in the onboard A/D FIFOs. The A/Ds can be programmed to generate an interrupt when all selected channels have been converted. Thus, when the interrupt service routine is invoked, it need only read the results from the A/D FIFOs – it does not have to initiate the A/D conversions, or wait for the results to become available.

The process repeats every 1 ms. The important observation to make is that the Counter triggers a *set* of A/D conversions every millisecond and not just a single conversion. Further-

more, those conversions take place at the fastest possible rate. This scenario is ideal for control systems.

The following example shows how to program the Counter to trigger A/D conversions on all eight analog inputs every millisecond. It also shows the code that reads the A/D conversion results. This example assumes that the global variable `intEnable` contains the previous contents of the Interrupt Enable Register.

```
const uint32_T period = (uint32_T)(1e-3 / 60e-9) - 1;

/* Disable interrupts from the A/D (may already be disabled) */
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable
    & (INT_ADC03RDY | INT_ADC47RDY | INT_ADC03EOC | INT_ADC47EOC));

/* Set the counter preload value (which will determine the period) */
WriteDWordToHwMem(&pQ8->counter.sq.preload, period);

/* Enable the counter and force the counter to load immediately */
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN);
WriteDWordToHwMem(&pQ8->counterControl, CCTRL_CNTREN
    | CCTRL_CNTRLD | CCTRL_CNTRVAL);

/* Program the counter to trigger A/D conversions using common clock */
WriteDWordToHwMem(&pQ8->control, CTRL_ADC03CT | CTRL_ADC03HS
    | CTRL_ADC47CT | CTRL_ADC03CT | CTRL_ADC03SCK | CTRL_ADC47SCK);

/* Program the A/Ds to convert all eight channels */
WriteDWordToHwMem(&pQ8->analogInput.select, 0x000f000fL);

/* Clear the interrupt status bits for the A/D */
WriteDWordToHwMem(&pQ8->interruptStatus, INT_ADC03RDY | INT_ADC47RDY);

/* Enable interrupts from one of the A/Ds */
intEnable |= INT_ADC03RDY;
WriteDWordToHwMem(&pQ8->interruptEnable, intEnable);
```

Then, in the interrupt service routine, the Interrupt Status Register bit must be cleared to acknowledge the interrupt, and the conversion results can be read from the A/D FIFOs.:

```
int i;
int16_T value[8];
real_T voltage[8];

/* Acknowledge interrupt */
WriteDWordToHwMem(&pQ8->interruptStatus,
    INT_ADC03RDY | INT_ADC47RDY | INT_ADC03EOC | INT_ADC47EOC);

/* Read conversion results for all eight channels */
for (i=0; i < 4; i++) {
    values     = (int32_T)ReadDWordFromHwMem(&pQ8->analogInput.two);
    value[i]   = (int16_T)(values);        /* channels 0, 1, 2 and 3 */
    value[i+4] = (int16_T)(values >> 16);  /* channels 4, 5, 6 and 7 */
```

```
}

/* Convert integer results to a floating-point voltage */
for (i=0; i < 8; i++)
    voltage[i] = value[i] * ADC_FACTOR;
```

There are some important observations to make about the above example. First, the A/D converters are programmed to use the common (3.36 usec / conversion) clock because use of the common clock guarantees that both A/D converters are synchronized. Hence, if one A/D converter has completed its conversions, then the other converter will also be finished because both converters are reading the same number of channels and use the same conversion clock. Thus, it is only necessary to interrupt using the ready signal from one of the A/D converters. The ADC03RDY flag is used in this example. The ready signal indicates that all conversions have been completed by the A/D. The code still acknowledges the other A/D interrupt sources (ADC47RDY and the end-of-conversion flags) just in case they were accidently enabled in the Interrupt Enable Register.

The output of the interrupt service routine code is an array of voltage values, as read from each of the eight analog input channels.

Appendix

This appendix contains a number of header files that are included by the register-level programming examples. The DataTypes.h header file defines convenient data types. The Hardware.h header file defines macros for accessing a PCI card using memory-mapped I/O. The Q8.h header file defines the register offsets and bit masks used to program the Q8 multifunction I/O card.

DataTypes.h

```
#if !defined(_DATATYPES_H)
#define _DATATYPES_H

typedef int            boolean_T;
typedef char           char_T;
typedef int            int_T;
typedef unsigned int   uint_T;
typedef double         real_T;
typedef signed char    int8_T;
typedef unsigned char  uint8_T;
typedef short          int16_T;
typedef unsigned short uint16_T;
typedef int            int32_T;
typedef unsigned int   uint32_T;

#ifdef _MSC_VER
typedef __int64 int64_T;
#else
typedef long long int64_T;
#endif

#ifdef _MSC_VER
typedef unsigned __int64 uint64_T;
#else
typedef unsigned long long uint64_T;
#endif

#if !defined(TRUE)
#define TRUE    (1)
#endif

#if !defined(FALSE)
#define FALSE   (0)
#endif

#if !defined(EXTERN)
#if defined(__cplusplus)
#define EXTERN extern "C"
#else
```

```
#define EXTERN extern
#endif
#endif

#if !defined(INLINE)
#define INLINE static __inline
#endif

#endif /* _DATATYPES_H */
```

Hardware.h

```
#if !defined(_HARDWARE_H)
#define _HARDWARE_H

#define readb(addr)         *(volatile unsigned char *) (addr)
#define readw(addr)         *(volatile unsigned short *)(addr)
#define readl(addr)         *(volatile unsigned long *) (addr)
#define writeb(value, addr) do { readb(addr) = (value); } while(0)
#define writew(value, addr) do { readw(addr) = (value); } while(0)
#define writel(value, addr) do { readl(addr) = (value); } while(0)

#define ReadByteFromHwMem(addr)             readb(addr)
#define WriteByteToHwMem(addr,val)          writeb(val, addr)
#define ReadWordFromHwMem(addr)             readw(addr)
#define WriteWordToHwMem(addr,val)          writew(val, addr)
#define ReadDWordFromHwMem(addr)            readl(addr)
#define WriteDWordToHwMem(addr,val)         writel(val, addr)

#if !defined(LoByte)
#define LoByte(word)   ((word) & 0x00ff)
#endif
#if !defined(HiByte)
#define HiByte(word)   (((word) >> 8) & 0x00ff)
#endif

#endif
```

Q8.h

```
/*++
Copyright (c) 2003 Quanser Consulting Inc. All Rights Reserved

Module Name:

    Q8.h

Abstract:
```

```
    This module contains the common declarations shared by device
    drivers.

Author:

    Daniel R. Madill

Environment:

    kernel

Notes:


Revision History:


--*/

#if !defined(_Q8_h)
#define _Q8_h

#include "DataTypes.h"

#ifndef FALSE
#define FALSE 0
#endif

#ifndef TRUE
#define TRUE 1
#endif

/* PCI device and vendor IDs. */
#define VENDORID_Q8               0x11E3          /* PCI VendorID */
#define DEVICEID_Q8               0x0010          /* PCI DeviceID */
#define SUBVENDORID_Q8            0x5155          /* PCI SubVendorID */
#define SUBDEVICEID_Q8            0x0200          /* PCI SubDeviceID */

/* A data structure in which each member maps to */
/* a register on the card. Ensure byte alignment */
/* of fields so that the members are at the correct */
/* offsets. This data structure is the most convenient */
/* way to access the registers on the card. */

#if defined(_MSC_VER)
#pragma pack(push, registers, 1)
#endif

typedef struct tagQ8Registers
{
    uint32_T interruptEnable;
    int32_T  interruptStatus;
```

```
uint32_T control;
uint32_T status;

union
{
    struct
    {
        uint32_T preload;
        uint32_T value;
    } sq;

    struct
    {
        uint32_T preloadLow;
        uint32_T preloadHigh;
    } pwm;
} counter;

union
{
    struct
    {
        uint32_T preload;
        uint32_T value;
    } sq;

    struct
    {
        uint32_T preloadLow;
        uint32_T preloadHigh;
    } pwm;
} watchdog;

uint32_T counterControl;
uint32_T digitalIO;
uint32_T digitalDirection;

union
{
    struct {
        int16_T adc03;
        int16_T adc47;
    } one;

    int32_T two;      /* on a read */
    int32_T select;   /* on a write, channel selection */
} analogInput;

union
{
    struct {
        uint8_T enc0;
```

```
        uint8_T enc2;
        uint8_T enc4;
        uint8_T enc6;

        uint8_T enc1;
        uint8_T enc3;
        uint8_T enc5;
        uint8_T enc7;
    } one;

    uint8_T encs[8];

    struct {
        uint16_T enc02;
        uint16_T enc46;

        uint16_T enc13;
        uint16_T enc57;
    } two;

    struct {
        uint32_T enc0246;   /* ENCODER_DATA_A */
        uint32_T enc1357;   /* ENCODER_DATA_B */
    } four;

    uint64_T all;
} encoderData;

union
{
    struct {
        uint8_T enc0;
        uint8_T enc2;
        uint8_T enc4;
        uint8_T enc6;

        uint8_T enc1;
        uint8_T enc3;
        uint8_T enc5;
        uint8_T enc7;
    } one;

    uint8_T encs[8];

    struct {
        uint16_T enc02;
        uint16_T enc46;

        uint16_T enc13;
        uint16_T enc57;
    } two;
```

```
        struct {
            uint32_T enc0246;   /* ENCODER_CONTROL_A */
            uint32_T enc1357;   /* ENCODER_CONTROL_B */
        } four;

        uint64_T all;
    } encoderControl;

    union
    {
        struct {
            uint16_T dac0;
            uint16_T dac4;

            uint16_T dac1;
            uint16_T dac5;

            uint16_T dac2;
            uint16_T dac6;

            uint16_T dac3;
            uint16_T dac7;
        } one;

        uint16_T dacs[8];

        struct {
            uint32_T dac04;  /* DAC_CONTROL_A */
            uint32_T dac15;  /* DAC_CONTROL_B */
            uint32_T dac26;  /* DAC_CONTROL_C */
            uint32_T dac37;  /* DAC_CONTROL_D */
        } two;

        uint32_T pairs[4];

        struct
        {
            uint64_T dac0415;
            uint64_T dac2637;
        } four;
    } analogOutput;

    union
    {
        struct {
            uint16_T dac03;
            uint16_T dac47;
        } four;

        uint32_T all;
    } analogUpdate;
```

```
    uint32_T reserved2[6];

    union
    {
        struct {
            uint16_T dac03;
            uint16_T dac47;
        } four;

        uint32_T all;
    } analogMode;

    union
    {
        struct {
            uint16_T dac03;
            uint16_T dac47;
        } four;

        uint32_T all;
    } analogModeUpdate;

    uint32_T reserved3[227];

} Q8Registers, * volatile PQ8Registers;

#if defined(_MSC_VER)
#pragma pack(pop, registers)
#endif

/* Interrupt Status Register bit definitions */

#define INT_PEND       0x80000000   /* Card has interrupt pending */
#define INT_EXTINT     0x00800000   /* Edge detected on EXT_INT line */
#define INT_FUSE       0x00400000   /* Fuse blown */
#define INT_WATCHDOG   0x00200000   /* Watchdog counter expired */
#define INT_CNTROUT    0x00100000   /* Counter expired */
#define INT_ADC47RDY   0x00080000   /* A/D channels in 4-7 converted */
#define INT_ADC03RDY   0x00040000   /* A/D channels in 0-3 converted */
#define INT_ADC47EOC   0x00020000   /* A/D channel in 4-7 converted */
#define INT_ADC03EOC   0x00010000   /* A/D channel in 0-3 converted */
#define INT_ENC7FLG2   0x00008000   /* Encoder channel 7 flag 2 int. */
#define INT_ENC7FLG1   0x00004000   /* Encoder channel 7 flag 1 int. */
#define INT_ENC6FLG2   0x00002000   /* Encoder channel 6 flag 2 int. */
#define INT_ENC6FLG1   0x00001000   /* Encoder channel 6 flag 1 int. */
#define INT_ENC5FLG2   0x00000800   /* Encoder channel 5 flag 2 int. */
#define INT_ENC5FLG1   0x00000400   /* Encoder channel 5 flag 1 int. */
#define INT_ENC4FLG2   0x00000200   /* Encoder channel 4 flag 2 int. */
#define INT_ENC4FLG1   0x00000100   /* Encoder channel 4 flag 1 int. */
#define INT_ENC3FLG2   0x00000080   /* Encoder channel 3 flag 2 int. */
#define INT_ENC3FLG1   0x00000040   /* Encoder channel 3 flag 1 int. */
#define INT_ENC2FLG2   0x00000020   /* Encoder channel 2 flag 2 int. */
```

```
#define INT_ENC2FLG1   0x00000010   /* Encoder channel 2 flag 1 int. */
#define INT_ENC1FLG2   0x00000008   /* Encoder channel 1 flag 2 int. */
#define INT_ENC1FLG1   0x00000004   /* Encoder channel 1 flag 1 int. */
#define INT_ENC0FLG2   0x00000002   /* Encoder channel 0 flag 2 int. */
#define INT_ENC0FLG1   0x00000001   /* Encoder channel 0 flag 1 int. */

/* Control Register bit definitions */

/* 0 = use COUNTER to trigger auto conversions, 1 = use CNTR_EN */
#define CTRL_CNTRENCV   0x10000000
/* 0 = deactivate watchdog features of EXT_INT, 1 = activate */
#define CTRL_EXTACT     0x08000000
/* 0 = EXT_INT active low, 1 = EXT_INT active high */
#define CTRL_EXTPOL     0x04000000
/* 0 = double-buffer DAC47, 1 = transparent mode on DAC47 */
#define CTRL_DAC47TR    0x02000000
/* 0 = double-buffer DAC03, 1 = transparent mode on DAC03 */
#define CTRL_DAC03TR    0x01000000
/* 0 = no conversions on ADC47, 1 = start conversions on ADC47 */
#define CTRL_ADC47CV    0x00800000
/* 0 = full power mode, 1 = A/D standby mode */
#define CTRL_ADCSTBY    0x00400000
/* 0 = manual conversions on ADC47, 1 = use auto conversions */
#define CTRL_ADC47CT    0x00200000
/* 0 = Control Register selects channels, 1 = A/D Register does */
#define CTRL_ADC47HS    0x00100000
/* 0 = do not include channel 7, 1 = include channel 7 */
#define CTRL_ADCSL7     0x00080000
/* 0 = do not include channel 6, 1 = include channel 6 */
#define CTRL_ADCSL6     0x00040000
/* 0 = do not include channel 5, 1 = incldue channel 5 */
#define CTRL_ADCSL5     0x00020000
/* 0 = do not include channel 4, 1 = include channel 4 */
#define CTRL_ADCSL4     0x00010000
/* 0 = no conversions on ADC03, 1 = start conversions on ADC03 */
#define CTRL_ADC03CV    0x00008000
                     /* 0x00004000  // reserved. Set to zero. */
/* 0 = manual conversions on ADC03, 1 = use auto conversions */
#define CTRL_ADC03CT    0x00002000
/* 0 = Control Register selects channels, 1 = A/D Register does */
#define CTRL_ADC03HS    0x00001000
/* 0 = do not include channel 3, 1 = include channel 3 */
#define CTRL_ADCSL3     0x00000800
/* 0 = do not include channel 2, 1 = include channel 2 */
#define CTRL_ADCSL2     0x00000400
/* 0 = do not include channel 1, 1 = include channel 1 */
#define CTRL_ADCSL1     0x00000200
/* 0 = do not include channel 0, 1 = include channel 0 */
#define CTRL_ADCSL0     0x00000100
/* 0 = disable channel 7 index pulse, 1 = enable channel 7 index */
#define CTRL_ENC7IDX    0x00000080
/* 0 = disable channel 6 index pulse, 1 = enable channel 6 index */
```

```
#define CTRL_ENC6IDX    0x00000040
/* 0 = disable channel 5 index pulse, 1 = enable channel 5 index */
#define CTRL_ENC5IDX    0x00000020
/* 0 = disable channel 4 index pulse, 1 = enable channel 4 index */
#define CTRL_ENC4IDX    0x00000010
/* 0 = disable channel 3 index pulse, 1 = enable channel 3 index */
#define CTRL_ENC3IDX    0x00000008
/* 0 = disable channel 2 index pulse, 1 = enable channel 2 index */
#define CTRL_ENC2IDX    0x00000004
/* 0 = disable channel 1 index pulse, 1 = enable channel 1 index */
#define CTRL_ENC1IDX    0x00000002
/* 0 = disable channel 0 index pulse, 1 = enable channel 0 index */
#define CTRL_ENC0IDX    0x00000001

/* 0 = use internal clock for ADC47, 1 = use common clock */
#define CTRL_ADC47SCK   CTRL_ADCSL5
/* 0 = use internal clock for ADC03, 1 = use common clock */
#define CTRL_ADC03SCK   CTRL_ADCSL1

#define CTRL_ADC03MASK  0x0000ff00  /* bitmask extract ADC03 bits */
#define CTRL_ADC47MASK  0x00ff0000  /* bitmask extract ADC47 bits */
#define CTRL_ADCMASK    (CTRL_ADC03MASK | CTRL_ADC47MASK)

/* Status Register bit definitions */

#define STAT_CNTREN     0x01000000
#define STAT_EXTINT     0x00800000
#define STAT_FUSE       0x00400000
#define STAT_ADC47FST   0x00200000
#define STAT_ADC03FST   0x00100000
#define STAT_ADC47RDY   0x00080000
#define STAT_ADC03RDY   0x00040000
#define STAT_ADC47EOC   0x00020000
#define STAT_ADC03EOC   0x00010000
#define STAT_ENC7FLG2   0x00008000
#define STAT_ENC7FLG1   0x00004000
#define STAT_ENC6FLG2   0x00002000
#define STAT_ENC6FLG1   0x00001000
#define STAT_ENC5FLG2   0x00000800
#define STAT_ENC5FLG1   0x00000400
#define STAT_ENC4FLG2   0x00000200
#define STAT_ENC4FLG1   0x00000100
#define STAT_ENC3FLG2   0x00000080
#define STAT_ENC3FLG1   0x00000040
#define STAT_ENC2FLG2   0x00000020
#define STAT_ENC2FLG1   0x00000010
#define STAT_ENC1FLG2   0x00000008
#define STAT_ENC1FLG1   0x00000004
#define STAT_ENC0FLG2   0x00000002
#define STAT_ENC0FLG1   0x00000001

/* Counter Control Register bit definitions */
```

```
/* 0 = no load, 1 = load Watchdog from active preload and WDOGVAL */
#define CCTRL_WDOGLD    0x02000000
/* value of watchdog counter output (ignored if WDOGLD = 0) */
#define CCTRL_WDOGVAL   0x01000000
/* 0 = deactive watchdog features of WATCHDOG counter, 1 = activate */
#define CCTRL_WDOGACT   0x00800000
/* 0 = WATCHDOG output is INT_WATCHDOG, 1 = is Watchdog counter out */
#define CCTRL_WDOGSEL   0x00400000
/* 0 = disable WATCHDOG output (output always high),
   1 = enable WATCHDOG output. Value determined by WDOGSEL */
#define CCTRL_WDOGOUTEN 0x00200000
/* 0 = use Watchdog Preload Low Register,
   1 = use Watchdog Preload High Register. Ignored if WDOGMODE = 1. */
#define CCTRL_WDOGPRSEL 0x00100000
/* 0 = use watchdog register set #0 for writes to preload registers,
   1 = use register set #1 */
#define CCTRL_WDOGWSET  0x00080000
/* 0 = use watchdog register set #0 for active set and reads,
   1 = use register set #1 */
#define CCTRL_WDOGRSET  0x00040000
/* 0 = square wave mode (WDOGPRSEL selects preload register),
   1 = PWM mode (both preload low and high registers used) */
#define CCTRL_WDOGMODE  0x00020000
/* 0 = disable counting of watchdog counter, 1 = enable */
#define CCTRL_WDOGEN    0x00010000

/* 0 = no load, 1 = load counter from active preload and CNTRVAL */
#define CCTRL_CNTRLD    0x00000200
/* value of counter output (ignored if CNTRLD = 0) */
#define CCTRL_CNTRVAL   0x00000100
/* 0 = CNTREN active high (CNTRENCV=0) or falling edge (CNTRENCV=1),
   1 = opposite */
#define CCTRL_CNTRENPOL 0x00000040
/* 0 = disable CNTR_OUT output (output always high),
   1 = enable CNTR_OUT output. Value is output of COUNTER */
#define CCTRL_CNTROUTEN 0x00000020
/* 0 = use Counter Preload Low Register,
   1 = use Counter Preload High Register. Ignored if CNTRMODE = 1 */
#define CCTRL_CNTRPRSEL 0x00000010
/* 0 = use counter register set #0 for writes to preload registers,
   1 = use register set #1 */
#define CCTRL_CNTRWSET  0x00000008
/* 0 = use counter register set #0 for active set and reads,
   1 = use register set #1 */
#define CCTRL_CNTRRSET  0x00000004
/* 0 = square wave mode (CNTRPRSEL selects preload register),
   1 = PWM mode (both preload low and high registers used) */
#define CCTRL_CNTRMODE  0x00000002
/* 0 = disable counting of counter, 1 = enable counter */
#define CCTRL_CNTREN    0x00000001
```

```
/* Encoder Control Byte bit definitions */
#define ENC_ONE_CHANNEL         0x00    /* operate on one channel */
#define ENC_BOTH_CHANNELS       0x80    /* operate on even and odd */

#define ENC_RLD_REGISTER        0x00    /* select RLD register */

#define ENC_RLD_RESET_BP        0x01    /* reset byte pointer (BP) */

#define ENC_RLD_RESET_CNTR      0x02    /* reset count value (CNTR) */
#define ENC_RLD_RESET_FLAGS     0x04    /* reset BT, CT, CPT, S */
#define ENC_RLD_RESET_E         0x06    /* reset error flag (E) */

#define ENC_RLD_SET_CNTR        0x08    /* CNTR = PR */
#define ENC_RLD_GET_CNTR        0x10    /* OL = CNTR */
#define ENC_RLD_SET_PSC         0x18    /* PSC = PR(0) */

#define ENC_CMR_REGISTER        0x20    /* select CMR register */

#define ENC_CMR_BINARY          0x00    /* binary count mode */
#define ENC_CMR_BCD             0x01    /* BCD count mode */

#define ENC_CMR_NORMAL          0x00    /* normal counting */
#define ENC_CMR_RANGE           0x02    /* range limit counting */
#define ENC_CMR_NONRECYCLE      0x04    /* non-recycle counting */
#define ENC_CMR_MODULO          0x06    /* modulo-N counting */

#define ENC_CMR_NONQUADRATURE   0x00    /* non-quadrature mode */
#define ENC_CMR_QUADRATURE_1X   0x08    /* quadrature 1X mode */
#define ENC_CMR_QUADRATURE_2X   0x10    /* quadrature 1X mode */
#define ENC_CMR_QUADRATURE_4X   0x18    /* quadrature 1X mode */

#define ENC_IOR_REGISTER        0x40    /* select IOR register */

#define ENC_IOR_DISABLE_AB      0x00    /* disable A and B inputs */
#define ENC_IOR_ENABLE_AB       0x01    /* enable A and B inputs */

#define ENC_IOR_LCNTR_LOAD      0x00    /* I/LD pin is Load CNTR */
#define ENC_IOR_LCNTR_LATCH     0x02    /* I/LD pin is Latch CNTR */

#define ENC_IOR_CARRY_BORROW    0x00    /* FLG1=CARRY, FLG2=BORROW */
#define ENC_IOR_COMPARE_BORROW  0x08    /* FLG1=COMPARE, FLG2=BORROW */
#define ENC_IOR_CARRY_UPDOWN    0x10    /* FLG1=CARRY/BORROW,FLG2=U/D*/
#define ENC_IOR_INDEX_ERROR     0x18    /* FLG1=IDX, FLG2=E */

#define ENC_IDR_REGISTER        0x60    /* select IDR register */

#define ENC_IDR_DISABLE_INDEX   0x00    /* Disable index */
#define ENC_IDR_ENABLE_INDEX    0x01    /* Enable index */

#define ENC_IDR_NEG_INDEX       0x00    /* Negative index polarity */
#define ENC_IDR_POS_INDEX       0x02    /* Positive index polarity */
```

```
#define ENC_IDR_LCNTR_INDEX     0x00    /* LCNTR/LOL pin is indexed */
/* Analog Mode bit definitions */

#define DAC0_UNIPOLAR_10V       0x0000
#define DAC0_BIPOLAR_5V         0x0080
#define DAC0_BIPOLAR_10V        0x0880
#define DAC1_UNIPOLAR_10V       0x0000
#define DAC1_BIPOLAR_5V         0x0040
#define DAC1_BIPOLAR_10V        0x0440
#define DAC2_UNIPOLAR_10V       0x0000
#define DAC2_BIPOLAR_5V         0x0020
#define DAC2_BIPOLAR_10V        0x0220
#define DAC3_UNIPOLAR_10V       0x0000
#define DAC3_BIPOLAR_5V         0x0010
#define DAC3_BIPOLAR_10V        0x0110

#define DAC4_UNIPOLAR_10V       0x00000000
#define DAC4_BIPOLAR_5V         0x00800000
#define DAC4_BIPOLAR_10V        0x08800000
#define DAC5_UNIPOLAR_10V       0x00000000
#define DAC5_BIPOLAR_5V         0x00400000
#define DAC5_BIPOLAR_10V        0x04400000
#define DAC6_UNIPOLAR_10V       0x00000000
#define DAC6_BIPOLAR_5V         0x00200000
#define DAC6_BIPOLAR_10V        0x02200000
#define DAC7_UNIPOLAR_10V       0x00000000
#define DAC7_BIPOLAR_5V         0x00100000
#define DAC7_BIPOLAR_10V        0x01100000

/* Conversion factors */
#define ADC_FACTOR        (10.0 / 8192)   /* analog input to volts */
#define DAC_BIPOLAR_ZERO  (0x0800)        /* analog value for 0V */

/* Register offsets from base address (equivalent to Q8Registers) */

#define Q8_INTERRUPT_ENABLE_REGISTER(base)      ((base) + 0x00)
#define Q8_INTERRUPT_STATUS_REGISTER(base)      ((base) + 0x04)
#define Q8_CONTROL_REGISTER(base)               ((base) + 0x08)
#define Q8_STATUS_REGISTER(base)                ((base) + 0x0C)
#define Q8_COUNTER_PRELOAD_REGISTER(base)       ((base) + 0x10)
#define Q8_COUNTER_REGISTER(base)               ((base) + 0x14)
#define Q8_COUNTER_PRELOAD_LOW_REGISTER(base)   ((base) + 0x10)
#define Q8_COUNTER_PRELOAD_HIGH_REGISTER(base)  ((base) + 0x14)
#define Q8_WATCHDOG_PRELOAD_REGISTER(base)      ((base) + 0x18)
#define Q8_WATCHDOG_REGISTER(base)              ((base) + 0x1C)
#define Q8_WATCHDOG_PRELOAD_LOW_REGISTER(base)  ((base) + 0x18)
#define Q8_WATCHDOG_PRELOAD_HIGH_REGISTER(base) ((base) + 0x1C)
#define Q8_COUNTER_CONTROL_REGISTER(base)       ((base) + 0x20)
#define Q8_DIGITAL_IO_REGISTER(base)            ((base) + 0x24)
#define Q8_DIGITAL_DIRECTION_REGISTER(base)     ((base) + 0x28)
#define Q8_AD_REGISTER(base)                    ((base) + 0x2C)
#define Q8_ENCODER_DATA_REGISTER_A(base)        ((base) + 0x30)
```

```
#define Q8_ENCODER_DATA_REGISTER_B(base)         ((base) + 0x34)
#define Q8_ENCODER_CONTROL_REGISTER_A(base)      ((base) + 0x38)
#define Q8_ENCODER_CONTROL_REGISTER_B(base)      ((base) + 0x3C)
#define Q8_DA_OUTPUT_REGISTER_A(base)            ((base) + 0x40)
#define Q8_DA_OUTPUT_REGISTER_B(base)            ((base) + 0x44)
#define Q8_DA_OUTPUT_REGISTER_C(base)            ((base) + 0x48)
#define Q8_DA_OUTPUT_REGISTER_D(base)            ((base) + 0x4C)
#define Q8_DA_UPDATE_REGISTER(base)              ((base) + 0x50)
                                                /* 0x54-0x6B reserved */
#define Q8_DA_MODE_REGISTER(base)                ((base) + 0x6C)
#define Q8_DA_MODE_UPDATE_REGISTER(base)         ((base) + 0x70)
                                                /* 0x74-0x3FF reserved */

#define Q8_SIZEOF_ADDRESS_SPACE                   0x0400 /* bytes */

#endif
```

Index

Index